CPSC 221 Notes: Induction, Invariants, . . . , Solution-ish Stuff

June 17, 2015

1 Gray Codes

A binary "Gray Code" is a sequence of binary numbers each of which differs from the previous number by exactly one bit, where the last number in the sequence is consider to come before the first. Here's an example: `000 001 011 010 110 111 101 100`.

Here's a function that generates what are called "reflected Gray codes":

```
/* NOTE: Use these #includes: cassert, vector, string; compile with -g */
// Produce a gray code of 2^n non-repeating n-bit strings.
// Precondition: n >= 1.       (Just for fun: you can make this work with n >= 0 instead :)
std::vector<std::string> gray(int n) {
  std::vector<std::string> result;
  if (n == 1) {            // Sequence is [0, 1] when n = 1.
    result.push_back("0");
    result.push_back("1");
  } else {                 // Otherwise, get gray(n-1) and..
    std::vector<std::string> sub_result = gray(n-1);

    // ..add the elements of gray(n-1) with 0 prepended and..
    for (int i = 0; i < sub_result.size(); i++) // you can assume that this
      result.push_back("0" + sub_result[i]);    // loop does what it claims

    // ..add the elements of gray(n-1) in reverse order, with 1 prepended..
    for (int i = sub_result.size() - 1; i >= 0; i--) // you can assume that this
      result.push_back("1" + sub_result[i]);         // loop does what it claims
  }
  return result;
}
```

1. **Crucial** scratch work before **any** induction analysis: Circle the recursive calls.

   SOLUTION: Hard for me to circle. It's `gray(n-1)`, not the one in the comment :)

2. **Crucial** scratch work before **any** inductive analysis: Identify the base case(s) and recursive case(s). (A recursive case is a case with a recursive call. A base case is a case **without** a recursive call.)

   SOLUTION: The base case is the `if`'s then block. The recursive/inductive case is the `else` block.

3. Prove that this algorithm terminates in a finite number of steps. To do so: (1) state when the algorithm terminates and (2) show that when called on any valid argument, any sequence of recursive calls reaches a termination case in a finite number of steps.

   (We often skip this termination part, but it's important! We'll rely on it in all the proofs below. However, we'll do it very informally since we're really only using it to set up for below.)

   SOLUTION: Each recursive call reduces the integer n by 1. $n \geq 1$ (and finite, of course), and the base case is when $n = 1$; so, after a finite number of recursive calls, we will reach the base case.

4. Prove that every bitstring this produces is different (i.e., no two bitstrings in the result are identical to each other). To do so:

   (a) State the theorem clearly in terms of the argument(s).

   SOLUTION: For all integers $n \geq 1$, no two elements of `gray(n)` are equal.

   (b) **Crucial** scratch work: Figure out how and why your base case(s) work. (**Usually** easy.)

   SOLUTION: The base case puts two strings in that clearly aren't equal; so, that will satisfy the theorem. (Note that it **could** put zero, one, or three strings in and satisfy the theorem, as long as no two strings were identical. That wouldn't "break" our current theorem, but it would break some later theorems we want to prove.)

   (c) **Crucial** scratch work: Write down your theorem with the arguments to each recursive call substituted in. These will be your Induction Hypotheses!

   SOLUTION: "No two elements of `gray(n-1)` are equal." See how I didn't work hard **at all**? I just wrote down the theorem with the arguments for the recursive call substituted in!

   (d) **Crucial** scratch work: Figure out how and why your recursive case(s) work, assuming what you wrote down is already known to be true.

   SOLUTION: So, the recursive case prepends 0 onto every element of `gray(n-1)`. Those elements will still be unequal to each other. It then prepends 1 onto the reversed list `gray(n-1)`. Again, **those** elements will be unequal to each other. However, the key here is that the 0 at the front differs from the 1 at the front. So, no element from the first set will be equal to any element from the second set, either!

