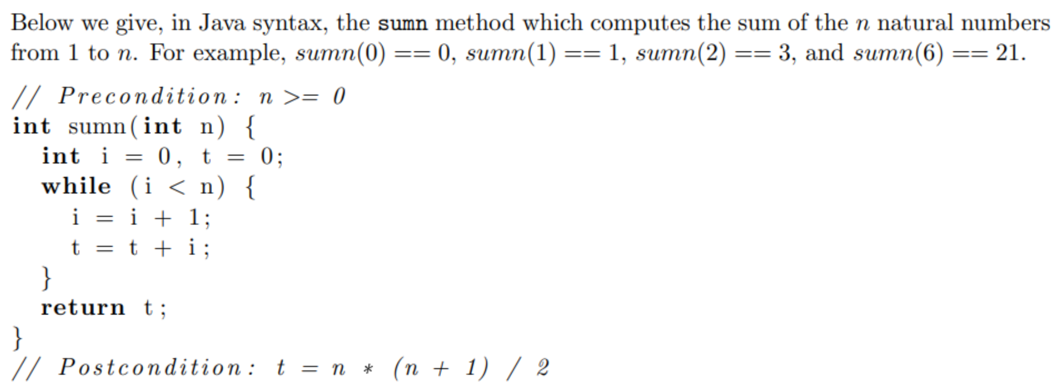
Joshua Collins

Principle of Software

2/8/2024

Homework 2

1. Sum of natural number:



Below we give, in Java syntax, the `sumn` method which computes the sum of the n natural numbers from 1 to n. For example, $sumn(0) == 0$, $sumn(1) == 1$, $sumn(2) == 3$, and $sumn(6) == 21$.

```
// Precondition: n >= 0
int sumn(int n) {
  int i = 0, t = 0;
  while (i < n) {
    i = i + 1;
    t = t + i;
  }
  return t;
}
// Postcondition: t = n * (n + 1) / 2
```

a. Find a suitable loop invariant.

Since we use a while loop to increment i until it is equal to n, we need an invariant to specify this. Not only this, but with each loop, t would have the output for the function at that value of i. Based off of these criteria, a suitable loop invariant would be:

Invariant (0 <= i <= n && t == i * (i + 1) / 2)

b. Show that the invariant holds before the loop (base case).

Base case: n = 1

i = 0, t = 0

0 <= 0 <= 1 && 1 == 1 * (2) / 2 && i < n ==> i = 1, t = 1

0 <= 1 <= 1 && 1 == 1 * (2) / 2 && i < n (false, 1 < 1) ==> output: 1

c. Show by induction that if the invariant holds after k-th iteration, and execution takes a k+1-st iteration, the invariant still holds (inductive step).

Assume t = n * (n + 1) / 2

T would be equal to the series S = 1 + 2 + 3 + 4 + … + k

Prove k+1th iteration:

t + 1 = 1 + 2 + 3 + 4 + … + k + (k+1) = $\frac{k(k+1)}{2} \times (k+1) = \frac{k(k+1)+2(k+1)}{2}$

$$\frac{k(k+1)+2(k+1)}{2} = \frac{k^2+3k+2}{2} = \frac{(k+1)(k+2)}{2}$$

Thus, the postcondition will be true for all iterations after the base case.

d. Show that the loop exit condition and the loop invariant imply the postcondition t = n∗(n+1)/2.

At the end of the loop: t = i * (i + 1) / 2 and i == n
Thus, t = n * (n + 1) / 2

e. Find a suitable decrementing function. Show that the function is non-negative before loop starts, that it decreases at each iteration and that when it reaches 0 the loop is exited.

Decrement = (n – i)
This holds true because I increases by 1 each iteration, so the value (n – i) will decrease with each iteration.

2. Loopy square root

```
method loopysqrt(n:int) returns (root:int)
    requires n >= 0
    ensures root * root == n
    {
        root := 0;
        var a := n;
        while (a > 0)
            //decreases //FILL IN DECREMENTING FUNCTION HERE
            //invariant //FILL IN INVARIANT HERE
            {
                root := root + 1;
                a := a - (2 * root - 1);
            }
    }
```

a. Test this code by creating the Main() method and calling loopysqrt() with arguments like 4, 25, 49, etc. to convince yourself that this algorithm appears to be working correctly. In your answer, describe your tests and the corresponding output. (1 pt.)

In the Main() method, I called for the loopysqrt() method four times and tried to assert those calls with specific outputs. All tests output the Assertion Error violation. The tests should've outputted the following:

```
var result := loopysqrt(4);
assert result == 2;

result := loopysqrt(9);
assert result == 3;

result := loopysqrt(25);
assert result == 5;

result := loopysqrt(49);
assert result == 7;
```

Thus, the error is caused by a post condition that is not implied by the loop invariant and precondition. The while loop also needs an invariant to solve the error.

b. Yet, the code given above fails to verify with Dafny. One of the reasons for this is that it actually does have a bug. More specifically, this code may produce the result which does not comply with the specification. Write a test (or tests) that reveals the bug. In your answer, describe your test(s), the corresponding outputs, and the bug that you found. Also, indicate which part of the specification is violated. (1 pt.)

