Computer Science 331 – Assignment 2

Methodology:

My calcProduct method is comprised of one while loop that traverses the first number until the product of the two numbers is calculated and returned. The while loop is split into two sub-while loops that first calculates the partial product resulting from multiplying the character at I in number 1 with all the characters in number 2, and secondly updates the product by adding the new partial product to it. I chose to do it this way because alternatively I could have first calculated all the partial products in one loop than add all of them together in another, but that did not seem efficient since I would need an extra loop on top of the three here. I decided to traverse the first number in one cycle and do both processes simultaneously.

Proof for calcProduct method:

**While statement:**

The loop invariant for the parent while loop is that the product is always accurate for all i

Base Property: Before the first iteration of the loop the product is empty, this is accurate since no numbers have been calculated, added or multiplied

Inductive Property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution. At the end of the execution the product depends on the inner while loops, so in order to prove that the product is correct at the end of each execute we must prove that the inner while loops are always correct.

**Inner while loop 1:**

The loop invariant for the first sub loop is that the intermediate product calculation is always correct for all j.

Base Property: Before the first iteration of the loop the intermediate product is empty, this is accurate since no numbers have been multiplied together.

Inductive Property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution. At the end of the execution the execution:

Case 1: tempProduct < 10, since the multiplication of the two characters and addition of the carry is less than 10 and both characters are the same distance from the end of their corresponding strings, when added together with the carry value from the previous operation, they produce an accurate character representation of the multiplication of these two characters, and this can be added to the inter string producing an updated and correct inter string.

Case 2: tempProduct >= 10, since the multiplication of the two characters and the addition of the carry is greater than or equal to 10 we set the new carry value and modify the tempProduct so that it accurately represents the multiplication of these two characters, thus adding this to the inter string produces an accurate inter string

tempProduct cannot be larger than 100 since the largest multiplication of two numbers is 9 x 9 = 81, so further cases are unnecessary.

Since inter is always accurate for each case, the loop invariant holds at the end of each execution, thus this while loop is partially correct. To prove termination, the loop continues until j < 0. Since initially j = pNumber2.length() – 1 > 0 and j is decremented with each iteration, j will eventually lead to 0. Thus the while loop terminates so it is correct

**Inner while loop 2:**

Base Property: Before the first iteration of the loop the intSum is empty, this is accurate since no numbers have been added together yet

Inductive Property: Assume that the loop body is executed I > 0 times and that R is satisfied at the beginning of the I-th execution. At the end of that execution:

Case 1: n - 1 >= 0 & m - 1>= 0, since the numbers corresponding to n - 1and m – 1 are the same distance from the end of their corresponding strings, these numbers can be added together. A carry value is also added, its value depending on whether the previous addition was larger than ten. If the number is larger than 10 we subtract 10 to get the accurate char value, if the number is less than 10 we already have an accurate char value, thus for this case the char calculated is always correct.

Case 2: n - 1< 0 & m – 1 >= 0, since n is out of bounds for the string number 1, a 0 must be added to the number corresponding to the index j in the partial product intermediate number, a carry value is also added where its value depends on the previous addition. If the number is larger than 10 as a result of adding the carry value, we subtract 10 to get the accurate char value, if the number is less than 10 we already have an accurate char value, thus for this case the char calculated is always correct.

Case 3: n – 1 >= 0 & m – 1 < 0, since j - 1is out of bounds for the partial product value, a 0 must be added to the number corresponding to the index n - 1in the current product value, a carry value is also added where its value depends on the previous addition. If the number is larger than 10 as a result of adding the carry value, we subtract 10 to get the accurate char value, if the number is less than 10 we already have an accurate char value, thus for this case the char calculated is always correct.

Case 4: n – 1 < 0 & m - 1 < 0 & carry == 1, in this case the previous addition produced a number greater than 10 and we have reached the end of the bounds of the string so a 1 must be added to the start of the intSum to get the correct character, thus this case is always correct

Since the character is always accurate for each case, the loop invariant holds at the end of each execution, thus this while loop is partially correct. To prove termination, the loop continues until n >= -1 or m >= -1, for every iteration both values n and m are decremented, and since n = product.length() – 1 > -1 and m = inter.length() - 1 > -1; n and m will eventually reach -1. Thus the while loop terminates so it is correct

Since both inner while loops are correct, all partial products, and int sum is correct for each loop. And these values evaluate the product in each loop, product is correct at the end of each iteration in the outer while loop. Thus the outer loop is partially correct. To prove termination, the loop continues until I < 0. Since I is decremented and I is initially pNumber1.length() – 1 > 0 I will eventually reach and pass 0. Thus this while loop terminates so it is correct, thus the algorithm is correct.

Worst Case Cost:

**Worst case for inner loop 1:**

The test has 2 comparisons (> or =) and it is checked m + 1 times so there is max 2(m + 1) instructions

Worst case cost for execution of loop body (10 + 3 + 5 + 4 + 3 + 6 + 2) = 33 units

Upper bound on worst-case cost to execute the loop 2(m + 1) + 33m

**Worst case for inner loop 2:**

The test has 5 comparisons, 2: (> or =) and 1: or, and it is checked n + 1 times so this is max 5(n + 1) instructions.

Worst case cost for execution of loop body (9 + 7 + 2 + 8 + 3 + 4) = 33 units

Upper bound on worst-case cost to execute loop 5(n + 1) + 33n

**Worst case for outer loop:**

The test has 2 comparisons (> or =) and it is checked p + 1 times so there is max 2(p + 1) instructions

Worst case cost for execution of loop body 22+ 2(m + 1) + 33m + 5(n + 1) + 33n units

Upper bound on worst case cost to execute loop: 2(p + 1) + [22+ 2(m + 1) + 33m + 5(n + 1) + 33n]p

Total cost of calcProduct method: 2(p + 1) + [15+ 2(m + 1) + 33m + 5(n + 1) + 33n]p + 7

Run time:

I tested each case 5 times, ignoring outliers and calculated the average of each experiment

1. Two strings of length 10: ~1436 milliseconds
2. Two strings of length 100: ~2733 milliseconds
3. Two strings of length 1000: ~3500 milliseconds
4. First string 100 characters and second string 10 characters: ~1601 milliseconds
5. First string 10 characters and second string 100 characters: ~1881 milliseconds

Observation: The larger the string the larger the runtime, with a growth rate of 1000\*log(n) where n is the average numerical size of both strings, i.e. 10 or 1000.