   (e) Fill in this proof structure using your notes above:
   SOLUTION: We filled it in using our notes above.

```
Theorem: For all n >= 1, no two elements of gray(n) are equal.
Base case: When n = 1 (which I just READ off the code for the base
   case, by the way!), the code produces ["0", "1"], which establishes
   the theorem because "0" != "1".

Inductive case: Consider the case when n > 1.

  Induction hypothesis: Assume the theorem holds for n - 1.

  Inductive step: The code produces the elements of gray(n-1) each
    with 0 prepended, followed by the elements of gray(n-1) (in
    reverse order) each with 1 prepended, which establishes the theorem
    because the first set of elements are all unequal (by the IH), the
    second set are all unequal for the same reason, and no two elements
    ACROSS the sets can be equal since everything in the first set
    starts with 0 while everything in the second starts with 1.

Termination: Each recursive call reduces the integer n by 1. n >= 1
    initially; so, a finite number of recursive calls will make n = 1
    and reach the base case.
```

   (f) You've proven your theorem. Make it match what you originally wanted to prove if it doesn't yet. (Not a problem in this case.)

   SOLUTION: Our proof by induction really does prove exactly what we wanted this time.

5. Prove that this produces 2^n bitstrings, each of length n. (How? Might **the steps above** help?)

   SOLUTION: We already know the base and recursive cases here. This time, the theorem is `gray(n)` produces 2^n bitstrings. (We're just going to prove it produces that many strings. The "bitstring" part

presumably means "all 0s and 1s", which we can prove by saying "every string operation in the function either creates a bitstring directly or adds only bits to an existing string; so the function can never create anything but a bitstring". This is actually an inductive argument as well, but it's a super-simple one!)

```
Theorem: For all n >= 1, gray(n) produces 2^n bitstrings
Base case: When n = 1, the code produces two strings ["0", "1"],
   which establishes the theorem because 2 = 2^1 = 2^n.

Inductive case: Consider the case when n > 1.

  Induction hypothesis: Assume the theorem holds for n - 1.

  Inductive step: The code produces gray(n-1) with 0 prepended
    and then gray(n-1) in reverse order with 1 prepended. That's
    twice as many strings as gray(n-1). gray(n-1) produces 2^(n-1)
    strings by the IH. So, the code produces 2*2^(n-1) strings,
    which establishes the theorem because:

      2*2^(n-1) = 2^(n-1+1) = 2^n

Termination: Each recursive call reduces the integer n by 1. n >= 1
       initially; so, a finite number of recursive calls will make n = 1
       and reach the base case.
```

6. Prove that the sequence of bitstrings produced form a Gray code.

   (You get to remember to use those steps and the structure yourself this time. No hints from us.)

   SOLUTION: Most things stay the same as earlier, but now we need to prove that (quoting the definition of Gray Codes at the start of this problem) `gray(n)` produces "a sequence of binary numbers each of which differs from the previous number by exactly one bit, where the last number in the sequence is consider to come before the first."

```
Theorem: For all n >= 1, gray(n) produces a Gray Code per the defitinion
         above.
Base case: When n = 1, the code produces ["0", "1"], which
    establishes the theorem because "0" differs from "1" in its only bit.

Inductive case: Consider the case when n > 1.

  Induction hypothesis: Assume the theorem holds for n - 1.

  Inductive step: The code produces gray(n-1) with 0 prepended
    and then gray(n-1) in reverse order with 1 prepended. Note
    that gray(n-1) is a Gray Code by the IH. So, this
    establishes the theorem because:

    (1) Each neighboring pair of strings wholly within the first
    set of strings differ from each other by one bit, by the IH. The
    prependend number matches in all these strings; so, it doesn't
```

```
        disturb the pattern.

        (2) The same is true for the second set of strings.

        (3) This leaves only two pairs: (a) the last string in the first
        set and the first string in the second set and (b) the last string
        in the second set and the first string in the first set. For (a),
        because the second set is reversed, BEFORE the prepending these
        two strings were identical. However, we prepended 0 to one and 1
        to the other, making them differ as required in exactly one bit.
        For (b), the same argument holds.

    Termination: Each recursive call reduces the integer n by 1. n >= 1
           initially; so, a finite number of recursive calls will make n = 1
           and reach the base case.
```

