Fabien Bessez

COMP 211: Computer Science I

Homework 3A: Program Costs and Loop Invariants

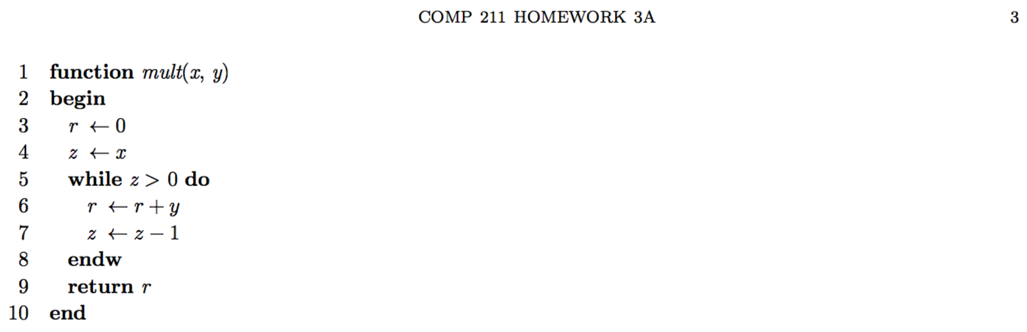
Problem 1. Prove that the merge_sort implementation posted in the HW 2B solutions runs in time O(n lg n). Count the number of key comparisons. Assume that list.append and list.extend perform no comparisons.

In the merge_sort implementation posted in the HW 2B solutions, we find that the algorithm runs in time O(n lg n). We find in the implementation that there are three main loops to talk about and a few start-up costs in key assignments. To begin, we could start at the inner-most loop of the implementation which we find in the merge function *while I < m and j < n*. Each time that this loop is iterated through, i or j is enumerated by one. Since we know that the difference between i and m or j and n is the value represented by *size*, we can conclude that this loop runs (in its worst case scenario) *2 * size -1* times. We get this value because if we iterate through the list adding 1 to i each time for m – 1 times and we do the same for j for n – 1 times, we get that we almost iterated through the loop size times *twice!* Then, we know that the next iteration through the loop will have to result in adding 1 to either i or j so we subtract one. Resulting in: *2 * size – 1*. The next portion of the implementation is the inner while loop of the merge_sort function and its condition is *while i < len(xs)*. Then, it goes on to assign *i = i + 2 * size*. Because this loop will occur until *i ≥ len(xs),* we can see that this loop will run *len(xs) / 2 * size* times. At this point, we should notice that the loop that runs *len(xs) / 2 * size* times will call the first loop I mentioned above each time. Therefore, we should multiply the two times together to get the current value of our analysis up until the two inner-most loops of the implementation. We get *2 * size -1 * (len(xs) / 2 * size).* The 1 is negligible and then the *2*size / 2 * size* can be divided through resulting in *len(xs),* which happens to equal n in this case. Since there are constants involved that get left out for simplicity sake, we write this as O(n). Next, we have to measure the run time the outer-most loop. It states: *while size < len(xs).* Each time through the list, takes *O(n)* time and it will run *log(n)* times because each iteration through the loop, the size variable is multiplied by 2. When we multiply these two values together, we get *O(n lg n)*. Then we have to notice that there is a key comparison at the beginning of merge_sort that is not in a loop, but since it only occurs once at start-up, it can be grouped into the big-Oh notation.

Problem 2. Compute the cost of the radix_sort implementation posted in the HW 2B solutions. Describe the cost in terms of k and n, where k is the number of digits in each numeral and n is the number of numerals (in the context of that code, k is the variable k and n is len(xs)).

In every cost analysis of an implementation, it is important to take note of the worst-case scenario. In this implementation, it seems as if the worst-case scenario is when all of the lists of lists of binary are equal and therefore, each bit has to be compared. In other words, if we had two numerals that had values [1,0,1] and [0,1,1], it would take shorter time than [0,0,1] and [0,0,1] because radix_sort would only consider the most significant bit once and come to a conclusion. Now, we must take a look at the code. The inner-most while loop will iterate through *len(xs)* or *n* times. We know this because one of the conditions is incremented by 1 after each iteration and it will continue to do so until it reaches *len(xs)* or until it completes the particular prefix. The middle loop appears to run *len(xs)* times as well but it is important to notice that we must divide that number by jj-j. The final loop will run *k* times where *k* represents the number of digits in each numeral. Basically, if the elements in the array are *k*-bits long, then it will take *k* passes over each numeral to perform the sort. First, the numerals are sorted into two bins by most significant bit. Then each proceeding bit will be measured within the bin that it was last seen in. In the worst-case scenario, each of the numerals will be sorted into k-bins, n-times and therefore we get O*(nk)*.

