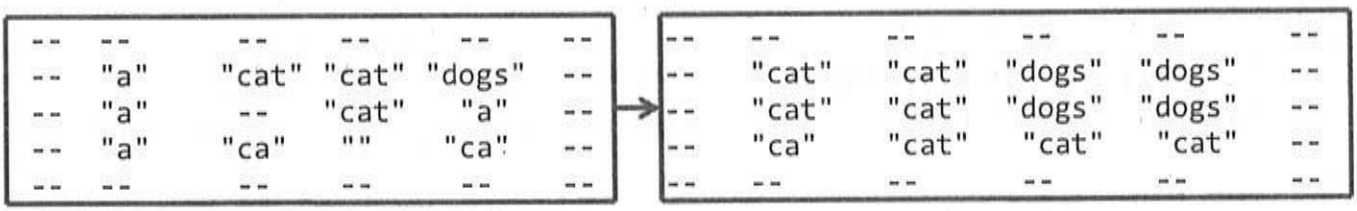
7. **Higher Order Functions and Arrays (8 Points).**

For this problem, you'll develop a method `step()` that takes a 2D array of strings and replaces every string with its longest neighbor, EXCEPT the edges, which you will assume are always null and should never change. The neighbors of a string are the eight strings surrounding it (including diagonals). For example, if given the 2D array on the left, `step` will return the array on the right. We use two dashes `--` to represent a null pointer. The `""` in row 4, column 3 is an empty string.




For the sake of allowing easy customization, your program will compare strings using an object that implements the `NullSafeStringComparator` interface defined below.

```
public interface NullSafeStringComparator {
    /** Returns a negative number if s1 is 'less than' s2, 0 if 'equal',
      * and a positive number otherwise. Null is considered less than
      * any String. If both inputs are null, return 0. */
    public int compare(String s1, String s2);
}
```

(a) Write a new class `LengthComparator` that implements `NullSafeStringComparator`. The `LengthComparator` should compare strings based on their lengths. The length of a string s can be retrieved using `s.length()`. Do not provide a constructor, as it is unnecessary.

see page 3.

(b) Complete the helper function `max`, which returns the maximum string of a 1D array using the `StringComparator sc` to judge the string. **You can do this part even if you skipped part a!**

```
public static String max(String[] a, NullSafeStringComparator sc) {
    String maxStr = a[0];
    for(int i = 1          ; i < a.length ; i += 1) {
    if (sc.compare(maxStr, a[i]) < 0                ) {
            maxStr = a[i]                ;
        }
    }
    return maxStr; } // bracket on this line for vertical space reasons
```

(c) Complete the `step` function so that it completes the task described on the previous page. You may assume that every row has the same number of columns, that the size is at least 3x3, and that the edges are all null (as in the figure on the previous page). You may **not** assume that the number of rows is equal to the number of columns. **You may not add any semicolons. You may not use the ++ or -- operators. Use only the blanks provided. You can complete this part even if you skipped parts a and b.**

```
public static String[][] step(String[][] arr) {
    /* Recall: All String references in stepped are null by
       default, so the edges are correct on initialization. */
    String[][] stepped = new String[arr.length][arr[0].length];

    for (int i = 1; i < arr.length - 1; i += 1) {
        for (int j = 1; j < arr[0].length - 1; j += 1) {
            String[] temp = new String[9];
            int idx = 0;
            for (int k = -1; k <= 1; k += 1) {
                for (int m = -1; m <= 1; m += 1) {
                    temp[idx] = arr[i+k][j+m];
                    idx = idx + 1;
                }
            }
            stepped[i][j] = max(temp);
        }
    }
    return stepped;
}
```

(a)

```
class LengthComparator() {
    public static int NullSafeStringComparator (String s1, String s2) {
        if (s1 == null && s2 == null) {
            return 0;
        } else if (s1 == null) {
            return -1;
        } else if (s2 == null) {
            return 1;
        } else {
            if (s1.length() > s2.length()) {
                return 1;
            } else if (s1.length() == s2.length()) {
                return 0;
            } else {
                return -1;
            }
        }
    }
```