Computer Science 331 – Assignment 1

Proof 1 for calcSum method:

While statement 1:

Loop Invariant: decSum.charAt(k) is always accurate for any k

Base Property: Before the first iteration of the loop the decSum 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: (i – 1) - pNumber1.indexOf(".") == (j -1) - pNumber2.indexOf("."), since they are the same distance from the decimal 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: (i – 1) - pNumber1.indexOf(".")) > (j – 1) - pNumber2.indexOf("."), since I – 1 is farther from the decimal than j – 1, a zero must be added to the character corresponding to I – 1, 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: (i – 1) - pNumber1.indexOf(".")) > (j – 1) - pNumber2.indexOf("."), since j – 1 is farther from the decimal than i – 1, a zero must be added to the character corresponding to j – 1, 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.

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 i > pNumber1.indexOf(".") or j > pNumber2.indexOf("."), for every iteration either I, or j or both values are decremented, and since I = pNumber1.length() – 1 > pNumber1.indexOf(".") and j = pNumber2.length() – 1 > pNumber2.indexOf("."); I and j will eventually reach pNumber1.indexOf(".") and pNumber2.indexOf(".") respectively. Thus the while loop terminates so it is correct

Maximum number of executions of the loop body: The worst case, which produces the longest runtime is achieved when we must cycle through the entire decimal portion of the integer with the longer decimal part. We will denote this by n

Worst case cost to evaluate test: 5 units: 2 comparisons and 1 Boolean operation and 2 calls of indexOf

Worst case cost for execution of loop body: 25 units, for when the first if statement is executed

Upper bound on worst-cast cost to execute the loop: 5(n + 1) + 25n

While statement 2:

Loop Invariant: intSum.charAt(k) is always accurate for any k

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: I - 1 >= 0 & j - 1>= 0, since the numbers corresponding to I - 1and j – 1 are the same distance from the decimal 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: i - 1< 0 & j – 1 >= 0, since I is out of bounds for the string number 1, a 0 must be added to the number corresponding to the index j in number 2, 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: i – 1 >= 0 & j – 1 < 0, since j - 1is out of bounds for the string number 2, a 0 must be added to the number corresponding to the index i - 1in number 1, 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: i – 1 == -1 & j - 1 == -1 & carryValue == 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 i >= -1 or j >= -1, for every iteration both values I and j are decremented, and since I = pNumber1.indexOf(".") – 1 > -1 and j = pNumber2.indexOf(".") - 1 > -1; I and j will eventually reach -1. Thus the while loop terminates so it is correct

Maximum number of executions of the loop body: The worst case, which produces the longest runtime is achieved when we must cycle through the entire integer portion of the integer with the longer integer part. We will denote this by p

Worst case cost to evaluate test: 3 units: 2 comparisons and 1 Boolean operation

Worst case cost for execution of loop body: 23 units, for when the first if statement is executed

Upper bound on worst-cast cost to execute the loop: 3(p + 1) + 23p

Proof 2 for calcDifference method:

While statement 1:

Loop Invariant: decDifference.charAt(k) is always accurate for any k

Base Property: Before the first iteration of the loop the decDifference is empty, this is accurate since no numbers have been subtracted 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: (i – 1) - pNumber1.indexOf(".") == (j -1) - pNumber2.indexOf("."), since they are the same distance from the decimal these numbers can be subtracted together. A carry value is also subtracted, its value depending on whether the previous addition was smaller than 0. If the number is smaller than 0 we add 10 to get the accurate char value, if the number is greater than 0 we already have an accurate char value, thus for this case the char calculated is always correct.

Case 2: (i – 1) - pNumber1.indexOf(".")) > (j – 1) - pNumber2.indexOf("."), since I – 1 is farther from the decimal than j – 1, a zero must be subtracted from the character corresponding to I – 1, a carry value is also subtracted where its value depends on the previous subtraction. If the number is smaller than 0 as a result of subtracting the carry value, we add 10 to get the accurate char value, if the number is greater than 0 we already have an accurate char value, thus for this case the char calculated is always correct.

Case 3: (i – 1) - pNumber1.indexOf(".")) > (j – 1) - pNumber2.indexOf("."), since j – 1 is farther from the decimal than i – 1, a zero must be subtracted by the character corresponding to j – 1, a carry value is also subtracted where its value depends on the previous subtraction. Since the number is guaranteed to be less than 0 we add 10 to get the accurate char value and set the carry value to 1.

