Assignment1 CPSC331

Date: May 21, 2019.

Group: Boxiao Li 30069613 \ Chunyu Li 30056553 \ Sijia Yin 30049836

1. If we want to prove he function is a bound function for this recursive algorithm. Then we need to proof those properties satisfied:

   "*(1). Recursive algorithm Mathematical function defined on inputs and global data of precondition*
   (2). Integer-valued
   (3). The function value decrease by at least one with every recursive call
   (4). If function value ≤ 0, then no recursive call".
   (*Proof bound function*, tutorial slide 03, Page 3.)

   Proof:

   a. *Precondition:* A non-negative integer n is given as input.
   b. The mathematical function f(n)=n is defined on the inputs of the recursive algorithm "Grindelwald Gaggle Computation".
   c. Since integer n is the only input that function needed, and there was no global data applied on "Grindelwald Gaggle Computation", therefore mathematical function defined on inputs and global data of precondition.
   d. From the pre-condition, we can see the type of inputs "n" is an integer since the function f assigns with integers, so we can say f is an integer-valued function.
   e. From line10 and line11, we could see that n is decreased by at least one with every recursive call.
   f. If the function value ≤0, which means n ≤ 0, but by the pre-condition, the value of input should ≥0, that is n≥0. Thus the only value for n to satisfies these two conditions(n≥0 and n≤0) is n=0; In line1 and line2, we found that when n equal to 0, then the function will return the value "1" and end the recursive call, which means there has no recursive call after that.
   g. Since the function satisfies all the properties of the bound function, so the function f(n)=n is a bound function for the recursive algorithm.

2. I will use mathematical induction to prove.

   Basis:

   From the pre-condition, input n should ≥0, I assume n equal to 0,1,2 and 3, by the line2, line4, line6, and line8, it returns 1, 2, 3 and 4. Then the algorithm ends. The value that sGrin algorithm returns are equal to the "Grindelwald Gaggle Computation's" output when we input the same value to "Grindelwald Gaggle Computation". It satisfies the post-condition.

   Inductive step:

   Let k be an integer such that k≥4.

   Inductive Hypothesis:

   Assume n=k as an input, then the algorithm will be executed, then the algorithm will eventually end and return the output. Output will be 2×sGrin(k−1)−2×sGrin(k−3)+sGrin(k−4) if k is an even number, and will be sGrin(k−1)+3×sGrin(k−2)−5×sGrin(k−3)+2×sGrin(k−4) if k is an odd number.

   Inductive Clam:

   We will prove the algorithm will execute when input n=k+1, and the algorithm will eventually end, the output will be 2×sGrin(k)−2×sGrin(k−2)+sGrin(k−3) if k is an even number, and will be sGrin(k)+3×sGrin(k−1)−5×sGrin(k−2)+2×sGrin(k−3) if k is an odd number.

   Proof Steps:

   Because we assume k≥4, thus n = k+1≥4, that will lead line1 to line8 failed, and come to line10 and line11. If k+1 is an even number, then line9 will be executed, by Inductive Hypothesis, the algorithm will eventually end and will return 2×sGrin(k)−2×sGrin(k−2)+sGrin(k−3). If k+1 is an odd integer, the algorithm will eventually execute and sGrin(k)+3×sGrin(k−1)−5×sGrin(k−2)+2×sGrin(k−3) will be given as the output. Then the algorithm ends.

   Proof closure:

   Since the execution ends and satisfied the post-condition of the algorithm, the sGrin algorithm correctly solves the "Grindelwald Gaggle Computation" problem.