I wrote the assertion statement "result * result == 27" for when we call the loopysqrt() method with an input of 27. The assertion fails, which is caused by the output being an integer, but the square root of 27 would be a decimal. The assertion fails due to the nature of the methods framework and the poor postcondition that would cause this output to fail is not integer squared equals 27.

c. Now find and fix this bug. Note that there might be several different ways of fixing the bug. Use the method that you think would be the best. You are not allowed to change the header of loopysqrt() or add, remove, or change any specifications or annotations. Do not worry about Dafny verifying the code for now, just fix the bug and convince yourself that loopysqrt() is now correct. You are also required to keep the overall algorithm the same as in the original version of the code. In your answer, describe how you fixed the bug and show the output of the same tests you ran before after fixing the bug. (2 pts.)

Before:

```
.code.tio.dfy(7,2): Error BP5003: A postcondition might not hold on this return path.
.code.tio.dfy(3,21): Related location: This is the postcondition that might not hold.
```

After:

This issue stems from the post condition not being implied from the loop invariant and precondition. To fix this I added a decreasing function that decreases a. a is the only part of this code decreasing so it's the obvious choice. Then I added a strong invariant for the while loop. In the while loop, the root will always be zero or greater and our input n is always equal to or greater than a. This allows us to find the closest integer to the root of n. If n is a square root, then we have our square root. If n is not a perfect square, like 27 for example, then we cannot output the actual root since our method outputs integers and $\sqrt{27} \approx 5.1962$. To fix this, I added an if statement that makes root equal to -1 if n is not a perfect square. This then allows us to imply the current post condition sometimes, but if we change It to (root*root == n || root == -1), then we can imply the entire postcondition. Now we have a valid output:

```
Var test := loopysqrt(27);
Print test;
```

Output: -1 (means the root is not valid, and n is not a perfect square)

d. Update the specification of loopysqrt() to match the way you fixed the bug. If you changed the postcondition, make sure that it is the strongest possible post condition. In your answer, describe your changes and explain why they were necessary. (1 pt.)

The only things I've added or changed are the post condition, decreasing function, invariant, and added a if statement. The decrease function targets a since a is the only variable in the while loop to be decreasing. The invariant for the loop states "0 <= root && n >= a". This is the strongest precondition and the initialization of variables a and root. I added the if statement for the cases where $root^2$ does not equal n. This if statement sets root to -1 for these cases since n is not a perfect square and root cannot be displayed as an integer. This means that our post condition needs a slight addition to be truly implied by our code. The new post condition would then be: root*root == n || root == -1. This means that root is either a root of a perfect square or not found (displayed as -1).

e. Does your Dafny code verify now? Why or why not? If it doesn't verify, does it mean that your code still has bugs in it? (1 pt.)

My code verifies correctly.

f. If your Dafny code doesn't verify, uncomment invariant and/or decreases annotations and supply the actual invariant and/or decrementing function. Make sure your code now verifies. In your answer, describe how you guessed the invariant and/or decrementing function. Explain why your code was failing Dafny verification earlier but does verify now, despite the fact that you have not made any changes to your actual code (annotations are not part of the code). (1 pt.)

I figured out the decreasing function by looking for variables that decrease within the while loop. a is the only function to do this so the following function is what I used:

Decrease a

The loop invariant has to be implied by the precondition and must remain true for each iteration of the while loop. Since a decrease and is initially set to n, that must be part of the invariant. The other part comes from the initialization of root. Root starts out as 0 and increases within the loop, so 0 <= root.

Invariant 0 <= root && n >= a

g. Finally, remove the precondition from your Dafny code and make necessary changes to the remaining annotations and/or code ensure that code still verifies. You are not allowed to change the header of loopysqrt(). You are also required to keep the overall algorithm the same as in the previous versions of the code. As before, make sure that your postcondition is the strongest. Did removing the precondition make your code more difficult? What effect did removing the precondition have on the client of loopysqrt()? (2 pts.)

The removal of the precondition did not change my code, this implication only affects outcomes when n is negative. When n is negative, the while loop never runs since a becomes negative and less than 0 by default. Then the if condition notice 0*0 does not equal a negative number and outputs as -1, meaning invalid input.

h. Use computational induction to prove by hand the total correctness of the final version of your Dafny code. (6 pts.)

Base case: n = 0
Root = 0 → 0 * 0 = 0

Assume P(k), where k is some int that is a perfect square of (m)

N = k
Root = m → m * m = k

3. Array of differences

Below is the pseudocode for creating an array containing elements each of which is the difference between two adjacent elements of the input array:

```
Precondition: arr != null ∧ arr.Length > 0
int[] difference(int[] arr) {
  diffs ← new int[arr.Length - 1]
  a ← 0
  while (a < diffs.Length)}
  {
    diffs[a] ← arr[a + 1] - arr[a]
    a ← a + 1
  }
  return diffs
}
Postcondition: diffs.Length = arr.Length - 1 ∧
  ∀ k, s.t. 0 ≤ k < diffs.Length, diffs[k] = arr[k + 1] - arr[k]
```

a. Find a suitable loop invariant. (2 pts.)