2 Three-Two-Three Sort

Here's a sorting algorithm... maybe?

```
// sorts numbers[lo..hi] in increasing order; numbers[lo..hi] includes
//   numbers[lo], numbers[lo+1], ..., numbers[hi-1], numbers[hi], but
//   numbers[x..x] has one element and numbers[x..(x-1)] is empty.
// precondition: 0 <= lo <= (hi+1) <= numbers.size()
void sort_helper(std::vector<int> & numbers, int lo, int hi) {
  int n = hi-lo+1;    // the length of numbers[lo..hi]
  if (n <= 1) return;
  else if (n == 2) {
    if (numbers[lo] > numbers[hi]) {
      int temp = numbers[lo];    // swap numbers[lo] and numbers[hi]
      numbers[lo] = numbers[hi];
      numbers[hi] = temp;
    }
  } else {
    int two_thirds_n = (int)ceil(2.0*n/3.0); // "reals" are clearer but dangerous
    sort_helper(numbers, lo, lo + two_thirds_n - 1); // sort first two-thirds
    sort_helper(numbers, hi - two_thirds_n + 1, hi); // sort last two-thirds
    sort_helper(numbers, lo, lo + two_thirds_n - 1); // sort first two-thirds
  }
}
void sort(std::vector<int> & numbers) { sort_helper(numbers, 0, numbers.size()); }
```

1. Go back and remind yourself what would be good to do first. (Something to do with identifying cases? :)

   SOLUTION: There are two-ish base cases. When $n \leq 1$ and when $n = 2$. (You might call that three cases if you wanted to have one when $n = 0$ and one when $n = 1$, but I think that's needlessly complex.) In the recursive case, there are **three** recursive calls, the last three substantial lines of `sort_helper`.

2. Prove that `sort_helper` terminates. (How? Check out the steps above!)

   SOLUTION: This is actually tricky. A careful read shows that in each recursive call, the length of the sub-array under consideration (i.e., `hi - lo + 1`) is $\lceil \frac{2}{3} \rceil$ the length of the array under consideration

in the whole call. For $n \geq 3$, $\lceil \frac{2}{3}n \rceil < n$. Note that that is **not** true for $n = 2$. Fortunately, our many base cases ensure that we won't get stuck in an infinite recursion when $n = 2$.

And by the way, reasoning like that is how whoever wrote this code figured out what base cases they needed! (I know because I'm sitting at the same computer right now as the person writing the code.)

3. Prove that `sort` is correct. (Do those steps above! Note that you'll prove a theorem about `sort_helper` using induction, but is that really want you want to prove overall?)

   SOLUTION: The theorem is fairly simple. We want to say that "For all $n \geq 0$, `sort` called on an array of length n sorts the array."

   However, `sort` isn't recursive. It's `sort_helper` that is. So, we'll want to switch to talking about it, like: "For all $n \geq 0$, `sort_helper` called with `n = hi - lo + 1` sorts the subarray `array[lo..hi]`."

   The base cases are pretty straightforward. However, the recursive case really isn't. I at least can tell easily that I want to assume my theorem for each recursive call. So, the first recursive call sorts the first two-thirds of the (sub)array. The second recursive call sorts the latter two-thirds. The third recursive call sorts the first two-thirds.

   Why should that sort the whole array?

   Well, let's take it step-by-step. Does the first recursive call sort the array? No. The elements in the last third didn't get touched, and they might belong in the middle or even the first third. Where does this call leave us? Um.. sorted in the first two-thirds?

   OK, but the second call **does** "touch" those last third of the elements. It doesn't touch the elements in the first third, however. So, it cannot necessarily put the last "third" where they belong, since they may belong in the first third.

   Oh, but the **last** call has the chance to put the "originally last third" of the elements where they belong. Do we know for sure that all element belonging in the first two-thirds of the array are present in the first two-thirds of the array by the last call?

   If so, by the IH, we'll know they sort into exactly the location where they belong.

   Well, clearly nothing that was originally in the first two-thirds and **belonged** in the first two-thirds got moved **out** of the first two-thirds. That would have required putting it **after** something that was larger than it, which we wouldn't do by the IH.

   The stuff originally in the last third that **belongs** in the last third was sorted in the second recursive call with the larger half of what was in the first two-thirds. When that call finishes, only the largest third of items should be in the last third, and they should be in sorted order.

   Good. So, together, that should do it!

