Lab 4 Question 1

1. Part A: Prove strange\_fib is correct

To show this, we must show that the three (well really 4) correctness conditions in 5.2 of the textbook hold.

**Condition 1: For any call of the function, its precondition holds.**

In the first call of this function, by the definition of the function, we can say the precondition holds.

In the two recursive calls of the function highlighted and underlined below, we can say first that n >= 4 because of the if-else rule: since no value has been returned and the final ‘else’ has been reached, this means the first if and the elif have both failed. We can also say n is an integer since it has passed the precondition above. A closer look at the n>=4 idea is as follows. Since n, an integer > 0, is not <= 2 and not = 3, it must be >= 4.

Now, since n >=4, the arguments with which the strange\_fib is called satisfy the precondition. Since n is an integer and n >=4, we know [(n-1) >= 3 and (n-1) is an integer] and [(n-3) >= 1 and (n-3) is an integer]. As a result, both (n-1) and (n-3) satisfy the precondition that they must be integers and they must be >0.

In short, both recursive calls have arguments that satisfy the precondition.

def strange fib(n):

precondition(is integer(n) and n>0)

# postcondition: return fib(n), where

# fib(1)=1, fib(2)=1, and fib(n)=fib(n-1)+fib(n-2) when n>2

# alternate postcondition: returns fib(n), where

# fib is the function from the previous lab

if n <= 2:

return 1

elif n == 3:

return 2

else:

# The following line deserves a better comment than this

return 2\*strange fib(n-1) - strange fib(n-3)

**Condition 2: The value for each return statement fits the postcondition.**

Here we decided to use the first postcondition: postcondition: return fib(n), where fib(1)=1, fib(2)=1, and fib(n)=fib(n-1)+fib(n-2) when n>2.

There are three return statements.

First Return

If we are about to use the first return (return 1), n must be 2 or 1 because the return is inside a block with a Boolean that n<= 2 (meaning the Boolean is true) and the precondition requires n to be an integer > 0.

If n is 2, we have:

Postcondition(1 == fib(2))

 substitute fib(2) for 1 because of the definition of the postcondition’s fib(n)

Postcondition(1 == 1)

 High school logic – Boolean algebra

Postcondition(True)

If n is 1, we have

Postcondition(1 == fib(1))

 substitute fib(1) for 1 because of the definition of the postcondition’s fib(n)

Postcondition(1 == 1)

 High school logic – Boolean algebra

Postcondition(True)

Second Return

For the second return (return 2), the elif block we are in tells us that n must equal 3. Thus:

Postcondition(2 == fib(3))

 using the definition of fib(n) in the postcondition

Postcondition(2 == fib(2) + fib(1))

using the definition of fib(n) for both fib(1) and fib(2) to substitiute

Post condition(2 == 1 + 1)

hs math and Boolean algebra

Postcondition(True)

Third Return

To show the third return (return 2\*strange fib(n-1) - strange fib(n-3)) satisfies the post condition, we use the recursive function substitution rule (which uses induction) that states we can substitute a call of a function that is recursive if we can show that a)the precondition holds and that b)some progress is made towards reaching a base case.

So:

1. We have shown above in condition 1, that the precondition holds for these recursive calls.
2. If we consider n to be the progress expression, we see that n approaches a base case where n<4. This is because n is an integer and both the recursive calls, strange\_fib(n-1) and strange\_fib(n-3) decrement n by 1 or more. Thus, n, which is >= 4 (because it is in that else block and all the other if or elif blocks have had negative Boolean tests) must approach a value <4. Hence, each recursive call makes progress.

Since each recursive call satisfies the precondition and shows progress, we can substitute the postcondition. Thus we have:

Postcondition(2\*strange\_fib(n-1) – strange\_fib(n-3) == fib(n))

Since, at this return n >= 4, fib(n) = fib(n-1) + fib(n-2)

Postcondition(2\*strange\_fib(n-1) – strange\_fib(n-3) == fib(n-1) + fib(n-2))

 As we have shown above, these calls follow the precondition and make progress so the post condition can be substituted.

Postcondition(2\*fib(n-1) - fib(n-3) == fib(n-1) + fib(n-2))

Here we do some HS math:

We know fib(n-1) = fib(n-2) + fib(n-3)

Thus:

fib(n-2) = fib(n-1) – fib(n-3)

thus, we can substitute the *fib(n-2)* in [ fib(n-1) + *fib(n-2)* ] with *fib(n-1) – fib(n-3)*

so [ fib(n-1) + fib(n-1) – fib(n-3) ] is [ 2\*fib(n-1) – fib(n-3) ]

Postcondition(2\*fib(n-1) – fib(n-3) == 2\*fib(n-1) - fib(n-3) )

Because by definition, these are the same, we can say with boolean arithmetic, that it evalutates to True.

Postcondition(True)

**Condition 3: The recursive function makes progress in each recursive call.**

We have already shown this above- since n is an integer >=4, and is decrementing by at least 1 in both recursive calls, n eventually reaches a values <= 3 (or <4) which are the base cases.

1. Part B: Explain the role of elif n ==3: return 2

If this line was absent, the call strange\_fib(3) would have a recursive call { return 2\*strange\_fib(3-1) – strange\_fib(3-3) }, or equivalently { return 2\*strange\_fib(2) - strange\_fib(0) }. The latter call, strange\_fib(0) violates the precondition of the function that requires n>0 (and 0 is not > 0). Further, what would happen, provided the precondition did not interrupt the function, is that strange\_fib(0) would return a value of 1 while strange\_fib(2) would also return 1, making the call strange\_fib(3) return 1\*\*2 -1 = 0 though strange\_fib(3) should return 2.

1. Part C: Suggest a Better Comment Before The Third Return

A more helpful comment to understand the validity of the last return would be as follows (s\_fib is abbreviated for strange\_fib):

# since s\_fib(n-1) – s\_fib(n-3) = s\_fib(n-2), we can substitute s\_fib(n-2) in s\_fib(n-1) + s\_fib(n-2) as follows: