Eric Olsen

Assumptions

The primary task for this assignment was to create two classes that utilize recursive methods in order to generate some simple calculations. Our first class, TestQ1, implements two calculations. Convert1 takes a base 10 number and transitions it into a base system indicated by the user. One of our first assumptions is that our selectable bases are between 2 and 9. This program will not deal with numbers in higher base systems like hexadecimal. The second method takes a string representation of a number, a base that is between 2 and nine and returns its base ten representation Again, one of our assumptions is that our base systems are limited between two and nine.

The second class, TestQ2 tests for membership within an array. There are two methods in this program. We will first look for a searched value in an unsorted array by recursively splitting the searched areas into halves. Our second method will utilize binary search in order to continually split the array into halves and search the middle value. As an obvious point, we will need to work with a sorted array in order to utilize binary search. As an additional note, we will limit our searched array to 20 indices and our searched values to Integers. We will also utilizer helper methods in TestQ2 to recursively call a slightly altered version of the original calling method.

Main Design

I will begin by describing the design decisions I made for the class TestQ1. The entire program is contained in a single class, so there is no complex class hierarchy to analyze. TestQ1 is also a GUI generated by the java Swing class. It is not a very visually appealing. The convert1 method occupies the top half of the GUI and the convert2 method occupies the bottom. Due to time constraints I did not build any key listeners into the GUI textfields and clicking the button is the only way to execute the convert1 and convert2 methods. There is also some number verification worked into the textfields which I will describe later.

More interesting than the GUI are the two recursive methods involved. In the case of convert1, we are taking a base10 number from the top textfield and its new base in the second textfield. In the event that our base 10 number is less than its base system, the method simply returns the base 10 number. For example, a two in base ten is also a two in octal numbers. When the decimal number is more than its new base, we have to extract two pieces of information: the number of times this base ten number goes into the new base and the number leftover. We do this by conducting a modulus operation from our base ten number by the new base. We then add this number to the recursive call of our decimal number divided by the base and multiplied by ten.

Convert2 is the combination of a String parsing method and recursive call. It returns any string that is a single digit since we can easily represent this in base ten. When there are remaining digits to process, we shrink the array by one index from the right and use our parsed integer for our recursive call. The remaining string is passed as a parameter to a recursive call and is also multiplied by its base so the value can continue to grow.

TestQ2 was a bit more complicated to implement, but was also uses a console driver program instead of a GUI. For this program, a series of inner and outer loops drive the user input for the array searches. As a general overview, I randomly generated an unsorted array and sorted array. Both arrays should not repeat any numbers and will be between -100 and 100. The first recursive method, findValueUnsortedArray, took me over 7 days to figure out. Helper methods were key to solving this problem. In this case the initial call takes an unsorted integer array and a searched integer value. If this value is in index zero, the call returns a zero. If the value is not found, this method calls its helper method and passes the unsorted array, the index zero, a length integer decremented by one, and the searched value. The helper method then takes these parameters and runs a similar process. It will search the first and last index. If the value is not found it will recursively call itself with a left half and right half specified as the beginning and end indices. In the event that nothing is found, the method returns a -1. Finally, a Math.max operation returns the greater of the left or right; this works because the unfound search should return a -1.

The findValueSortedArray utilizes a binary search to locate the searched item. The array is already sorted and our initial method takes both the sorted integer array and the integer search value. We do not even bother checking the middle value of the first call but instead call our altered version of the method which also includes a starting and ending index. Within the helper method we first calculate the middle index by averaging the first and last index number. Before we do

anything else we also check to see if the end value was decremented below the starting value in the last recursive call. If so we return a -1. After this we check the middle value to see if it matches the searched value. If it does, we return that index value. If the middle value is less than the searched value, we recursively call the helper method with the middle index as the last index. If the middle value is greater than the searched value, we recursively call the helper method with the middle index as its starting index.

As far as Big-O considerations go, Binary Search is a faster operation. Since we are cutting the size of the searched array in half during every operation, our Big-O should equal O(Logn). On the other hand, our recursive call to continually sub-divide the unsorted array will continue to execute even after it locates the searched item. When evaluating the worst case scenario, our unsorted array search will give us BigO(n) plus the initial query, so BigO(n+1). As input size grows, the difference between O(LogN) and O(N) can be quite significant. However, the real question is how expensive is it to sort the array values? In this assignment we just assumed that the array was already sorted, but in a production environment we should also calculate the cost of sorting every item before deciding between two methods like these.

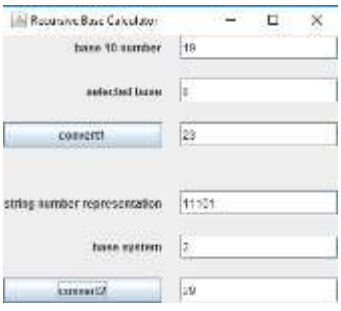
Error Handling

The only significant error checking included in both classes is a isNumber boolean method which returns true when a String successfully parses an integer from our GUI or console. In both cases this cleans up the data for the calling methods. There was also one conditional check included in TestQ1 which forces the user to use a base number between 2 and 9.

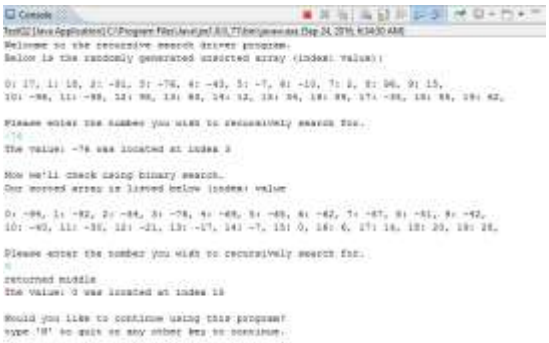
Test Plan: This test plan evaluated the assignment examples, proper use, and also user errors.

Test Case #	Input	Description	Expected Output	Actual Output	Pass / Fail
1	TestQ1: Convert1(19, 8)	Best case (example)	23	23	pass
2	TestQ1: Convert2("11101",2)	Best case (example)	29	29	pass
3	TestQ1: Convert1(255,2)	Proper use	11111111	11111111	pass
4	TestQ1: Convert2(11111111,2)	Proper use	255	255	pass
5	TestQ1: Convert 1(red, &)	Character input error	Error message	Error message	pass
6	TestQ1: Convert1(100, 16)	Base out of range	Error message	Error message	pass
7	TestQ1: Convert2 (red, &)	Character input error	Error message	Error message	Pass
8	TestQ1: Convert2(578, 16)	Base out of range	Error message	Error message	Pass
9	TestQ2: unsortedArray(-76)	Find an existing element	found	found	Pass
10	TestQ2: sortedArray(0)	Find an exsiting element	found	found	Pass
11	TestQ2: unsortedArray (101)	Look for non-existent element	Not found	Not found	Pass
12	TestQ2: sortedArray(101)	Look for non-existent element	Not found	Not found	Pass
13	TestQ2: unsortedArray(red)	Improper user input	User re-inputs	User re-inputs	Pass
14	TestQ2: sortedArray(red)	Improper user input	User re-inputs	User re-inputs	Pass

Test Case1-2



Test Cases 9-10



Lessons Learned: Helper methods were key in this assignment. Calling a slightly altered version of a method is a great deal easier than mathematically manipulating these methods through all their recursive calls. I also once again found that boolean values work great to control user error.