Tom Fenyak

CPSC 3273 Homework Assignment 02

Exercise 1 (15 points)

Consider this algorithm ***Mid(a,b):***

**Input**: two integers a and b

**Output**: an integer that is the midpoint between a and b.

**Example**: Mid(4,10) = 7

Mid(a,b)

m = (a + b) >> 1

return m

1. (5 points) Is this algorithm correct? (Answer only Yes or No)

* Yes

1. (10 points) Whatever is your answer, prove it using the **fastest** method: a counterexample. A proof by contradiction, **or** a proof by induction.

* The fastest way of proving this answer is proof by contradiction and in order to prove that Mid(4,10) != 7 is false to show that our algorithm is correct.
* When you input (4,10) and add 4 + 10 and then use the shift bitwise operator >>1 it gives us the midpoint of 7. Which makes our contradiction statement of Mid(4,10) != 7 to be false, therefore our algorithm is correct.

Exercise 2 (25 points) Proof by contradiction

Consider a program *P* that contains two threads of execution. In a simplistic way, it means that two routines *Thread1* and *Thread2* in Program *P* can be running concurrently. The integer variable *Turn* is initialized to 0 by Program *P*. The variable Turn is shared by the threads *Thread1* and *Thread2*. Below are the codes for *Thread1* and *Thread2*.

|  |  |
| --- | --- |
| **Thread 1** | **Thread 2** |
| while (1) {  while (Turn == 1) ; //loop here while Turn == 1  Code A  Turn = 1;  } | while (1) {  while (Turn == 0) ; //loop here while Turn == 0  Code B  Turn = 0;  } |

Code A and Code B are blocks of multiple instructions.

Pay attention: Code A and Code B are not part of the *while (Turn ==..)* loops. For example, if *(Turn == 1)* for Thread 1, this while loop keeps looping and Code A will not run unless the variable *Turn* becomes 0.

We assume that Code A and Code B do not modify the variable *Turn*. Answer the following questions:

1. (2 points) When instructions of Code A are running, what is the value of *Turn*?

* Turn = 0 because we need to exit the while loop to produce Code A.

1. (2 points) When instructions of Code B are running, what is the value of *Turn*?

* Turn = 1 because we need to exit the while loop to produce Code B.

1. (4 points) Can Code A and Code B be running simultaneously? Answer and justify your answer

* No they cannot be running simultaneously because while Turn = 0, Code B will not be produced, and while Turn = 1, Code A will not be produced. This occurs because of the while loops that prevent the program reaching Code A and B based off of the value of Turn.

1. Use a proof by contradiction to show that Code A and Code B CANNOT be running simultaneously. For this proof, I suggest to follow these steps:
   1. (4 points) What will be your starting assumption?

* We will start by assuming if Turn = 0 🡪 Code A and Turn = 1 🡪 Code B.
  1. (10 points) Can you infer from this assumption some contradiction?
* For a proof by contradiction we must assume the opposite.
* So, if Turn = 0 🡪 !Code A and Turn = 1 🡪 !Code B.
* First we prove that Turn = 0 🡪 !Code A is false, this is pretty clear because if we are not producing Code A, then we are supposed to produce Code B, and we know from the while look that Code B will never be reached while Turn = 0 because the while loop will continue until Turn != 0 in Thread 2. This contradiction is False.
* In return we can prove that Turn = 1 🡪 !Code B in the same manner because !Code B would be Code A, and we know that while Turn = 1, Code A will not be reached because the while loop above it will continue as long as Turn = 1. This contradiction is False
  1. (3 points) After you show the contradiction, what will be your conclusion?
* By proving that both cases of our contradiction are False, it means that the assumption that Code A will be produced while Turn = 1 AND Code B will be produced while Turn = 0 is correct.

Exercise 3 (25 points) Proof by induction

Let us prove this formula:

1. (4 points) Try this expression with a = 2 and n = 4. Find *S* using the two expressions (the sum and the fraction).

* Sum 🡪 S = 2^0 + 2^1 + 2^2 + 2^3 +2^4

S = 31

* Fraction 🡪 S = (1 – 2^(4+1)) / (1 – 2)

S = 31

1. Let us prove this expression using induction
   1. (5 points) Show the base (basis) case

* Base case is when i = 0 S(0) should equal 1.

So a^0 == (1-a^(0+1)) / (1 – a) or a^0 == (1-a) / (1-a) which == 1.

This proves our base case.

* 1. Show the induction step by answering these questions:
     1. (4 points) What is your hypothesis to use for the induction step?
* My hypothesis for the induction step is:

We assume that the property where S = the sum = the fraction holds true for n and will continue to hold true for (n+1). In other words as we increase n each “step up the ladder” (adding 1 to n) our formula will still be true.

* + 1. (12 points) Now, complete the induction step
* We want to prove that S(n+1) = (a^0 + a^1 + a^2 ……a^n) + a^n+1

We can simplify this to (1 – a^(n + 1)) / (1 – a) + a^n+1

Even further to (1-a^(n+1) + a^(n+1) – a^(n+2) / (1 – a)

And finally (1-a^(n+2)) / (1-a) which is what we would expect from S(n+1)

Exercise 4 (35 points) Loop Invariant

Consider the naïve sorting algorithm we presented in Module 1:

Sort-Array(A)

for j = 1 to (A.length – 1)

for i = (j + 1) to A.length

if (A[i] < A[j])

// swap A[i] and A[j]

buffer = A[j]

A[j] = A[i]

A[i] = buffer

The objective is to prove that the above sorting algorithm is correct. Consider getting inspiration from the textbook in Section 2.1: the author shows that the *Insert Sort* algorithm is correct using loop invariants. This should help you with this exercise.

1. (2 points) Express the property that Sort-Array(A) must satisfy to be correct:

* When comparing elements at index [i] and index [i +1], [i] must be less than or equal to [i+1] for 1 to A.length.

1. (4 points) Can you find some loop invariants for the outer for loop? List these invariants (even if they are not that helpful for our ultimate proof of correctness of Sort-Array)

* A[i] >= A[j] for all j from 1 to A.length
* j <= A.length - 1
* i <= A.length
* A[i] >= A[1]

1. (9 points) Propose a loop invariant for the outer loop that is the closest to our ultimate objective: Sort-Array is correct.

* A[i] >= A[j] for all j from 1 to A.length because we want each element in the array at index – 1 to be less than index from index 0 to A.length – 1.

1. Use the three steps:
   1. (4 points) Initialization

* We have to show that the loop invariant holds true before the first loop. When j = 1 the subarray consists of one single element which is the initial element in A[0] which proves that the loop invariant holds true before the first iteration.
  1. (10 points) Maintenance
* Here we have to show that A[i] >= A[j] for all j from 1 to A.length so for A[j+1], A[j+2], A[j+3] until you reach A.length. The subarray now consists of [1….j] where each element at index[j] is less than index[j+1]. Before the next iteration we must check that j <= A.length – 1 and that i <= A.length. We will update by swapping A[i] and A[j] A[i] if it is less than A[j].
  1. (6 points)Termination
* Here we look at what happens when the loop is completed. The condition that causes the loop to terminate is when j > A.length – 1. The array will be made of of elements from A[1] to A[n] but in sorted order.