CS 577: Introduction to Algorithms 9/14/2015

Homework 1 Solutions (Review Problems)

Instructor: Dieter van Melkebeek TA: Kevin Kowalski

Problem 1

Part (a)

An adequate loop invariant is

$$c \geq 0 \text{ is an integer and } r \cdot c! = n!. \tag{1}$$

Let r_i and c_i be the values of r and c respectively just before line 4 is executed for the ith time. We prove this by induction on i. The base case is $i = 1$. Then $r_i = r_1 = 1$ and $c_i = c_1 = n$, an integer. Since $n \geq 0$, we have $c_i = n \geq 0$. Thus $c_i \geq 0$ is an integer (since n is an integer). Also, $r_i \cdot c_i! = 1 \cdot n! = n!$ as needed.

For the inductive step, assume for some $i \geq 1$

1. that $c_i \geq 0$ is an integer and
2. that $r_i \cdot c_i! = n!$.

We need to show

1. that $c_{i+1} \geq 0$ is an integer and
2. that $r_{i+1} \cdot c_{i+1}! = n!$.

Since line 4 is about to be executed for the $(i+1)$th time, its condition was true during the ith execution, so we have $c_i > 0$. Since c_i is an integer by item 1 of our inductive hypothesis, we further have $c_i \geq 1$. From line 5 and line 6, we have $r_{i+1} = r_i c_i$ and $c_{i+1} = c_i - 1$ respectively. Therefore, $c_{i+1} \geq 0$ is an integer and

$$\begin{aligned} r_{i+1} \cdot c_{i+1}! &= r_i c_i (c_i - 1)! && \text{(definitions of } r_{i+1} \text{ and } c_{i+1}) \\ &= r_i \cdot c_i! && \text{(definition of factorial function)} \\ &= n! && \text{(item 2 of induction hypothesis)} \end{aligned}$$

as needed.

Part (b)

We have the loop invariant given in (1). Since we did not enter the while loop, the condition $c > 0$ in line 4 is false. Thus $c \leq 0$. From (1), we have $c \geq 0$, so $c = 0$. Then it follows that

$$\begin{aligned} n! &= r \cdot c! && \text{(by (1))} \\ &= r \cdot 0! && \text{(definition of } c) \\ &= r \cdot 1 && \text{(definition of factorial function)} \\ &= r. \end{aligned}$$

Therefore, the correct value $r = n!$ is returned.

Algorithm 1

Input: an integer $n \geq 0$
Output: $n!$

```
1: procedure Factorial(n)
2:     if n = 0 then
3:         return 1
4:     else
5:         return Factorial(n − 1) · n
```

Algorithm 2

Input: an integer $n \geq 0$
Output: $n!$

```
1: procedure Factorial(n)
2:     return RecFactorial(1, n)
```

Input: $c \geq 0$ is an integer
Output: $rc!$

```
3: procedure RecFactorial(r, c)
4:     if c = 0 then
5:         return r
6:     else
7:         return RecFactorial(rc, c − 1)
```

Part (c)

The specification and implementation for a recursive version of the factorial function are given in Algorithm 1 as the procedure Factorial. We also give a tail recursive version in Algorithm 2 as the procedure RecFactorial.

Part (d)

It is slightly more work to prove the correctness of Algorithm 2, so we give that proof. For the purposes of providing another example, we also prove its correctness in two ways: (1) proving partial correctness and termination, and (2) proving (full) correctness at once via structural induction.

Proof via Partial Correctness and Termination

Partial correctness We have to show that all calls to RecFactorial have valid inputs and both Factorial and RecFactorial return correct values assuming they return at all. The call to RecFactorial on line 2 has a valid input since $n \geq 0$. The other call to RecFactorial is on line 7. Since we are in the else branch, it must be that the condition $c = 0$ is false, so $c \neq 0$. The precondition says $c \geq 0$ is an integer, so we have $c \geq 1$. Then clearly $c - 1 \geq 0$ is a valid input.

If RecFactorial returns the correct value on line 2, then Factorial returns $1 \cdot n! = n!$, which is correct. It suffices to show that RecFactorial returns the correct value if it terminates.

If $c = 0$, then on line 5 we return $r = r \cdot 1 = r \cdot 0! = r \cdot c!$, which is the correct value. Otherwise,

$c \geq 1$ (as argued above), and we return the recursive call on line 7, which we can assume returns $rc(c-1)! = rc!$, which is also the correct value.

Termination The algorithm terminates if $c = 0$. In every recursive call, the value of c is decreased by 1. Since $c \geq 0$ is also an integer, this will eventually happen.