   Armed with those insights, I'll try my proof:

```
Theorem: For all n >= 0, =sort_helper= called with n = hi - lo + 1
         sorts the subarray array[lo..hi].
Base case #1: When n <= 1, the code does nothing, which establishes
  the theorem because a 0- or 1-element array cannot be out of
  sorted order. (Note that I just read the condition off the code!)

Base case #2: When n = 2, the code checks whether the two elements
  are out of order. If they're not, it does nothing. If they are, it
  swaps them. This establishes the theorem by leaving sorted order
  intact in the first case and establishing it in the second. (Again,
  I just read my condition off the code. My code had two base cases
```

```
  so my proof, whose structure follows my code, had two base cases.)

Inductive case: Consider the case when n > 2.

  Induction hypothesis: Assume the theorem holds for recursive calls
    on an array of two-thirds the size (rounded up). (I could also
    be more literal and say "for the subarrays numbers[lo, lo + two_thirds_n - 1]
    and numbers[hi - two_thirds_n + 1, hi], where two_thirds_n is etc."
    This is more readable, however; so strive for it instead!)

  Inductive step: The code computes two-thirds of the array length and
    then: (1) sorts the first two-thirds, (2) sorts the second two-thirds,
    and finally (3) sorts the first two-thirds. This establishes the
    theorem by first ensuring none of the largest third of elements is in
    the first third of the array. (By the IH, call (1) does that. Any such
    element in the first third of the array must be out of sorted order
    with some smaller element.) Then, it puts those elements precisely
    where they belong in the array and ensures that the smallest two-thirds
    of elements are not in the last third. Both are because by the IH and
    the fact that the largest third of elements are in the last two-thirds
    of the array, call (2) puts them in the last third (or they would be
    out of sorted order with some smaller element) in sorted order. Thus,
    by elimination the smallest two-thirds of elements are in the first two-
    thirds of the array, and by the IH call (3) puts them exactly in their
    sorted position, which makes the WHOLE subarray sorted.

Termination: Since for all n >= 3, ceil(2/3 n) < n, each recursive call is
    on a subarray that is at least one element shorter. In a finite number
    of steps, then, the recursive calls must reach one of the base cases.
```

That proves that `sort_helper` does what **it** is supposed to do. `sort` calls `sort_helper` with a subarray that includes every element of the original array. Thus, `sort` sorts the entire array.

4. Consider this recurrence:

$$T(n) = \begin{cases} 1 & \text{when } n = 1 \\ 3T(\lfloor 2n/3 \rfloor) + 1 & \text{otherwise} \end{cases}$$

This **almost** models the runtime of 2-3-2 sort. (It's just a tiny bit simpler.)

(a) Prove that this recurrence "terminates" in a finite number of "expansions". (That is, after substituting in for the $T(\lfloor 2n/3 \rfloor)$ term on the right a finite number of times, we reach the base case of $T(1)$.)

SOLUTION: It's really the same argument as above but easier, since we use "floor" rather than "ceiling" here.

(b) Prove that $T(n) \geq n^{\log_{3/2} 3}$ for all integers $n \geq 1$. It will be handy to know: $a^{\log_b c} = c^{\log_b a}$ and $\log_a b = -\log_a \frac{1}{b}$ (with some restrictions on the variables a, b, c that you need not worry about here).

SOLUTION: This is a recurrence. Where's the recursive call(s)? Where's the base case(s)? Where's the recursive case(s)? Well, it's really just yet another language that allows recursion. A recursive call to the function T looks like $T(\cdot)$. So, $T(\lfloor 2n/3 \rfloor)$ is the recursive call. That means the case with that call in it is a recursive case. The other case—with no recursive call—is

a base case. My theorem is just what I was given to prove. So, I'll dive in and cross my fingers that the math works out nicely :)

I'm going to use a regular font so it's easier for me to embed formulae. I'll **bold** the "section headers" in my proof:

Theorem: $T(n) \geq n^{\log_{3/2} 3}$ for all integers $n \geq 1$.

Base case: When $n = 1$, the recurrence is equal to 1, which establishes the theorem because $1 = 1^{\log_{3/2} 3} = n^{\log_{3/2} 3}$.

Inductive case: Consider the case when $n > 1$.

Induction hypothesis: Assume the theorem holds for $\lfloor 2n/3 \rfloor$.

Inductive step: The recurrence is $3T(\lfloor 2n/3 \rfloor) + 1$, which establishes the theorem because:

$$
\begin{aligned}
T(n) &= 3T(\lfloor 2n/3 \rfloor) + 1 && \text{given} \\
&\geq 3(2n/3)^{\log_{3/2} 3} && \text{by the IH} \\
&= 3(2/3)^{\log_{3/2} 3} n^{\log_{3/2} 3} && \\
&= 3 3^{\log_{3/2}(2/3)} n^{\log_{3/2} 3} && \text{by the log identity given above} \\
&= 3 3^{-\log_{3/2}(3/2)} n^{\log_{3/2} 3} && \text{by the log identity given above} \\
&= 3 3^{-1} n^{\log_{3/2} 3} && \text{by another log identity} \\
&= 3(1/3) n^{\log_{3/2} 3} && \\
&= n^{\log_{3/2} 3} &&
\end{aligned}
$$

(Yay!)

