CPSC 331 Fall 2020

Dr. Wayne Eberly

Assignment 1

Reid Erb - 10089638

Matthew Newton - 30094756

Charith Pathirathna - 10162790

1. Suppose that the function f(n, m) = (n + m) is a bound function (Lecture Notes #2 is referenced here). To prove that it is a bound function of a recursive algorithm, we should assume that the problem's precondition is satisfied when the algorithm is executed, as well as satisfy the following properties:

I. That it is an integer-value function.
II. Whenever the algorithm is applied recursively, the value of the function has been decreased by at least one.
III. If the value of the function is less than or equal to zero when the algorithm is applied then the algorithm does not call itself recursively during this execution.

Proof:

I. From the precondition, we know that n and m are integer inputs. This would mean that (n + m) should result in an integer. Therefore, this is certainly an integer-value function.

II. When we consider the recursive function at line 3, we can see that the value of n and m is reduced by at least 1 ((n - 1, m), (n, m - 1)). So, the value of our hypothetical bound function would be either (n - 1) + m or n + (m - 1), which are both equivalent to saying (n + m) - 1, which is less than (n + m). Therefore, the function has decreased by at least one.

III. From the precondition, we know that n and m are nonnegative integers, which means that n >= 0 and m >= 0. Thus, it is only possible that f(n, m) = (n + m) <= 0 when n = 0 or when m = 0. Thus, the test at line 1 passes, continues the execution on line 2 and the execution ends without having the algorithm to call itself recursively.

Therefore, the function f(n, m) = (n + m) is a bound function as it satisfies all the conditions of a bound function for a recursive algorithm.

2. Suppose that n and m are nonnegative input integers and the execution of the algorithm eventually ends with f(n, m) is returned as output. [Uses Lecture #2 as a reference]

Proof: This theorem will be proved by induction on n and m. The strong form of mathematical induction will be used and the cases that n = 0 and m = 0 will be considered in the basis.

Basis:
Suppose, first that n = 0: With the execution of the sRoute algorithm on inputs n and m, the test at line 1 passes, so the execution continues at line 2. This causes the execution to terminate with the value of $f_{(0, m)} = 1$ being returned as output, as required.

Suppose, next that m = 0: With the execution of the sRoute algorithm on inputs n and m, the test at line 1 passes, so the execution continues at line 2. This causes the execution to terminate with the value of $f_{(n, 0)} = 1$ being returned as output, as required.

Inductive Step: Let k and l be integers such that k >= 0 and l >= 0. We use the inductive step to prove the following inductive claim:-
Inductive Hypothesis: Suppose that n and m are nonnegative integers such that 0 <= n <= k and 0 <= m <= l. If the sRoute algorithm is executed with n and m given as inputs, then this execution of the algorithm eventually ends, with the $F_{(n,m)}$number returned as output.

Inductive Claim: If the sRoute algorithm is executed given n = k + 1 and m = l + 1 as input, then this execution of the algorithm eventually ends, with $F_{(k+1, l+1)} = F_{(n, m)}$ returned as output.

Suppose that the sRoute algorithm is executed with n = k + 1 and m = l + 1 given as inputs. Since k and l are integers such that k >= 0 and l >= 0, n and m are integers such that n >= 1 and m >= 1. Since both n and m are greater than or equal to 1, the test at line 1 fails and the algorithm continues with the execution on line 3.

Line 3 contains a recursive execution of the sRoute algorithm with input (n – 1, m). Since n = k + 1 >= 1, 0 <= n – 1 = k, and it follows by the inductive hypothesis that this recursive execution of the sRoute algorithm eventually ends, with $F_{(n-1, m)} = F_{(k)}$ returned as output.

Line 3 also contains a recursive execution of the sRoute algorithm with input (n, m – 1). Since m = l + 1 >= 1, 0 <= m – 1 = l, and it follows by the inductive hypothesis that this recursive execution of the sRoute algorithm eventually ends, with $F_{(n, m - 1)} = F_{(l)}$ returned as output.