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 i > pNumber1.indexOf(".") or j > pNumber2.indexOf("."), for every iteration either I, or j or both values are decremented, and since I = pNumber1.length() – 1 > pNumber1.indexOf(".") and j = pNumber2.length() – 1 > pNumber2.indexOf("."); I and j will eventually reach pNumber1.indexOf(".") and pNumber2.indexOf(".") respectively. Thus the while loop terminates so it is correct

Maximum number of executions of the loop body: The worst case, which produces the longest runtime is achieved when we must cycle through the entire decimal portion of the integer with the longer decimal part. We will denote this by n

Worst case cost to evaluate test: 5 units: 2 comparisons and 1 Boolean operation and 2 calls of indexOf

Worst case cost for execution of loop body: 25 units, for when the first if statement is executed

Upper bound on worst-cast cost to execute the loop: 5(n + 1) + 25n

While statement 2:

Loop Invariant: intDifference.charAt(k) is always accurate for any k

Base Property: Before the first iteration of the loop the intDifference is empty, this is accurate since no numbers have been subtracted 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: i – 1 >= 0 & j – 1 >= 0, since the numbers corresponding to I – 1 and j – 1 are the same distance from the decimal these numbers can be subtracted together. A carry value is also subtracted, its value depending on whether the previous subtraction was larger smaller than 0. If the number smaller than 0 we add 10 to get the accurate char value, if the number is greater than 0 we already have an accurate char value, thus for this case the char calculated is always correct.

Case 2: i -1 < 0 & j – 1 >= 0, since I – 1 is out of bounds for the string number 1, a 0 must be subtracted from the number corresponding to the index j - 1 in number 2, a carry value is also subtracted where its value depends on the previous subtraction. If the number is smaller than 0 as a result of subtracting the carry value, we add 10 to get the accurate char value, if the number is greater than 0 we already have an accurate char value, thus for this case the char calculated is always correct.

There is no case 3 because we will be always subtracting smaller numbers from larger numbers

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 i >= -1 or j >= -1, for every iteration both values I and j are decremented, and since I = pNumber1.indexOf(".") – 1 > -1 and j = pNumber2.indexOf(".") - 1 > -1; I and j will eventually reach -1. Thus the while loop terminates so it is correct

Maximum number of executions of the loop body: The worst case, which produces the longest runtime is achieved when we must cycle through the entire integer portion of the integer with the longer integer part. We will denote this by p

Worst case cost to evaluate test: 3 units: 2 comparisons and 1 Boolean operation

Worst case cost for execution of loop body: 21 units, for when the first if statement is executed

Upper bound on worst-cast cost to execute the loop: 3(p + 1) + 21p

Proof 3 for calcGreater method:

For the first section declaring and initializing variables the correctness is trivial

For the next 2 section of for loops the correctness is trivial

The third if statement contains a while loop that needs to be proven correct in order to prove the if statement correct. The correctness of the declarations of variables at the beginning is trivial

While statement:

Loop Invariant: getGreater is always accurate for any i

Base Property: Before the first iteration of the loop getGreater returns null, this is correct because the first test for I has not been evaluated

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:

The first two if statements are trivial since they don’t determine the value of greater

Case 1: charNumb1 == '.' | charNumb2 == '.', since neither characters are numbers they don’t have a greater than or less than property, thus the getGreater is still null which is accurate

Case 2: charNumb1 > charNumb2, since that character charNumb1 corresponds to a character at pNumber1.charAt(I - 1) and it is greater than charNumb2 which either corresponds to a character at pNumber1.charAt(j -1) or 0 in the case where the index is out of bounds, thus setGreater is set to number 1 and the loop terminates. This is the correct result

Case 3: charNumb1 < charNumb2, since that character charNumb2 corresponds to a character at pNumber1.charAt(j - 1) and it is greater than charNumb2 which either corresponds to a character at pNumber1.charAt(i -1) or 0 in the case where the index is out of bounds, thus setGreater is set to number 2 and the loop terminates. This is the correct result

Since getGreater 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 either I >= end or !done, since I = 0 =< end and I is incremented with every iteration the loop will eventually reach end and terminate regardless of whether done is true or false thus the while loop terminates so it is correct.

Since the while loop is correct, all the components of the if statement are correct, so all if statements are correct.

Maximum number of executions of the loop body: The worst case, which produces the longest runtime is achieved when we must cycle through entire string to get to end because the strings are identical.

Worst case cost to evaluate test: 3 units: 1 comparisons, negation and Boolean operation

Worst case cost for execution of loop body: 17 units, for when the last if statement is executed

Upper bound on worst-cast cost to execute the loop: 3(end + 1) + 17end

Worst case for entire program:

O(N) = 2[5(n + 1) + 25n] + [3(p + 1) + 23p] + [3(p + 1) + 21p] + [3(end + 1) + 17end]

Methodology:

To compare the numbers and determine which one is greater I broke the method up into three if statements. If the length of one number’s integer portion is larger than the other’s we set the one with the larger integer portion as the larger number. This of course assumes there are no redundant zeros at the beginning of the string. If the integer portions are the same length then third if statement is executed where a while loop compare each character till if finds one that is larger than another and sets the string corresponding to that character as the larger number. If the loop goes out of bounds for one string, in the case for example where we compare 2.3 and 2.34, the character of the first number is set to 0 appropriately.

To add the numbers together I broke the method up into two while loops, one that adds together the decimal portion of the string character by character and another that adds together the integer portion of the string character by character and carrying a one if needed, at the end the two strings are combined to get the final correct string

To subtract the numbers I chose the same methodology. I broke the method up into two while loops, one that subtracts together the decimal portion of the string character by character and another that subtracts together the integer portion of the string character by character carrying a 1 if needed, at the end the two strings are combined to get the final correct string