Proof via Structural Induction

We prove (full) correctness at once via induction on c. The base case is $c = 0$. Then on line 5 we return $r = r \cdot 1 = r \cdot 0! = r \cdot c!$, which is the correct value. Otherwise, $c \neq 0$ and we return the recursive call on line 7. The precondition says $c \geq 0$ is an integer, so we have $c \geq 1$. Clearly $rc \geq r \geq 1$ and $c - 1 \geq 0$ are valid inputs to the recursive call. Therefore by our induction hypothesis, we return $rc(c-1)! = r \cdot c!$, which is also the correct value.

Problem 2

Part (a)

Intuitively, BinarySearch works by iteratively dividing the list in half and either discarding the bottom half if x is greater than its largest element, or discarding the top half otherwise. In other words, in each iteration of the while loop, the procedure discards the half of the list that is guaranteed to not contain the first occurrence of x (if any).

Formally, an invariant for the loop is

$$\begin{gathered} i \text{ and } j \text{ are integers satisfying } 0 \leq i \leq j \leq n-1 \text{ and} \\ \text{if } x \text{ is in } A, \text{ then } x \text{ is in the subarray } A[i..j] \text{ but not in the subarray } A[0..i-1]. \end{gathered} \tag{2}$$

Let i_t and j_t be the values of the variables i and j respectively just before the tth execution of line 4. We prove (2) by induction on t. The base case is $t = 1$. Then $i_t = i_1 = 0$ and $j_t = j_1 = n-1$. Since $n \geq 1$, this proves the first part of 2. If x is in A, then x is in the subarray $A[0..n-1] = A[i_t..j_t]$. Since the subarray $A[0..i_t - 1] = A[0.. -1]$ is empty, this proves the second part of (2).

For the inductive step, assume for some $t \geq 1$

1. that i_t and j_t are integers satisfying $0 \leq i_t \leq j_t \leq n-1$ and
2. that if x is in A, then x is in the subarray $A[i_t..j_t]$ but not in the subarray $A[0..i_t - 1]$.

We need to show

1. that i_{t+1} and j_{t+1} are integers satisfying $0 \leq i_{t+1} \leq j_{t+1} \leq n-1$ and
2. that if x is in A, then x is in the subarray $A[i_{t+1}..j_{t+1}]$ but not in the subarray $A[0..i_{t+1} - 1]$.

Since line 4 is executing for the $(t+1)$th time, its condition was true during the tth execution, so we have $i_t < j_t$. Let $m = \lfloor (i_t + j_t)/2 \rfloor$. Then

$$0 \leq i_t \leq m < j_t \leq n-1, \tag{3}$$

so $0 \leq m \leq n-1$.

We do a case analysis on the condition in line 6. If $A[m] < x$, then $i_{t+1} = m + 1 \geq 0$ and $j_{t+1} = j_t \leq n-1$, which are integers. From (3), we have $m < j_t$, and both are integers, so we further have $m + 1 \leq j_{t+1}$. Thus $i_{t+1} \leq j_{t+1}$. This proves item 1. If x is in A, then by item 2 of our inductive hypothesis, x is in the subarray $A[i_t..j_t] = A[i_t..j_{t+1}]$. But since $A[m] < x$ and A

is sorted from smallest to largest, we know that x is not in the subarray $A[0..m] = A[0..i_{t+1} - 1]$. Therefore, x is in the subarray $A[m+1..j_{t+1}] = A[i_{t+1}..j_{t+1}]$. This proves item 2.

Otherwise $A[m] \geq x$. Then $i_{t+1} = i_t \geq 0$ and $j_{t+1} = m \leq n - 1$. From (3), we have $i_{t+1} = i_t \leq m = j_{t+1}$, so $i_{t+1} \leq j_{t+1}$. This proves item 1. If x is in A, then by item 2 of our inductive hypothesis, x is in the subarray $A[i_t..j_t] = A[i_{t+1}..j_t]$ but not in the subarray $A[0..i_t - 1] = A[0..i_{t+1} - 1]$. Since $A[m] \geq x$ and A is sorted from smallest to largest, we know that x is in the subarray $A[i_{t+1}..m] = A[i_{t+1}..j_{t+1}]$. This proves item 2 and completes the proof of (2).

Part (b)

Suppose that we have reached line 10. Since we did not enter the loop, its condition $i < j$ on line 4 is false. Thus, $i \geq j$. From (2), we have $0 \leq i \leq j \leq n - 1$, so we must have $i = j$, which is a valid index of A.