By inspection of line 3, we can see that this execution of the sRoute algorithm eventually ends. Since k + 1 >= 1 and l + 1 >= 1, it follows the definition of $F_{(k+1, l+1)}$ that the value returned as output is

$$F_{(n-1, m)} + F_{(n, m-1)} = F_{(k)} + F_{(l)}$$

as required to establish the inductive claim. The result now follows by induction on n.

4. For the values of the program sRoute, we have to consider the cost of the algorithm in terms of the steps executed. This is the recurrence for the algorithm:

$$T_{Routes1}(n, m) = \begin{cases} 2 & \text{if } n = 0, \text{ or } m = 0, \text{ or both} \\ T_{Routes1}(n-1, m) + T_{Routes1}(n, m-1) + 2 & \text{if } n >= 1 \;\&\&\; m >= 1 \end{cases}$$

If either n or m equal zero, then the “if” statement at line 1 of the algorithm will be true, so the algorithm will return a value of 1, then terminate the recursion. The cost of executing these instructions will simply be 2.

$$T_{Routes1}(0, 0) = 2$$

$$T_{Routes1}(0, 1) = 2$$

$$T_{Routes1}(1, 0) = 2$$

Next we will take into account the case where n and m are both greater than or equal to 1. If both n and m are greater than zero, there will be one step where the program checks the if statement, T(n - 1, m) steps to find the value of sRoute(n - 1, m), T(n, m - 1) steps to find the value of sRoute(n, m - 1), and one step to add sRoute(n - 1, m) and sRoute(n, m-1) together. So, for example, at (1, 1):

$$\begin{aligned} T_{Routes1}(1, 1) &= T_{Routes1}(1 - 1, 1) + T_{Routes1}(1, 1 - 1) + 2 \\ &= T_{Routes1}(0, 1) + T_{Routes1}(1, 0) + 2 \\ &= 2 + 2 + 2 \\ &= 6 \end{aligned}$$

5. Now, we will attempt to prove that T(n, n) is greater than or equal to 2^n

From the recurrence in the solution to problem 4, we can see that T(0, 0) = 2, which is greater than $2^0 = 1$. We can also use the recurrence to find the value of T(1,1):

$$T(1,1) = T(0,1) + T(1,0) + 2$$
$$T(1,1) = 2 + 2 + 2$$
$$T(1,1) = 6$$

So, T(1,1) = 6, which is greater than $2^1 = 2$. 6 is also divisible by 2, which will be important in proving our claim.

T(0,0) and T(1,1) are special cases. We see an interesting pattern develop when we consider T(n,n) where n > 1. Let's look at T(2,2), for example:

$$T(2,2) = T(1,2) + T(2,1) + 2$$
$$T(2,2) = [T(0,2) + T(1,1) + 2] + [T(1,1) + T(2,0) + 2] + 2$$
$$T(2,2) = 2 \times T(1,1) + T(0,2) + T(2,0) + 6$$

Here, we see that the value 2 x T(1,1) appears in our calculation of T(2,2). Since we know T(1,1) = 6, and 6 = 2 x 3, the value 2 x T(1,1) could be stated in a different form as $(2^2) \times 3$, or 3 multiplied by 2^2.

In this calculation, it would be very easy to find the exact value of T(2,2), but what is important to proving our claim is that the term $(2^2) \times 3$ is included in the calculation. We know that the values of any given T(n, m) will always be non-negative, so

regardless of the actual value of T(0,2) and T(2,0) here, we know that T(2,2) will be greater than (2^2) x 3, and therefore greater than (2^2), and therefore greater than $2^2 = 4$.