3. See the Java file. The code provided here is copy from the Java file, which is handed in the D2L.

```
package cpsc331.assignment1;

import java.util.Scanner;

import java.io.IOException;


public class SGrindelwald{

      //An recursive algoritm which made for assignment1

      //It expect the user types one number as augument in the command line

      //The first number that user types is expected to be an positive intger

      //This recursive algorithm will take that positive integer as the input.

      //

      //The result for this algorithm will be printed as an output.

      //

      //Author: Sijia Yin 30049836; Chunyu Li 30056553; Boxiao Li 30069613

      public static void main(String[] args) {

            try{

                  if (args.length != 1) {

                        throw new NumberFormatException();

                  }

                  System.out.println(sGrin(Integer.parseInt(args[0])));

            }catch (NumberFormatException e) {

                  System.out.println("Gadzooks! One integer input is required.");

            }catch(IllegalArgumentException e){

                  System.out.println("Gadzooks! The integer input cannot be negative.");

            }

}

      //An recursive algorithm for "Extended Grindelwald Gaggle Computation" problem.

      //Precondition: An integer n is given as input.

      //Postcondition:If n is greater than or equal to 0, then the nth Grindelwald number, G_n,
is returned as out-put.

      //An IllegalArgumentException is thrown if input n is an negative number.

      //An NumberFormatException will be thrown if input n is not an number.

      //

      //Bound function:n

public static int sGrin(int n){

      //Assertion:n is an integer input such that n is greater than

   //or equal to zero.

      if (n < 0) {

      throw new IllegalArgumentException();

       }
      //Assertion: An integer n is given as an input
      //n should be greater or equal to 0;
      //An IllegalArgumentException will be thrown otherwise.
      if(n==0)
      //Assertion: n is an integer input with the value 0.
      {
            return 1;
      //Assertion:
      //1. n is an integer input with the value 0.
      //2. The nth Grindelwald number, G_n=G_0=1 will be printed as an output.
      }
      else if(n==1)
      //Assertion: n is an integer input with the value 1.
      {
            return 2;
      //Assertion:
      //1. n is an integer input with the value 1.
      //2. The nth Grindelwald number, G_n=G_1=2 will be printed as an output.
      }
else if(n==2)
      //Assertion: n is an integer input with the value 2.
      {
            return 3;
      //Assertion:
      //1. n is an integer input with the value 2.
      //2. The nth Grindelwald number, G_n=G_2=3 will be printed as an output.
      }
else if(n==3)
      //Assertion: n is an integer input with the value 3.
      {
       return 4;
      //Assertion:
      //1. n is an integer input with the value 3.
      //2. The nth Grindelwald number, G_n=G_3=4 will be printed as an output.
      }
else if((n % 2)==0)
      //Assertion: n is an integer input whose value is greater than 3,
      //and n is an even number(n divides 2 have no remainder).
      {
      return (2 * sGrin(n - 1)) - (2 * sGrin(n - 3)) + (sGrin(n - 4));
      //Assertion:
```

```
      //1. n is an integer input whose value is greater than 3,
      //and n is an even number(n divides 2 have no remainder).
      //2. The nth Grindelwald number, G_n=(2 * G_(n - 1)) - (2 * G_(n - 3)) + (G_(n - 4))
      //will be printed as an output.
      }
 else
      //Assertion: n is an integer input whose value is greater than 3,
      //and n is an odd number(n divides 2 have an remainder).
      {
      return (sGrin(n - 1) + 3 * sGrin(n - 2)) - (5 * sGrin(n - 3))+(2 * sGrin(n - 4));
      //Assertion:
      //1. n is an integer input whose value is greater than 3,
      //and n is an odd number(n divides 2 have an remainder).
      //2. The nth Grindelwald number, G_n=(G(n - 1) + 3 * G(n - 2)) - (5 * G(n - 3))+(2 * G(n
- 4))
      //will be printed as an output.
      }
  }
}
```

4. In order to obtain the runtime problem of this recursive algorithm, we will use the uniform cost criterion to give the recurrence for $T_{sGrin}(n)$. As described in the premise of the question, we will get the answer by the value of input $n \geq 0$:

When n = 0 and n as the input, the algorithm executes 2 steps (at lines1 and 2).

When n = 0 and n as the input, the algorithm executes 3 steps (at lines1, 3 and 4).