Termination: For any $n \geq 3$, $\lfloor 2n/3 \rfloor < n$; so, a finite number of steps of expanding the recurrence will reach the base case.

(c) **Just for fun:** How does that $n^{\log_{3/2} 3}$ "runtime" compare to MergeSort's or QuickSort's runtime?

SOLUTION: Figure out what $\log_{3/2} 3$ is on your calculator. It may help to know that $\log_{3/2} 3 = \frac{\log 3}{\log(3/2)}$. How does it compare to $n \lg n$ asymptotically?

3 Maximum

Let's prove this code for finding the max of an array of `int`s correct:

```
// assume array is an array of ints of length n > 0
int max_so_far = array[0];
for (int i = 1; i < n; i++)
  if (array[i] > max_so_far)
    max_so_far = array[i];
return max_so_far;
```

1. **Crucial** scratch work: Convert this to a `while` loop. (**Just for fun:** The easiest conversion technically has (at least) one problem. Find and fix it.)

   SOLUTION:

   ```
   // assume array is an array of ints of length n > 0
   int max_so_far = array[0];
   int i = 1;  // technically, this could clash with another i
   ```

```
            // how would you fix it?
while(i < n) {
  if (array[i] > max_so_far)
    max_so_far = array[i];
  i++;
}
return max_so_far;
```

You can **always** follow this pattern to convert a `for` loop to a `while`. What this means practically is that you **should** consider the initializer of a `for` before you think about the invariant for the first time, but you should **not** consider the update. (The new C++ for-each loops are even easier to reason about, even if a bit harder to convert. The rarely-used (but easier for novice programmers to understand) do-while loops can be turned into a while loop by putting one copy of the body of the loop in before the loop itself.)

Note: we're ignoring cases where the loop guard itself changes the state of the world. Loops like that are hard to read. Convert them into easier-to-read versions :)

2. State an "invariant": a theorem about the code (usually an important variable in the code) that holds each time the loop guard is tested, including the first and last times.

   SOLUTION: This is much like a "result-so-far" accumulator, and that pattern is very common. `max_so_far` holds the largest value of the array elements seen so far, which are: `array[0..(i-1)]`.

3. **Crucial** scratch work: Write out as much as you know about the values of each variable the first time the loop guard is tested (when you first reach the loop).

   SOLUTION: I know by assumption that the array has at least one element. I know `max_so_far = array[0]`. I know `i = 1`. All of that I just read off of the code and comments.

4. Prove that the invariant holds the first time the loop guard is tested. (The base case.)

   SOLUTION: First, note that this **IS** part of the proof itself. So is the statement of the invariant. (It's our theorem, at least while working with the loop itself.) Now, on to the proof:

   Base case: When the loop guard is tested for the first time, the invariant holds because `max_so_far = array[0]`, which is the only (and therefore the largest) element in `array[0..(i-1)] = array[0..(1-1)] = array[0..0]`.

5. Consider a moment when the loop guard is about to be tested, but **not** for the first time. (The recursive/inductive case.)

   (a) Assume the invariant holds the **previous** time the loop guard was tested, which **could** be the first time. (The induction hypothesis.) You can write that out explicitly here.

   SOLUTION: Note first that this whole part (including the "stem" of this problem "Consider a moment...") is all **part of your proof**. Now:

   "Assume for some value `k` of `i` larger than 1 (that is, not the base case), the invariant holds when `i = k-1` on the **previous** time the loop guard is tested."

   How did I know to focus on `i`? I looked at the update of the loop. How did I know to think about `i` being one smaller? Because the update makes `i` one larger!

   (b) Prove that the invariant still holds at the moment considered above. (How? Just as with recursion: `The loop body <does X>. This establishes the invariant because <Y>.` Notice how the loop's "update" impacts what you have to say! For example, if the update is `i++`, then `i` is getting one larger, and it was therefore one **smaller** when you made the induction hypothesis.)