This result can be generalized, and T(n,n) can always be put in a form such that the term 2*T(n-1,n-1) will be included and added to other non-negative integer values. This can be shown simply by manipulating the recurrence. For values of n greater than 1:

$$T(n,n) = T(n-1, n) + T(n, n-1) + 2$$
$$T(n,n) = [T(n-2, n) + T(n-1, n-1) + 2] + [T(n-1, n-1) + T(n, n-2) + 2] + 2$$
$$T(n,n) = 2 \times T(n-1, n-1) + T(n-2, n) + T(n, n-2) + 6$$

It would follow from this that T(n-1, n-1) would also contain the term 2 x T(n-2, n-2) and so on.

Let's consider a simple recurrence where V(n) = 2 x V(n-1) and V(0) = 1. This is similar to the more complicated recurrence we're considering, so if we can prove that this recurrence is equivalent to 2^n , then our claim will be proven. We will use standard induction to prove this.

So, our inductive hypothesis is that $V(n) = 2^n$

The basic case for this recurrence would be V(1).

$$V(1) = 2 \times 1 = 2 = 2^1$$

So, the claim is true for the basic case. Now, we will induct on n to show that the claim would be true at (n + 1)

$$V(n + 1) = 2 \times V(n) = 2 \times (2^n) = 2^{(n+1)}$$

So, we see that our claim is true for this simplified recurrence. T(n,n) is more complicated and includes other elements, but we know there will be some part of it that will be formed by being recursively multiplied by 2. Moreover, the value of T(n, n) will always be greater than that of V(n). Therefore, we know that whatever T(n,n)'s value is, it is definitely greater than 2^n.

6. The loop invariant of the inner loop is provided below as well a proof for the invariant:

a) n and m are integer inputs such that n >= 1 and m >= 1
b) R is an ((n + 1) x (m + 1)) array of integers.
c) i is an integer with value greater than or equal to 0 and less than or equal to n
d) j is an integer with value greater than or equal to 0 and less than or equal to (m + 1)
e) R[k][L] == Routes(k, L) for all integers k and L such that 0 <= k <= i and 0<= L < j

a) It is a precondition of the algorithm that n and m are non-negative integers, and if either n or m are equal to zero, then the if statement at line 1 would be true and the algorithm would return a value before it even gets to the loop. To get to the loop at all, both integers need to be positive, not merely non-negative. This proves property (a) of the loop invariant.

b) The array R is created at line 3 with ((n + 1) x (m + 1)) elements. Once an array's dimensions are defined they can't be changed, so at any given iteration and any given point of the loop, the array will have this size, proving (b).

c) The integer i is initialized with the value of 0 and is incremented by 1 after each iteration of the interior loop (but isn't changed in any other way), so 0 <= i. At the end of the final iteration of the exterior loop, i will have the value (n + 1), but it's only given this value after the final iteration of the interior loop, at which point it has the value n, proving (c).

d) The integer j is also initialized with the value of 0 and is incremented by 1 before the end of each iteration of the interior loop, so 0 <= j. The "while" condition of the interior loop checks if j <= m, so at the beginning of the final iteration of an interior loop will have m, and then in the loop body it will be increased to (m + 1). Then, when the loop checks its condition, it will be false and the interior loop will terminate and either the exterior loop will start over and j will be reset to 0, or the exterior loop will also terminate. So, 0 <= j <= (m + 1) and (d) is true.

e) On any given iteration of the loop, R[k][L] will only be defined if it was defined on some previous iteration. Whatever value i has at the beginning of an iteration of the interior loop, it will retain that value until the interior loop terminates. So, if k < i, R[k][L] is defined with the value of Routes(k, L). When k == i, R[k][L] will definitely be defined for all L < j, and may or may not be defined for L == j, so, for the loop invariant we can only definitively say R[i][L] == Routes(i, L) if L < j. Arrays can't have negative indices, so k >= 0 and L >= 0. All this proves that (e) is true.