When n = 0 and n as the input, the algorithm executes 4 steps (at lines1, 3, 5 and 6).

When n = 0 and n as the input, the algorithm executes 5 steps (at lines1, 3, 5, 7and 8).

When n≥4 and n is an even number. Let n as the input, the algorithm executes 6 steps (at lines1, 3, 5, 7, 9 and 10). But it also has recursions, at line10, with input n-1, n-3, n-4.

When n≥4 and n is an odd number. Let n as the input, the algorithm executes 6 steps (at lines1, 3, 5, 7, 9 and 11). But it also has recursions, at line11, with input n-1, n-2, n-3, n-4.

$$T_{sGrin}(n) = \begin{cases} 2 & \text{if } n = 0, \\ 3 & \text{if } n = 1, \\ 4 & \text{if } n = 2, \\ 5 & \text{if } n = 3, \\ T_{sGrin}(n-1)+ T_{sGrin}(n-3)+ T_{sGrin}(n-4)+6 & \text{if } n\geq 4 \text{ and n is even,} \\ T_{sGrin}(n-1)+ T_{sGrin}(n-2)+ T_{sGrin}(n-3)+ T_{sGrin}(n-4)+6 & \text{if } n\geq 4 \text{ and n is odd.} \end{cases}$$

(*Recursive Algorithms: An Example*, from lecture slides L05, pages 26.)

5. To prove that T_{sGrin} (n) $\geq (3/2)^n$ for every non-negative integer n, we prove this by using strong mathematical induction.

Basis: (n = 0), then TsGrin = 2 $\geq$ 1 = $(3/2)^n$,
(n = 1), then TsGrin (n) = 3 $\geq$ 1.5 = $(3/2)^n$,
(n = 2), then TsGrin (n) = 4 $\geq$ 2.25 = $(3/2)^n$,
(n = 3), then TsGrin (n) = 5 $\geq$ 3.375 = $(3/2)^n$,

Inductive step: Suppose that for some integer k $\geq$ 3.

Inductive Hypothesis: Suppose that that for all integers m, $0 \leq m \leq k$, $T_{sGrin}(m) \geq (3/2)^m$.

Inductive Claim: T_{sGrin} (k+1) $\geq (3/2)^{k+1}$.

Proof:

Case1: when k is even

T_{sGrin} (k+1) = T_{sGrin}(k)+ T_{sGrin}(k-2)+ T_{sGrin}(k-3)+6
$\geq (3/2)^k + (3/2)^{k-2} + (3/2)^{k-3}+6$
$= (3/2)^{k+1}x(3/2)^{-1}+ (3/2)^{k+1}x(3/2)^{-3}+ (3/2)^{k+1}x(3/2)^{-4}+6$
$=(3/2)^{k+1}x[(3/2)^{-1}+ (3/2)^{-3}+ (3/2)^{-4}]+6$
$=(3/2)^{k+1}x(94/81)+6 \geq (3/2)^{k+1}$
Thus, for all even integers n $\geq$ 0, T_{sGrin} (n) $\geq (3/2)^n$.

Case2: when k is odd

T_{sGrin} (k+1) = T_{sGrin}(k)+ T_{sGrin}(k+1)+ T_{sGrin}(k-2)+ T_{sGrin}(k-3)+6
$\geq (3/2^{)k} + (3/2)^{k-1}+(3/2)^{k-2} +(3/2)^{k-3}+6$
$=(3/2)^{k+1}x(3/2)^{-1}+(3/2)^{k+1}x(3/2)^{-2}+(3/2)^{k+1}x(3/2)^{-3}+(3/2)^{k+1}x(3/2)^{-4}+6$
$=(3/2)^{k+1}x[(3/2)^{-1}+ (3/2)^{-2}+(3/2)^{-3}+ (3/2)^{-4}]+6$
$=(3/2)^{k+1}x(130/81)+6 \geq (3/2)^{k+1}$
Thus, for all odd integers n $\geq$ 0, T_{sGrin} (n) $\geq (3/2)^n$.