If x is in A, then by (2), x is in the subarray $A[i..j] = A[i..i]$. Thus $A[i] = x$. Also by (2), x is not in the subarray $A[0..i - 1]$, so i is the smallest index satisfying $A[i] = x$. Therefore, returning i on line 11 is the correct behavior.

Otherwise x is not in A. Then the condition on line 10 is false, so the procedure correctly returns -1 on line 13.

Part (c)

Algorithm 3

Input: an integer $n \geq 1$, an array $A[0..n-1]$ of integers that is *sorted* from smallest to largest, an integer x, and integers i and j satisfying $0 \leq i \leq j \leq n - 1$

Output: -1 if x does not appear in the subarray $A[i..j]$, otherwise the smallest index k such that $i \leq k \leq j$ and $A[k] = x$

```
1:  procedure RecBinarySearch(n, A, x, i, j)
2:      if i = j then                                          ▷ base case
3:          if A[i] = x then
4:              return i
5:          else
6:              return −1
7:      else                                                   ▷ recursive case
8:          m ← ⌊(i + j)/2⌋
9:          if A[m] < x then
10:             return RecBinarySearch(n, A, x, m + 1, j)
11:         else
12:             return RecBinarySearch(n, A, x, i, m)
```

The procedure in Algorithm 3 is our recursive version of BinarySearch. The basic idea behind our conversion is that we want to replace the iterative halving of the relevant array indices with recursive halving.

More specifically, instead of reassigning the values of i and j on each loop iteration, we recursively call our procedure with new values of i and j. This necessitates adding i and j to the list of

inputs for the procedure. Invoking RecBinarySearch$(n, A, x, 0, n-1)$ is equivalent to invoking BinarySearch(n, A, x).

Part (d)

We can prove the correctness of RecBinarySearch by induction on $i - j$. More specifically, let $S(t)$ be the statement that RecBinarySearch terminates with the correct output on all inputs (n, A, i, j, x) that satisfy $i - j \leq t$. Then, if we can show that $S(t)$ is true for all t, this automatically implies that RecBinarySearch terminates with the correct output on all possible inputs.

Base case: If $i - j = 0$, then either $A[i] = x$ or $A[i] \neq x$. In either case, we trivially return the correct answer.

Induction step: Now, assume $S(t)$, so to complete the induction it suffices to show $S(t+1)$. Let (n, A, i, j, x) be an input such that $i - j = t + 1$.

Since $t + 1 > 0$, $i \neq j$ so we immediately fall to the else block that starts on line 7. Let $m = \lfloor (i + j)/2 \rfloor$. We then have two cases.

First, assume that $A[m] < x$. In this case,

$$\text{RecBinarySearch}(n, A, i, j, x) = \text{RecBinarySearch}(n, A, m + 1, j, x).$$

Now, note that m is the mean of i and j, rounded down, so $m \geq i$ and $m + 1 > i$. Hence, $j - (m + 1) \leq t$ (since $j - i = t + 1$), so we can apply our inductive hypothesis to get that RecBinarySearch$(n, A, m+1, j, x)$ returns -1 if x does not appear in $A[m+1..j]$, or the smallest index k at which x does appear in this range.

Since $A[m] < x$, we know that x cannot be present in the subarray $A[i..m]$, so x is present in $A[i..j]$ if and only if it is present in $A[m + 1..j]$. Thus,

$$\begin{aligned}\text{RecBinarySearch}(n, A, i, j, x) &= \begin{cases} -1 & \text{if } x \notin A[m+1..j], \\ k & \text{otherwise,} \end{cases} \\ &= \begin{cases} -1 & \text{if } x \notin A[i..j], \\ k & \text{otherwise,} \end{cases}\end{aligned}$$

where k is the smallest index at which x appears in $A[m+1..j]$ (and thus the smallest at which it appears in $A[i..j]$). This exactly matches the output specification, so in this case, RecBinarySearch terminates with the correct answer, as desired.

The remaining case $A[m] \geq x$ differs from the $A[m] < x$ case only in that the indices we recurse on are $i..m$ instead of $m + 1..j$. Since $m - i \leq t$, we know that RecBinarySearch(n, A, i, m, x) returns -1 if x does not appear in $A[i..m]$ and the smallest index at which x appears otherwise, and since $A[m] \geq x$, we know that x appears in $A[i..j]$ if and only if it appears in $A[i.m]$. Hence, RecBinarySearch(n, A, i, j, x) returns the correct answer in all possible cases, completing the induction.