7.

a) n and m are both integer inputs such that n >= 1 and m >= 1
b) R is an ((n + 1) x (m + 1)) array of integers
c) i is an integer variable such that 0 <= i <= (n + 1)
d) R[k][L] = Routes(k, L) for all integers k and L such that 0 <= k <= (i - 1) and 0 <= L <= m

a) According to the precondition of the algorithm, n and m will both be non-negative integers, so they will not be less than zero. If either variable is equal to zero, then (n == 0 or m == 0) will be true, which satisfies the if statement at line 1, and the program won't execute the loop. For the algorithm to execute the loop, both n and m will have to be non-zero, non-negative integers, or, to put it in other words, positive integers. The smallest possible value for a positive integer is 1, so n and m must both be at least equal to 1 for the loop to execute. There is no upper bound to the value of n and m (though it may take a long time to calculate Routes(n,m) with large values of n or m). The values for n and m are not changed at any point in the loop. So n >= 1 and m >= 1, and (a) is a loop invariant.

b) The array R is created at line 3 of the algorithm with size ((n + 1) x (m + 1)) before the loop begins. An array's size can't be changed once it's initialized, so this will be true at any point in the loop and (b) is true.

c) The integer i is initialized with the value 0 at line 4, before the loop begins. Its value is increased by one on every iteration of the exterior loop, but isn't changed in any other way. The condition for the while loop at line 5 checks if i <= n and executes the loop body if this is true, so, we can deduce that on the

final iteration of the loop, i == n at the beginning, and then it will be incremented to (n + 1) at line 12. There will be one more condition check for the while loop, but since i > n at this point, the condition will not be true, and the loop will terminate and thus i will not be incremented either. So, whatever value i has, it will be in the range 0 <= i <= (n + 1), and (c) is true.

d) Any given row in R has (m + 1) elements (as stated in assertion (c)), so the last element in a row i will be R[i][m]. Whatever the current value of i is, it represents some row in the array. If R[k][L] is defined with the value Routes(k, L) for all integers k and L such that 0 <= k <= (i - 1) and 0 <= L <=m, a simpler way to state that assertion would be to say that whichever row is currently being given value in the loop, every row before that (and every element therein) already has a definite value, namely, the value of Routes(k, L), where k is the row index and L is the column index. If we look at the loop we can see that the interior loop will define values for the array R a row at a time. So, at any given i, the previous row (i - 1) will be defined, (provided (i - 1) >= 0) and (d) is true.

8. The following is the proof for the partial correctness of sRoutes2 (n, m).

Proof: In order to prove partial correctness, one or the other of the following properties should be satisfied:

a) that the execution of the algorithm ends, where the postcondition being satisfied with no access or modification to the inputs or global data. Or

b) the execution of the algorithm never halts, at all.

Consider the execution of the sRoutes2 algorithm where the precondition is satisfied. Thus, the inputs n and m are nonnegative integers where n >= 0 and m >= 0. By inspection, the algorithm does not access or modify any undocumented inputs or global data.

So, if the precondition of the algorithm is satisfied, and n and m are both non-negative integers, we can show from the loop invariants that the algorithm is partially correct. If an algorithm eventually terminates and satisfies its

postcondition, then that would be sufficient proof that it is partially correct. The postcondition of the algorithm here is that it will return the value of Routes(n, m) as output. We know from assertion (d) of the loop invariant for the outer loop that R[k][L] will equal Routes(k, L) for all values of the integer k and the integer L such that 0 <= k <= (i - 1) and 0 <= L <= m. We also know from assertion (c) of the same loop invariant that i will equal (n + 1) at the end of the last iteration of the loop. Putting these together, we can say that once the loop ends, R[k][L] will be defined with the value for every k such that 0 <= k <= n and every L such that 0 <= L <= m. Since i == (n + 1) then:

$$i - 1 == (n + 1) - 1 == n$$

We know that n and m are non-negative integers and their values won't be changed anywhere in the algorithm after they're passed in. The integers i and j are both initialized with value zero, and they are increased on every iteration of their respective loops, but aren't changed in any other way, so we know that at some point i will become greater than n and j will become greater than m, so it is certain that the loops will terminate.