Thus, for all integers n $\geq$ 0, T_{sGrin} (n) $\geq (3/2)^n$.

6. The loop invariant for the algorithm provided is:

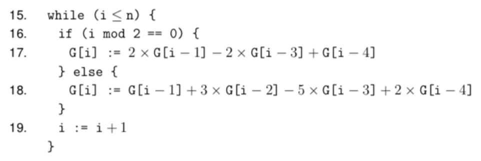
Loop invariant:

1. n is a non-negative integer $\geq$ 4.
2. G is an integer array, and its size is n+1.
3. i is an integer variable such that $4 \leq i \leq n$.
4. G[j] = G_j for every integer j such that $0 \leq j \leq i$.

(*Establishing a Loop Invariant*, from lecture slides L03, pages 7.)

Boxiao Li 30069613
Chunyu Li 30056553
Sijia Yin 30049836

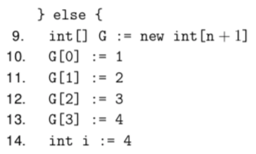
7. Prove the loop invariant in question 6:



```
15. while (i ≤ n) {
16.   if (i mod 2 == 0) {
17.     G[i] := 2 × G[i − 1] − 2 × G[i − 3] + G[i − 4]
      } else {
18.     G[i] := G[i − 1] + 3 × G[i − 2] − 5 × G[i − 3] + 2 × G[i − 4]
      }
19.   i := i + 1
    }
```

Prove that the test function of the loop has no side-effects: We can check the whole code that the algorithm with input integer n does not have any side-effects, which means that the algorithm does not access any global data, and it won't change the value of the input integer n. (*Establishing the Loop Invariant: A Simpler Proof*, from lecture slides L03, pages 26.)

Invariants are true at the beginning of the algorithm, that means invariants are satisfied with the precondition of the algorithm. To explain it, we know that to get in the while loop we need an input integer n ≥ 4. Now, we have an integer n as input with the value at least 4, then the line1, 3, 5, 7 will fail, and the step will go to else line. On line14, we get a i for value 4 (int i: = 4). On line10, we get G[0] = 1 which is G_0=G_{i-4}. On line11, we get G[1] = 2 which is G_1=G_{i-3}. On line12, we get G[2] = 3 which is G_2=G_{i-2}. On line13, we get G[3] = 4 which is G_3=G_{i-1}.



```
    } else {
 9.   int[] G := new int[n + 1]
10.   G[0] := 1
11.   G[1] := 2
12.   G[2] := 3
13.   G[3] := 4
14.   int i := 4
```

It can be concluded from the above explanation that the loop invariant is true at the beginning of the loop execution if the preconditions of the algorithm are satisfied. (*Establishing the Loop Invariant: A Simpler Proof*, from lecture slides L03, pages 26.)

The loop invariant is satisfied both at the start and the end of the loop. By proving the precondition of the start of the loop above, we are going to prove the loop invariant is also satisfied with the end of the algorithm. From above, we get n ≥ 4, and we know that during the loop the n value won't be changed, so that we can still get n≥4 at the end of the loop. At the last time to execute the loop, if i > n the loop will stop, so, the last time to execute the loop when i = n. During the final time execution, line19 will let i add 1. So, we have 4 ≤ i ≤ n+1. On the line16, it will separate the i value to even or odd, if i is even, then line17 will be executed. Then we have G[i]:=2×G[i−1]−2×G[i−3]+G[i−4], which can be written as G_i= 2×G_{i-1}−2×G_{i-3}+G_{i-4}, due to i = n in the final execution, we have G_i= G_n. It is similar to the odd number i. If i is even, then line18 will be executed. Then we have G[i]:=G[i−1]+3×G[i−2]−5×G[i−3]+2×G[i−4], which write as G_i=Gi-1+3×G_{i-2}−5×G_{i-3}+2×G_{i-4}, due to i = n in the final execution, we have G_i= G_n. Thus, at the end of the loop, we have i = n. (*Establishing the Loop Invariant: A Simpler Proof*, from lecture slides L03, pages 27.)

