Vishal Yathish
506-166-275
14 October 2024

Programming Project #2 – Recursion

PART 1: PROJECT IMPLEMENTATION

I started this report with the *modulo ()* method, which is supposed to return the remainder value between two given integers *m* and *n*. I started this method by identifying the base case: when the value of *m/n* (which will return only the truncated integer value) is 0, meaning that *m* is less than *n*, return *m*. I wrote this as the first condition of an if statement, with the recursive case in the else statement. In all other cases, where the value of the division is not 0, then we were to set the *m* value to be $m = m - n*div$, where *div* is the value *m/n*, and call the method again with the new m parameters. At first, this implementation gave an error, as the method was supposed to return an integer and there was no default return statement; I arbitrarily set it to 0, thinking that this would never come up since *div* would eventually hit 0 and the if statement would be called. While this implementation worked for the first few test cases, it failed the third one *modulo* (12,5) == 2. I ran through the math on paper, and it seemed to work. I then tried printing out the values of m, n, and div for that test case, and everything seemed to be accurate at every step.

Step 1: $m = 12$, $n = 5$, $div = 2$
Step 2: $m = 2$, $n = 5$, $div = 0$

Eventually, I realized that the problem was that even though *div* hit 0, the if statement did not register, meaning that the function simply passed it to its default return value of 0. I solved this by modifying the function so that the default return would be *m*; I then restructured the method entirely, to get rid of the redundant first return statement, and had the recursive call be in the main if statement, where *div* is not equal to 0. This implementation seemed to work for all my current test cases.

The next method I completed was *occurrences ()*, which returns the number of times a digit *d* (between 0 and 9 inclusive) is repeated within a larger integer, *number*. My first instinct was to use the *modulo ()* method I had just implemented; *modulo (number, 10)* should give the value of the final digit in the number, which we can then compare to *d*. We could then use recursion to iterate over the *number* and add one to our output every time there was a match. For my base case, I set when *number* is 0, since there can only be two possibilities there: either *d* is 0 and we need to return 1, or *d* is some other number and we need to return 0. For the recursive case, I created a variable to store the current digit of *number*, which I found as described, and every time is was equal to *d*, I added 1 to an output variable (which I had initialized to 0 at the beginning of the program). I would then call the method again, with the *number* parameter set to *number*/10. The method would then finally return my output variable.

This implementation failed the first test case. This was because I did not realize that the method reset the output variable to 0 every time the function called itself. Also, even if it did, the fact that the base case returned a value other than the output variable made the entire thing pointless –

quite embarrassing mistakes on my part. I could not pass the output variable as a parameter of the method, since I wasn't sure I was allowed to do that. So, I solved this problem by having the method return 1+*occurrences* (*number*/10, *d*) if the digit matched, and return *occurrences* (*number*/10, *d*) if it did not. This would iterate the result by 1 every time there was a match, until it finally reached the base case and either added 1 or 0. This implementation passed all preliminary tests.

The next method I worked on was *lucky7s ()*. This method returns a string where repeated (consecutive) characters in a given string *s* get separated by the string "77". I utilized a similar structure to the previous *occurrences* method; if the first character of the string was the same as the second character, then the method would return the first character + "77" + the recursive call (with the remainder of the string passed as its new parameter). If it was not the same, then it would return the same value, except without the "77." I ran into a few issues here. First, I couldn't concatenate *s* [0] with the rest of the string since it was not a string object. I had to use a string constructor to turn it into a string of length 1 before concatenation. Second, when initially writing this method, I wrote it how I would in Python, with the remainder of the string (minus the first character) being *s* [1:]. This does not work in C++, but I looked up how to properly utilize the *substr* () method and solved this rather quickly. The base case of my method was when the input string was empty (""); this would mean that the function had traversed the entire string. This iteration of my method passed all the preliminary test cases.

The last problem was the most difficult for me to solve. At first, I tried putting a recursive call-in a for loop to try and get every possible subset of the array. That logic didn't work, and it ended up causing a stack overflow. Then I tried creating a vector to hold all possible subsets; not only did that not work, but Howard also made it clear that he did not want us to utilize them, and that no for loops in the code were permitted. At this point, I was rather confused, so I went to office hours and Howard was able to point me the right direction.

The goal was to utilize a method like *lucky7s ()*, but instead of traversing the array from the start to the end, we want to go from the end to the front. The second hint Howard provided was that we are passing in a size parameter, which tells the programming the upper bound of the array. If we decreased that upper-bound, then the program would only read from the first element to the upper bound and ignore anything after it. This means that when passing into the recursive call, we wouldn't need to initialize brand new sub-arrays, which I had been doing beforehand. The logic for the program is that we take the last element of the array, and either subtract it from the target value or don't (either the element is included in a combination that adds up to the target or it isn't). Then continue until it reaches the front of the array. If by the end, the target is equal to 0, that means that some combination of values in the array is equal to the initial target – the program would return true. If the target is never 0, then this is not the case – the program would return false. The base case for the method was when the size parameter is equal to 0; this would mean that the method had traversed the entire array. If target == 0, then return true. If target != 0, then return false. In the recursive case, we have two function calls: when the final element is included and when it is not included. Both return Boolean values, so they are stored as temporary variables: bool1 and bool2. If either of these two values is true, we return true. If neither are true, we return false.

PART 2: PROJECT TESTING & DEBUGGING

With most of the project done, I moved on to more thorough testing. All my methods passed the initial test cases that Howard provided; they also passed a variety of extra tests, such as increasing the size of the input data, etc. However, issues did arise with two methods that required debugging: *occurrences ()* and *combinations ()*.

Combinations () was the easier of the two to fix.

The test case I set it was the following:

```
int array2[10] = {7,6,4,1,5,9,2,8,3,10};
assert (!combinations(array2,8,0));
```

Given a populated array, with a target of 0, the result should be false since no combination of positive integers will add up to 0. My method initially resulted in true. The reason for this was because the method was traversing the entire array, and eventually got to the front, and checked whether the target element was zero (for all combinations of values). Because the target was always zero, it thus registered as true. I fixed this by adding another base case to the outer if statement: if the target is zero and the size variable is equal to the actual size of the array – using the *sizeof* method – then return false.

Occurrences () ran into issues, whenever the digit being checked for was equal to 0, as shown in the following test cases.

```
assert(occurrences(123456789, 0) == 0);
assert(occurrences(0, 0) == 1);
```

The reason for this was because the base case of the method was when it finally got down to zero, having divided itself by 10 in each recursive call to check each digit. It was registering that final zero (the base case) as an instance of zero being in the original number, when it shouldn't be (as in the first case above). I modified my method so that whenever the input parameter *number* reached 0, the function would return zero. This then raised problems with the second case shown above, since even if both the original *number* and the given digit were zero, the function would return 0. However, I ignored this problem since the project instructions said that 0 was an acceptable result for this case.