Testing Exercise

— based on an old exam question (Spring 2011) —

In this task you should test the method `maximalBorderLengths` (see Listing 1).

Please note the following:

- `&&` is a short-circuit operator; i.e. the right-hand boolean expression will not be evaluated if the left-hand expression evaluates to `false`.
- After creating an `int[]` array all elements are initialized to `0`.
- The ith character of the string `s` can be accessed using `s[i]`
- You are not required to write full-blown unit tests. However, you should provide both required inputs and expected outputs for each test.

Your tasks:

1. Draw a Control Flow Graph (CFG) for the method `maximalBorderLengths`.
2. Branch coverage
   (a) Provide tests for the method `maximalBorderLengths` that guarantee 100% branch coverage.
   (b) What is the minimal number of tests that are necessary to guarantee 100% branch coverage?
   (c) Can you achieve statement coverage with fewer tests?
3. Loop coverage
   (a) Provide tests for method `maximalBorderLengths` that guarantee 100% loop coverage.
   Hint: For an inner loop, you need to test that: for some iteration of the outer loop, the inner loop is executed as required by the loop coverage criterion for a single non-nested loop.

4. DU-Pair coverage
   (a) Provide all DU-pairs for the local variable `borderLength` and write down the details of the analysis necessary to compute them.
   (b) Provide tests for method `maximalBorderLengths` that maximize DU-Pair coverage for the variable `borderLength`.
   (c) If you are not able to reach 100% DU-Pair coverage for the variable `borderLength`, provide the infeasible DU-pairs.
5. Bug detection: Suppose we introduce a bug in method `maximalBorderLengths` by dropping Line 45. For each coverage criterion (branch coverage, loop coverage, and DU-Pair coverage for the variable `borderLength`) determine whether this bug will be detected, if we run any test suite that guarantees maximal coverage with respect to that particular criterion.

```
/// <summary>
/// Calculate the maximal border length for each prefix of 'pattern'.
///
/// The border of a string is a proper prefix that is also a suffix.
/// The maximal border length of the empty string is defined to be -1.
/// </summary>
/// <example>
/// For example, for the string "abcab" it should return {-1, 0, 0, 0, 1, 2}, where
/// -1 is the maximal border length of the prefix "", 0 is the maximal border length of
/// the prefix "a", and so on.
/// </example>
/// <param name="pattern">string to be analyzed</param>
/// <returns>maximal border lengths for each prefix of 'pattern'</returns>
/// <exception cref="ArgumentNullException">
///   Thrown when <paramref name="pattern"/> is equal to null.
/// </exception>
public static int[] maximalBorderLengths(string pattern) {
  if (pattern == null) {
    throw new ArgumentNullException("pattern");
  }

  int pi, borderLength;
  pi = 0;
  borderLength = -1;

  int[] borderLengths = new int[pattern.Length + 1];
  borderLengths[0] = -1;

  // We iterate over each prefix of 'pattern' and compute its maximal border length.
  while (pi < pattern.Length) {
    while (0 <= borderLength && pattern[pi] != pattern[borderLength]) {
      // The last character of the current prefix (pattern.substring(0, pi + 1))
      // doesn't match the last character of the current border
      // (pattern.substring(0, borderLength + 1).

      // The next shorter border that might work is the maximal border of the
      // current border.
      borderLength = borderLengths[borderLength];
    }

    // We can extend the current border, because the last character of the current
    // prefix matches the last character of the current border or the current
    // border length went negative.
    pi = pi + 1;
    borderLength = borderLength + 1;
    borderLengths[pi] = borderLength;
  }

  return borderLengths;
}
```

Listing 1: `maximalBorderLengths`