8. Pre-condition: a non-negative integer n has been given as an input. For proving that this algorithm is partially correct.
   We need to satisfy one of the two requirements.
   A). "*Execution of the algorithm eventually ends with the problem's precondition satisfied — and with no undocumented inputs or global data accessed, and no undocumented data being modified*." (*Partial Correctness,* From lecture slides L03, pages 29.)
   B) "*The execution of the algorithm never ends, at all*. (*Partial Correctness,* From lecture slides L03, pages 29.)
   If we let n be 0, 1, 2, 3, which satisfies the pre-condition: a non-negative integer as an input, then the line1,2 will be executed. Or line1 fails,3, 4will be executed. Or line1,3 fails, 4,5 will be executed. Or line1,3,5fails 7,8 will be executed. The algorithm will return the integer value which satisfies the post-condition. After we check the code, there has not global data or undocumented input access in the algorithm. As (A) needed.
   In the second case, if $n \geq 4$, and we also satisfy the pre-condition. lines 1,3,5,7 fails, 9,10,11,12,13,14 will be executed. Then it will reach the while loop, and satisfy the pre-condition of the algorithm, the algorithm will either execute or never execute. Because $n>4$ meet the needs for while loop executes, then the while loop will be executed. Then there only have two cases, either stop or never stop. If the algorithm never stops, then the situation satisfies the requirement that partial correctness needs. As (B) needed. If the algorithm eventually ends, then the loop invariant is satisfied. Thus, $i \leq n$, if line15 failed, that means $i \geq n$ then at this point, $i=n$ at the end of the while loop. $G[i]=G[n]$, so $G[n]$ has been given as the output. The post-condition is satisfied. As (A) needed.
   Since either requirement (A) or (B) is satisfied, the algorithm is partially correct.

9. The bound function for this while loop is f(n,i)=n-i in the algorithm.
   If we want to proof f(n,i)=n-i is a bound function for the while loop, we need to prove those properties satisfied.
   “a. Function is integer-valued, total function of some of the inputs, variables and global data that are accessed and modified when the loop is executed.
   b. When the loop body is executed, the value of this function is decreased by at least one before the loop’s test is checked again (if it is reached again, at all).
   c. If the value of this function is less than or equal to zero and the loop’s test is checked then the test fails (ending this execution of the loop).”
   (*Bound Functions for While Loops,* from lecture slide 03, pages 37)
   Proof:

   By the pre-condition, we could find that n is an integer input, and in line14, variable “i” is declared as an integer. Thus, both i and n are well defined before reach the while loop. And after we check the code, we can see that there has no global data accessed in the algorithm.

   “n” is not changed during the execution of the loop body, which means the value of n is keeping the same in every loop call. The value of the local variable “i” is increased by 1 for each loop run, we can find it in line19. Thus, for every loop run, the value of function f is decreasing at lease one.

   Suppose the value of the function is ≤0, which means n-i≤0, n≤i. That will cause line15—the loop test fails, and the loop will stop.

   Above sentences has satisfied all the properties that bound function need, thus, f(n,i)=n-i is a bound function.

10. Proof:

The precondition for the “Grindelwald Gaggle Computation” is satisfied. A non-negative integer n has been given as input.

If n = 0 the test at step1 checked, and the algorithm ends after step2 it returns 1.

If n = 1 the test at step1 fails, and test at step3 will be executed, and end in step4 return 2.

If n=2 the test at step 1 fails, the test of step3 fails and test at step5 passed and end in step6 return 3.

If n=3, the test of step1,3,5 fail and the test in step7 will be executed. The execution of the algorithm will end in step8.

If n≥4 the tests in step1,3,5,7 fail, then it will execute step9-14, and the while loop will be reached.