Problem 3. Prove that the program in Figure 1 satisfies the following specification: mult(x, y) = x · y Pre–conditions: x ≥ 0 Do so by establishing and then using the following loop invariant: 0 ≤ z ≤ x and r = (x – z) · y


COMP 211 HOMEWORK 3A 3

```
1  function mult(x, y)
2  begin
3    r ← 0
4    z ← x
5    while z > 0 do
6      r ← r + y
7      z ← z − 1
8    endw
9    return r
10 end
```

FIGURE 1. An algorithm for multiplication.

Initialization:

Just before the loop, in lines 3 and 4, the value 0 is assigned to variable r and the given value x is assigned to variable z. Therefore, we know that r = 0 and z = x. <u>Need to Show:</u> 0 ≤ z ≤ x with the value for z substituted in. To show that 0 ≤ z ≤ x holds true in initialization, we substitute x for z. This looks like this: 0 ≤ 0 ≤ x. Since we know from the precondition (x ≥ 0) that x is greater than 0, the inequality is true. <u>Need to Show:</u> r = (x – z) * y. To do this, we substitute values for r and z into the equation. This looks like this: 0 = (x – x) * y. Now, we evaluate the equation. (x-x) = 0 so we get 0 = 0 * y. And by fact of multiplication, multiplying any value by 0 results in 0. The equation is held true as well and therefore initialization is complete.

Preservation:

Case1: Test case in line 5 fails.

In order for this to happen, it must be true that z ≤ 0. In the assignment stage, we know that z = x ≥ 0 so if we combine those two statements, we have that 0 ≥ z ≤ 0. This means that it must be true that z = 0. Now we try to verify first part of loop invariant: 0 ≤ z ≤ x. We know that it is true that 0 ≤ 0 ≤ 0. Then, we have to verify the second part of the loop invariant: r = (x – z) * y. Since variables r and z have not been modified, we know by the initialization process that this holds true. Case1 has been completed.

Case2: Test case in line 5 passes.

First, we must understand that r_{end} is the value of r after the loop is executed and that r_{st} is the value of r before the loop is executed. Same thing goes for z_{st} and z_{end}. Now, we follow the pseudo-code. It assigns $r_{end} = r_{st} + y$. It also assigns $z_{end} = z_{st} – 1$. Now that completes the loop. We must now verify both parts of the loop invariant. Need to Show: $0 \leq z_{end} \leq x$ and $r_{end} = (x – z_{end}) * y$. To figure out that $0 \leq z_{end} \leq x$ we must plug in the values for these variables. We know that $z_{end} = z_{st} – 1$ so we plug in that for z_{end}. This gives us: $0 \leq z_{st} - 1 \leq x$. Since we know that $z_{st} = x$, and that we are dealing with integers here, we know that z_{st} (in its worst case scenario when it equals x) will be less than x when it has one subtracted from it. This verifies first part of the loop invariant. Now to show: $r_{end} = (x – z_{end}) * y$. To do this, we must also plug in variable values. We know that $r_{end} = r_{st} + y$. We know that $z_{end} = z_{st} – 1$. So when we plug in these values we get: $r_{st} + y = (x – z_{st} – 1) * y$. We also know that z_{st} is equal to x from line 4 of the code. If we distribute the y, we get : $r_{st} + y = (xy – xy – y)$. Then we solve for r_{st} to get $r_{st} = -2y$. Plug that back into equation we had just before solving for r_{st} and we get: $-2y + y = (xy – xy – y)$. Simplify that and we get $–y = -y$. This verifies the second part of the loop invariant.

Usefulness:

Need to show: This loop invariant is useful.

To do this, we must show that when we return r in line 9 of the code, that r = x * y So, at this point, when the function returns r, we can assume that it has gone through the loop until z = 0. To verify that r = x * y with the loop invariant r = (x-z) * y, we must plug in our current value for z and see if r = x * y is equivalent to r = (x – z) * y. This is obvious since z = 0 , we get that r = (x – 0) * y. This is equivalent to r = x * y which proves this loop invariant useful!