   SOLUTION: By the IH, `max_so_far` is the largest element in `array[0..(i-1)]`. The loop body handles two cases: (1) When `array[i] > max_so_far`, then `array[i]` is larger than the

largest element in the previous portion of the array, and so the largest portion of `array[0..i]` is the new value of `max_so_far`, `array[i]`. (2) Otherwise, `max_so_far` is the largest value of both `array[0..(i-1)]` and `array[i]` and so **also** the largest value in `array[0..i]`. The update then increases `i`, changing this statement to `max_so_far` is the largest element in `array[0..(i-1)]`, as desired.

6. Prove that the loop terminates after a finite number of iterations. (Usually easy with `for` loops!)

   SOLUTION: The array has finite length. `i` starts at 1 and increases by 1 in each iteration of the loop. So, the loop will terminate in a finite number of steps.

7. State what you know just after the loop terminates. (Hint: it's the invariant plus that the loop guard is false.)

   SOLUTION: I know that $i \geq n$. In particular, since I know $n \geq 1$, and i started out as 1 and went up by 1 each time, $i = n$. I also know `max_so_far` is the largest element in `array[0..(i-1)]`.

8. Finish the proof of what you really wanted to prove in the first place.

   SOLUTION: Since the loop has terminated, $n \geq 1$ (by assumption), and i started out as 1 and went up by 1 each time, $i = n$. By the invariant, `max_so_far` is the largest element in `array[0..(i-1)]`. Since `i = n`, `array[0..(i-1)] = array[0..(n-1)]`, which is the whole array. So, `max_so_far` is the largest element in the array, and the code correctly finds the maximum element in the array.

For more practice: write a small piece of code to find the sum of a list of numbers and prove its correctness similarly.

4 Fibonacci Madness

The Fibonacci sequence is defined by this recurrence:

$$Fib(n) = \begin{cases} 1 & \text{when } n = 1 \text{ or } n = 2 \\ Fib(n-1) + Fib(n-2) & \text{otherwise} \end{cases}$$

We can compute this recursively:

```
// return Fib(n) per the definition above
// precondition: n >= 1
int fib(int n) {
  if (n == 1 || n == 2) return 1;
  else                  return fib(n-1) + fib(n-2);
}
```

You could prove this correct by induction. (You'd find the proof so simple it would feel like you were missing something.) Here's a loop you've (roughly) seen before that's more efficient because it avoids recomputation:

```
int fibn = 1, fibn_minus_1 = 1;
for (int i = 2; i < n; i++) {
  int temp = fibn;
  fibn = fibn + fibn_minus_1;
  fibn_minus_1 = temp;
}
return fibn;
```

As an exercise for yourself, prove this correct by induction. (The variable names should suggest invariants. I'd handle the case where $n = 1$ as a totally separate special case and then assume $n \geq 2$.)

Of course, this can be written recursively as well. So, it's not really iteration that makes for more efficient code but the twin insights of computing the Fib numbers from the "bottom up" and only keeping the last two around.

Hey... maybe we can make the iterative code as bad as awesomely bad as the recursive code:

```
// computes Fib(n) per the standard definition starting w/n=1 and n=2
// precondition: n >= 1
int fib(int n) {
  std::stack<int> s;
  int result = 0;
  s.push(n);
  while (!s.empty()) {
    int i = s.top();
    s.pop();
    if (i == 1 || i == 2)
      result++;
    else {
      s.push(i-1);
      s.push(i-2);
    }
  }
  return result;
}
```

1. State a good invariant for this loop. Hint: your invariant should be about the key variables: `result` and (the contents of) the stack `s`. Are the contents of `s` arguments to `fib` or results of `fib`?

   SOLUTION: This is **VERY** hard. Let's take it piece-by-piece. `result`—at least eventually—should be the result of a Fib computation because we return it at the end. However, the elements on the stack are arguments to Fib. We can tell (or at least guess!) because we pop them off the stack and then push on their "minus 1" and "minus 2" pieces, which is how Fib arguments work.

   So, we want to think about `result` alone and Fib of each element of the stack. Something like: `result ... Fib(i) ... for each i on the stack`. That had better start out coming out to Fib(n). So, when `result = 0` (as it clearly does the first time we test the loop guard) and the stack has only `n` on it, we want that to equal `Fib(n)`. Furthermore, when the stack is empty (as it is when we exit the loop), we need `result` to be equal to Fib(n). So, maybe we can add these?