“Loop Theorem #2

a. Execution of the loop test has no side-effects.

b. If the problem’s precondition is satisfied when the execution of the algorithm including this loop begins, then every execution of the loop body ends.

c. A bound function for this while loop exists. It is obvious that in step15 is a comparison, and there are no undocumented side-effects. The body of this while loop (4 steps from steps16-19) it terminates when it is executed ”

(Proving Termination, from lecture slide 03, pages 40).

Thus, it satisfied the Loop Theorem#2. Every step in the execution of the while loop, beginning with the precondition of “Grindelwald Gaggle Computation” satisfied, when loop executed, it will end at step20.

11. Basic on question 8, we have proved this algorithm is partially correct. From question 10, we know the algorithm is terminated. Therefore, the algorithm is correctly. (*Proving Correctness,* from lecture slide 03, pages 44)

12. We suppose that n belong to Z, and the precondition is satisfied, then assume P(n) is the upper bound for the $T_{fGrin}(n)$.

    when n = 0, line1 will execute, and end in line2, the total executions are two steps.

    when n = 1, line1 test fails. Then to line3, it passes, and end in line4 the total executions are 3 steps.

    when n = 2, line1 and line 3 tests both fail, then to line5, it passes and ends in line6, the total executions are 4 steps.

    when n = 3, line1, 3,5 all are failed, then line7 will execute, end in line 8, total executions of are 5 steps.

    when $n \geq 4$, line1,3,5,7 are failed, then line9-14 will execute and the total executions are 10 steps.

    We prove the bound function is f(n,i) = n+1-i. And i = 4. Then it will change to the, f(n,i)= n+1-4= n-3.

    And the loop test is also executed one more time, therefore, the loop test should be n-3+1 = n-2. Next, the while loop has two conditions, 1) the loop will execute line16, 17, 19 or line16, 18, 19. each of condition has 3 steps. When the loop end, there is one more step in line20 as a return.

    So, the function can be shown as: 10+3x(n-3)+(n-2)+1 = 4n
    (*Recursive Algorithms: An Example*, from lecture slides L05, pages 26.)

$$p(n)=\begin{cases} 2 & n=0, \\ 3 & n=1, \\ 4 & n=2, \\ 5 & n=3, \\ 4n & n\geq 4. \end{cases}$$

