Loop Testing

Even though we have good logic, errors can still creep into our code through typos or bad math.

Here is a general testing strategy that will catch most, but not all, errors.

Test first, test middle, test last.

Test 0, test 1, test many.

Recall the palindrom problem from last lecture: a string is a palindrome if it reads the same front to back. Also look at the algorithm in the sample code.

For palindromes, we must test palindromes of length 0, 1, and more than 1. Also note, that we will need to test odd and even lengths.

"", "a", "radar", "rabbar"

For non-palindromes, we need to test when the misplaced character comes as the first character of the string (test first), or the last character the loop checks (in this case "test last" really means test the middle characters),

or any other character (test middle):

"radam", "noan", "ragecar"

Designing Good Loops, part 1.

Loop example:

Write a method that determines if an English string is a palindrome.

- we will say that a string is a palindrome if

1) non-letters are ignored

2) letter capitalization does not matter

3) the remaining characters read the same forwards as backwards

This problem is a little harder than the ones done previously. We will first look at some possible algorithms, but it turns out that the best, and simplest, solution is not

going to be the first ones considered. Often, it is a good idea to think through the benefits and disadvantages of different possible solutions and to try to improve the solutions

before writing code.

Here are the solutions proposed by the students in lecture:

Proposed solution 1:

1) Create a new String (or StringBuilder) that changes all letters to upper (or lower) case.

2) Create another new String (or StringBuilder) that removes all non-letters.

3) Now use the original isPalindrome to check if the new String reads the same forwards as backwards.

Proposed solution 2:

1) Create two new Strings (or StringBuilders). In each String, all spaces are removed and all letters changed to upper case. Store the second String in reverse order.

2) Check that the two Strings are the same.

While these are the obvious solutions, and both are similar, neither is going to be the best.

What we want for a good algorithm:

- Time and space efficiency:

Do not use more memory that is needed, do not make more traversals of the String than needed.

(We have to be a little careful when counting traversals because two traversals that do one operation at each step is the same as one traversal that does two operations in each step.)

- Logical simplicity

The loop should do one logical task. If we need two different tasks, we should use separate loops for each.

Problems and advantages of the above solutions:

The solutions are simple with each loop doing one task.

The solutions create extra, unnecessary Strings.

The solutions use multiple traversals of the String.

Here is a much better soluton proposed in the class:

Keep track of the front and back index.

If either points to a non-letter, increment that index.

Otherwise, both point to letters so compare them just as in the isPalindrome.

Note: this solution uses no extra space and does a single traversal of the String.

The solution is logically simple because we are still doing one task: comparing each letter at the front to the corresponding letter at the rear.

The loop has a simple organization because each index will increment or decrement AT MOST once each iteration, and at least one will change.

Since either the front or end index changes, it is easy to logically reason that our loop will eventually terminate.

public static boolean isEnglishPalindrome(String s) {

int front = 0;

int back = s.length() - 1;

while (front < back) {

if (/\* the front is not a letter \*/)

front = front + 1;

else if (/\* the back is not a letter \*/)

back = back - 1;

else if (/\* the two letters match \*/) {

front = front + 1;

back = back - 1;

}

else {

return false;

}

}

return true;

}

Note the solution has a single traversal of the string, it does up to 3 steps in each traversal, and so it will be just as fast, and possibly faster, than the other two propsed algorithms.

As noted before, the solution uses much less memory.

The solution really does only one thing at each step, and just as importantly, the increments are simple.

At each step, we only do one thing: either we increment the front (not a letter), the back (not a letter), or both (both are letters and they match).