   Something like: `Fib(n) = result + (... Fib(i) for each i on the stack)`.

   Later, we'll put `i-1` and `i-2` onto the stack to replace `i`, and we know by definition that `Fib(i) = Fib(i-1) + Fib(i-2)`, at least for large enough `i`. So adding the elements of the stack also sound good. Thus, my guess at the invariant is:

   `Fib(n) = result + the sum of Fib(i) for each i on the stack`.

2. **Crucial** scratch work: Write out as much as you know about the values of each variable the first time the loop guard is tested (when you first reach the loop).

   SOLUTION: We did some of this just to figure the invariant. Here's what we saw: `s` contains only `n`. `result = 0`.

3. Prove that the invariant holds the first time the loop guard is tested. (The base case.)

SOLUTION: We set this up to work.

Base case: The first time the loop guard is tested, the stack `s` contains only `n`, and `result = 0`. Therefore: `result + the sum of Fib(i) for each i on the stack` is `0 + Fib(n) = Fib(n)` as desired.

4. Consider a moment when the loop guard is about to be tested, but **not** for the first time. (The recursive/inductive case.)

   (a) Assume the invariant holds the **previous** time the loop guard was tested, which **could** be the first time. (The induction hypothesis.) You can write that out explicitly here.

   SOLUTION: Assume the invariant holds when testing the loop guard just before an iteration of the loop that will not leave the stack empty.

   (b) Prove that the invariant still holds at the moment considered above. (How? Just as with recursion: `The loop body <does X>. This establishes the invariant because <Y>.` Note: if your code has an `if`, your proof is likely to proceed by cases, not just here but **always**. Your proof follows the structure of the code!)

   SOLUTION: By the IH, just before the loop body executes, `result + the sum of Fib(i) for each i on the stack = Fib(n)`. The loop body does one of two things:

   (1) If the top element of the stack was 1 or 2, increment result and remove that top element from the stack. Notice that `Fib(1) = Fib(2) = 1` by definition. So, if `result + the sum of Fib(i) for each i on the stack` was equal to `Fib(n)` before, now we remove either 1 or 2 from the stack, which reduces the total by 1 and add 1 to result, which increases the total by 1. These two actions balance, preserving the invariant.

   (2) If the top element of the stack was 3 or more, the body pops that element `i` off, leaves `result` unchanged, and pushes on `i-1` and `i-2`. So, `result + the sum of Fib(i) for each i on the stack` changes only be removing `Fib(i)` for some `i > 2` and adding `Fib(i-1)` and `Fib(i-2)`, but by definition `Fib(i) = Fib(i-1) + Fib(i-2)` for such a value of `i`. Thus, the sum is unchanged and the invariant is preserved.

5. Prove that the loop terminates after a finite number of iterations. (**Not** so easy here. Just give it a try.)

   SOLUTION: This is perhaps even trickier than the invariant itself. Alan Hu gives a beautifully elegant proof that involves looking at the lexicographic (alphabetic, roughly) order of sorted stacks, which decreases at every step. I'm going to give a much more hand-wavy argument: At each step, we remove a positive integer from the stack, replacing it with a finite number of smaller numbers (none, in the case where the removed number is 1 or 2) and never adding anything less than 1. A finite number of such steps will result in all remaining numbers being 1 or 2, and then an empty stack. Thus, the loop will terminate in a finite number of steps.

6. State what you know just after the loop terminates. (Hint: it's the invariant plus that the loop guard is false.)

   SOLUTION: The stack is empty (because the loop terminated) and `result + the sum of Fib(i) for each i on the stack = Fib(n)` because we proved the invariant holds throughout.

7. Finish the proof of what you really wanted to prove in the first place.

   SOLUTION: The stack is empty (because the loop terminated) and `result + the sum of Fib(i) for each i on the stack = Fib(n)` because we proved the invariant holds throughout. But, if the stack is empty, the sum of `Fib(i)` for each `i` on the stack is just 0. So, `result + 0 = Fib(n)`, and `result = Fib(n)`. Thus, the function correctly computes `Fib(n)`.

(Of course, if you try this function for large values of `n`, you'll find it does **not** correctly compute `Fib` because of the limitations of the `int` type and that the computation is painfully slow because we intentionally replicated the horrid behaviour of the naïve recursive solution. But, you could change this to use arbitrary precision integers instead of `int`, and we signed on for the bad performance from the start with our design!)