Invariant $\boldsymbol{0 \leq a \leq diffs.length \wedge diffs.length == arr.length - 1 \wedge}$
$\boldsymbol{forall\, b :: (0 <= b < a) ==> diffs[b] == arr[b+1] - arr[b]}$

b. Show that the invariant holds before the loop (base case). (1 pt.)

Example: arr.length = 7.
Diffs = 7 – 1 = 6
A = 0

$\boldsymbol{0 \leq 0 \leq 6 \wedge 6 == 7 - 1 \wedge True}$ **==> True**

c. Show by induction that if the invariant holds after k-th iteration, and execution takes a k+1-st iteration, the invariant still holds (inductive step). (4 pts.)

Prove $\boldsymbol{0 \le a \le diffs.length \wedge diffs.length == arr.length - 1 \wedge}$
$\boldsymbol{forall\, b :: (0 <= b < a) ==> diffs[b] == arr[b+1] - arr[b]}$

Base case:
Arr.length = 2
Diffs = 1
A = 0
$\boldsymbol{0 \le 0 \le 1 \wedge 1 == 2 - 1 \wedge True}$ **==> True**

Assume $\boldsymbol{0 \le a \le diffs.length \wedge diffs.length == arr.length - 1 \wedge}$
$\boldsymbol{forall\, b :: (0 <= b < a) ==> diffs[b] == arr[b+1] - arr[b]}$

Prove a+1:

$\boldsymbol{0 \le a + 1 \le diffs.length,}$
would hold true because if a = $\boldsymbol{diffs.length}$**, the while loop wouldn't reiterate**

$\boldsymbol{forall\, b :: (0 <= b < a) ==> diffs[b] == arr[b+1] - arr[b]}$
Holds true since $\boldsymbol{diffs.length == arr.length - 1}$ **so arr has more indexes than diffs, so diffs has enough indexes from arr to compute and store the difference.**

Thus, the K+1 iteration holds true.

d. Show that the loop exit condition and the loop invariant imply the postcondition diffs.Length = arr.Length − 1 ∧ ∀k, s.t.0 ≤ k < diffs.Length, diffs[k] = arr[k + 1] − arr[k]. (1 pt.)

Arr = [1,2,3,4,5,6]

By the end of the while loop, diffs = [1,1,1,1,1]

Thus the postconditon is true because for each index of diffs, it can be computed by the same index in arr subtracted from the k-1 index.

It is also true because the invariant statement implies the post condition in the statement:

forall b :: (0 <= b < a) ==> diffs[b] == arr[b + 1] - arr[b

e. Find a suitable decrementing function. Show that the function is not negative before loop starts, that it decreases at each iteration and that when it reaches 0 the loop is exited. (2 pts.)

decreases diffs.Length – a

Since a starts out equal to 0, and diffs.Length is larger than a to start the while loop, then (diffs.Length – a) > 0 and would then decrement down to 0 to stop.

4. The Simplified Dutch National Flag Problem

a. Given an array arr[0..N-1] where each of the elements can be classified as red or blue, write pseudocode to rearrange the elements of arr so that all occurrences of blue come after all occurrences of red and the variable k indicates the boundary between the regions. That is, all arr[0..k-1] elements will be red and elements arr[k..N-1] will be blue. You might need to define method swap(arr, i, j) which swaps the ith and jth elements of arr. (7 pts.)

```
Method dutch_flag (arr: array?<char>) returns (k: int){
  k := 0;
  index := 0;
  While (index < arr.Length){
    If (arr[index] == 'r'){
      Swap(arr, index, k);
      k = k + 1;
    }
    index = index + 1;
  }
}
```

b. Write an expression for the postcondition. (2 pts.)

$$\mathbf{0 \le k \le arr.Length} \wedge$$
$$\mathbf{Forall\, l :: l :: 0 \le l < k ==> arr[l] == 'r'} \wedge$$
$$\mathbf{forall\, m :: k <= m < arr.Length ==> arr[m] == 'b'}$$

c. Write a suitable loop invariant for all loops in your pseudocode. (4 pts.)

We must show that everything in the array is either red or blue since those are the only two colors mentioned. On top of that, up to index k-1 should be red, and k to arr.Length – 1 should be blue. As well as k being within the right range.

$$\mathbf{0 <= k <= index <= arr.Length} \wedge$$
$$\mathbf{forall\, n :: 0 <= n < k ==> arr[n] == 'r'} \wedge$$
$$\mathbf{forall\, m :: k <= m < i ==> arr[m] == 'b'} \wedge$$
$$\mathbf{forall\, p :: i <= p < arr.Length ==> (arr[p] == 'b' \vee arr[p] == 'r')}$$

d. Implement your pseudocode in Dafny. (2 pts., autograded)