13. See the Java file. The code provided here is copy from the Java file, which is handed in the D2L.

```
package cpsc331.assignment1;

import java.util.*;

import java.io.*;


public class FGrindelwald {

	//An recursive algoritm which made for assignment1

	//It expect the user types one number as augument in the command line

	//The first number that user types is expected to be an positive intger

	//This recursive algorithm will take that positive integer as the input.

	//

	//The result for this algorithm will be printed as an output.

	//

	//Author: Sijia Yin 30049836; Chunyu Li 30056553; Boxiao Li 30069613

	public static void main(String[] args){

		try{

			if (args.length != 1) {

				throw new NumberFormatException();

		}

			System.out.println(fGrin(Integer.parseInt(args[0])));

		} catch (NumberFormatException e) {

			System.out.println("Gadzooks! One integer input is required.");

		} catch (IllegalArgumentException e) {

			System.out.println("Gadzooks! The integer input cannot be negative.");

		}

	}


	//An recursive algorithm for "Grindelwald Gaggle Computation" problem.

	//Precondition: An integer n is given as input.

	//Postcondition:If n is greater than or equal to 0, then the nth Grindelwald number, G_n, 
is returned as out-put.

	//An IllegalArgumentException is thrown if input n is an negative number.

	//An NumberFormatException will be thrown if input n is not an number.

	//

	//Bound function:n-i


	public static int fGrin(int n){

	//Assertion:n is an integer input whose value is greater than

	//or equal to 0.

		if (n < 0) {
        throw new IllegalArgumentException();
        }
//Assertion: An integer n is given as an input
//n should be greater or equal to 0;
//An IllegalArgumentException will be thrown otherwise.

if(n == 0){
//Assertion: n is an integer input with the value 0.
        return 1;
//Assertion:
//1. n is an integer input with the value 0.
//2. The nth Grindelwald number,G_n=G_0=1 will be printed as an output.
}else if(n == 1){
//Assertion: n is an integer input with the value 1.
        return 2;
//Assertion:
//1. n is an integer input with the value 1.
//2. The nth Grindelwald number, G_n=G_1=2 will be printed as an output.
}else if(n == 2){
//Assertion: n is an integer input with the value 2.
        return 3;
//Assertion:
//1. n is an integer input with the value 2.
//2. The nth Grindelwald number, G_n=G_2=3 will be printed as an output.
}else if(n == 3){
//Assertion: n is an integer input with the value 3.
        return 4;
//Assertion:
//1. n is an integer input with the value 0.
//2. The nth Grindelwald number, G_n=T_G_3=4 will be printed as an output.
} else {
        int[] G = new int[n+1];
        G[0]=1;
        G[1]=2;
        G[2]=3;
        G[3]=4;
        int i = 4;
// Loop Invariant:
// 1. n is an integer input whose value is greater than or equal to 4.
// 2. G is an integer array who lenth is n+1.
// 3. variable i is an integer whose valueis greater than or equal to 4
```

```
            //   and less than or equal to n.
            // 4.G[j] is Gj, the jth Grindelwald number, for every integer j whose value
            // is betwen 0 and i (inclusive).
                  while (i <= n) {
                        if ((i % 2 ) == 0) {
                              G[i] = 2*G[i-1]-2*G[i-3]+G[i-4];
                              //Assertion:
                              //1. n is an integer input whose value is greater than 3;
                              //2. if n is an even number(n divides 2 have no remainder),
                              //then algorithm will given 2*G[i-1]-2*G[i-3]+G[i-4]
                              //printed as an output.
                        }else{
                              G[i] = G[i-1]+3*G[i-2]-5*G[i-3]+2*G[i-4];
                              //Assertion:
                              //1. n is an integer input whose value is greater than 3;
                              //2. if n is an odd number(n divides 2 have a remainder),
                              //then algorithm will given G[i-1]+3*G[i-2]-5*G[i-3]+2*G[i-4]
                              //printed as an output.
                        }
                        i++;
                  }
                  return G[n];
            }
      }
}
```

14. For this question:

We suppose that f(n) = n+1 is defined as the n^{th} Grindelwald number.

We prove that by using strong form induction:

Basis case: (n = 0, 1, 2, 3)

n = 0, f(0) = 0+1=1=$G_{0,}$

n = 1, f(1) = 1+1=2=$G_{1,}$

n = 2, f(2) = 2+1=3=$G_{2,}$

n = 3, f(3) = 3+1=4=$G_{3.}$

Inductive step: Suppose that for some integer $k \geq 3$.

Inductive Hypothesis: Suppose that that for all integers m, $0 \leq m \leq k$, G_m= f(m).

Inductive Claim: We want to prove G_{k+1}=f(k+1).

Proof:

Case 1: When k+1 is even

$G_{k+1}= 2G_k - 2G_{k-2} + G_{k-3}$

$= 2f(k) – 2f(k-2) + f(k-3)$ By (IH)

$=2(k+1) – 2(K-2+1) + (k-3+1)$

$=2k+2-2k+2+k-2$

$=k+2$

Thus, for all the even integer, it is satisfied.

Case 2: When k+1 is odd

$G_{k+1}= G_k+ 3G_{k-1} - 5G_{k-2} + 2G_{k-3}$

$= f(k)+3f(k-1) – 5f(k-2) + 2f(k-3)$ By (IH)

$=(k+1)+3(k-1+1)– 5(K-2+1) + 2(k-3+1)$

$=k+1+3k-5k+5+2k-4$

$=k+2$

Thus, for all the odd integer, it is satisfied.

Therefore, whenever k+1 is even or odd. It is satisfying. So, G_n=f(n) for every integer n as input. (*Algorithms with Loops: Two Unusual Things about This Example*, from lecture slides L05, pages 17.)