CPSC 221 Notes: Induction, Invariants, and Program Correctness

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.

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.)

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.)

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).

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

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

   (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.

   (e) Fill in this proof structure using your notes above:

```
Theorem: <theorem in terms of the arguments or a property of them>
Base case(s): When <condition that holds in the base case>, the code
   <does X>, which establishes the theorem because <Y>.

Inductive case(s): Consider the case when <the arguments have an
    arbitrary value that matches this inductive case; typically,
    this just means "not the base case">

  Induction hypothesis: Assume the theorem holds for <the arguments'
    values in each recursive call in this inductive case>.

  Inductive step: The code <does X>, which establishes the theorem
    because <Y>. (USE the IH as soon as you mention the recursive call.)

Termination: <Show that a finite number of steps through any
    sequence of recursive calls will reach some 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.)

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

   Here's a copy of the proof structure, in case it's handy.

```
Theorem: <theorem in terms of the arguments or a property of them>
Base case(s): When <condition that holds in the base case>, the code
   <does X>, which establishes the theorem because <Y>.

Inductive case(s): Consider the case when <the arguments have an
    arbitrary value that matches this inductive case; typically,
    this just means "not the base case">

  Induction hypothesis: Assume the theorem holds for <the arguments'
    values in each recursive call in this inductive case>.

  Inductive step: The code <does X>, which establishes the theorem
    because <Y>. (USE the IH as soon as you mention the recursive call.)

Termination: <Show that a finite number of steps through any
    sequence of recursive calls will reach some 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.)

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? :)

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

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?)

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 $T(1)$.)

(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).

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

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.)

   A `while` loop is easy to convert to recursion. So, we can use the same style of approach on the loop as on recursive code.

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.

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).

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

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.

   (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.)

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

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

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

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`?

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).

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

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.

   (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!)

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

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

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