So, R[n][m] will equal Routes(n, m), and the algorithm will terminate. The algorithm ends by returning R[n][m], so the postcondition is satisfied. The algorithm is partially correct.

9. The bound function for the inner loop is F = m - j + 1. Since j increases by one on each iteration, the function is decreased by one on each iteration. At the end of the final iteration of an interior loop, j = (m + 1) so:

$$F = m - (m + 1) - 1$$
$$F = (m - m) + (1 - 1)$$
$$F = 0$$

So, the interior while loop terminates when the bound function equals zero. Bound functions for while loops must decrease by at least one on each iteration, and equal zero when the loop terminates. F satisfies both of those conditions, so F = m - j + 1 is a bound function for the inner loop.

10. Again, a bound function for a while loop must decrease by at least one on each iteration and equal zero when the loop terminates. We know that the value for i

increases on each iteration of the exterior loop, so it would seem that G = n - i + 1 could be a bound function for it. Similarly to the interior loop, the value for i at the end of the final iteration will be i = (n + 1), so:

G = n - (n + 1) + 1
G = n - n - 1 + 1
G = 0

After the final iteration of the loop, the program will check if i satisfies the condition of the while loop one last time, then see that it doesn't because (n + 1) is not less than or equal to n, so the loop will terminate. That means that the value of G is equal to zero when the loop terminates.

G = n - i + 1 both decreases by at least one on each iteration of the exterior loop, and the loop terminates when G is less than or equal to zero, therefore n - i + 1 is a bound function for the exterior loop.

11. In the solution to Problem 8 we showed that the algorithm has partial correctness, and that as long as the algorithm terminates, it would return the correct result. In solutions 9 and 10, we showed the bound functions of the interior and exterior loops, meaning that each loop will terminate when passed any non-negative integers n and m. Since the algorithm is partially correct and will definitely terminate if the preconditions are satisfied, we know it is correct.

Now, we can make a function of n and m to represent the upper bound of the algorithm. If either n or m were zero, the function would immediately return a value. We are interested in longer running times, so we can assume that both n and m will be greater than or equal to one.

No matter how many times the loops run through, lines 1, 3, 4, and 13 will only execute once, so we can assume they will have a fixed cost of 4.

The program will check the condition of the while loop at line 5 a total of (n + 2) times.

The code in the body of the exterior loop (lines 6 - 12) will execute (n + 1) times, but calculating the cost of this is somewhat complicated because the loop includes

another loop. We know that lines 6 and 12 will both have a cost of one on each iteration, but to calculate the cost of the exterior loop we first have to calculate the cost of the inner loop.

The condition check for the interior loop at line 7 will be executed (m + 2) times, and its body will be executed (m + 1) times. The loop body will have the same cost whether or not the if statement at line 8 is true. This is because both the code executed if it's true, and that executed if it's false both have a cost of one. So, the loop body will have a cost of one to check the if statement, a cost of one to assign value to some element of the array R, and one to increment the value of j at line 11. So, the cost of the inner loop is:

$= (m + 2) + 3(m + 1)$
$= m + 2 + 3m + 3$
$= 4m + 5$

Now that we know the cost of the inner loop, we can find the cost of the exterior loop body. It will execute the inner loop and two other lines of code (n + 1) times. So the cost of the loop body will be:

$= (n + 1) \times ((4m + 5) + 2)$
$= (n + 1) \times (4m + 7)$
$= 4nm + 7n + 4m + 7$

Putting together the cost of the lines outside the loops, the cost of the condition checks for the exterior loop, and the cost of the exterior loop body, we get:

$= 4 + (n + 2) + (n + 1) \times ((m + 2) + 3(m + 1) + 2)$
$= 4 + (n + 2) + (4nm + 7n + 4m + 7)$
$= 4nm + 8n + 4m + 13$

The upper bound of the cost of running the algorithm can be expressed as:

$T(n, m) = 4nm + 8n + 4m + 13.$