Problem 4. Prove that the following is a loop invariant for the inner loop of selection sort (Figure 2, Lines 7–12): xs[min idx] = min(xs[i : j]). A fact that you may find useful is that min(ys[a : b+1]) = min(ys[a : b], ys[b]).

```
1  function sel_sort(xs)
2  begin
3    i ← 0
4    while i < len(xs)−1 do
5      min_idx ← i
6      j ← i + 1
7      while j < len(xs) do
8        if xs[j] < xs[min_idx] then
9          min_idx = j
10       endif
11       j ← j + 1
12     endw
13
14     swap(xs, i, min_idx)
15     i ← i + 1
16   endw
17 end
```

Figure 2. A selection sort implementation. *swap*(*ys*, *i*, *j*) swaps *ys*[*i*] and *ys*[*j*].

Initialization:

Need to show: xs[min idx] = min(xs[i : j]) holds true at the beginning of line 7. To do this, we must look at the facts that we know before we get to line 7. We know that i = 0, that i < len(xs) – 1, that min_idx_{st} = i, and that j_{st} = i + 1. So, if we plug in our known values for these variables, we find that we need to show xs[i] = min(xs[i: i+1]). By definition of the slicing notation, we know that xs[i:i+1] is simply xs[i]. When we call the min() function on just xs[i] we are left with no other option but to return xs[i]. This is what we needed to show, that xs[i] = xs[i]. This proves initialization!

Preservation:

Let me first introduce notation for j_{st} and j_{end}, min_idx_{st} and min_idx_{end}.

Case1: Test on line 7 fails.

This would mean that $j_{st} \geq$ len(xs). If this occurs, then we know by the initialization that the loop invariant is preserved since none of the variables are changed until we are passed line 12.

Case2: Test on line 7 passes.

This would mean that j_{st} < len(xs). Need to show: xs[min_idx_{end}] = min(xs[i: j_{end}]) In Case2a, the if-statement on line 8 is executed. This means that xs[j_{st}] < xs[min_idx_{st}]. Then we follow the code on line 9 which states that $\text{min\_idx}_{end} = j_{st}$ and on line 10 which states that $j_{end} = j_{st} + 1$. When we plug in these values for what we needed to show a few sentences ago, we get that we need to show that: xs[j_{st}] = min(xs[i: j_{st} + 1]). From our fact that xs[j_{st}] < xs[min_idx_{st}], we conclude that the min value of xs[i: j_{st} + 1] is in fact xs[j_{st}]. Which is exactly what we were trying to show.

In Case2b, the if-statement on line 8 is not executed and we therefore jump to line when $j_{end} = j_{st} + 1$. Notice that $\text{min\_idx}_{st} = \text{min\_idx}_{end}$ because it was never changed on line 9. We also get from the fact that line 8 is not executed that xs[j_{st}] $\geq$ xs[min_idx_{st}]. Need to show: xs[min_idx_{end}] = min(xs[i: j_{end}]). When we plug in values for the variables in our need-to-show sentence, we get **xs[min_idx_{st}] = min(xs[i: j_{end}])** as our new objective to show. However, we also know that min_idx_{st} = i and that $j_{end} = j_{st} + 1$ so we plug those in as well. Need to show: xs[i] = min(xs[i: j_{st} + 1]). A FACT given to us is that min(xs[a:b+1]) = min(xs[a:b], xs[b]). In this scenario, that FACT can be utilized in the following way: min(xs[i: j_{end}]) = min(xs[i:j_{st}], xs[j_{st}]). Since we know that xs[j_{st}] $\geq$ xs[min_idx_{st}], we can conclude that the min(xs[i:j_{st}], xs[j_{st}]) = xs[min_idx_{st}]. And since min_idx_{st} = min_idx_{end} we find that xs[min_idx_{end}] = xs[min_idx_{end}] which is what we needed to show in the bolded section right above.

Preservation is now complete!

Usefulness:

In order for this loop invariant to be considered useful, we must know that it holding true is vital to the outcome of the function call. The values min_idx and j in xs[min_idx] = min(xs[i : j]) are to be swapped on line 14 of the code so therefore it must be true that xs[min_idx] is the smallest value in the slicing of xs[i:j]. Otherwise, the final result of xs will not be in sorted order!