Overview of Sections 2.2, and 3.1-3.3

March 5, 2021

Abstract

Your test will resemble the problems from your homework assignments, and problems from previous tests I've given. You will probably have 4 equally weighted questions or so (one every fifteen minutes).

1 Section 2.2

Induction is a proof technique which is useful for demonstrating a property for an infinite ladder of propositions (think of our property as being indexed by n, as in $P(n)$. Induction begins with a base case (or an anchor) and then proceeds via an inductive case (often $P(k) \rightarrow P(k+1)$).

Note that $P(n)$ is a proposition – almost never (that I can think of) a number, or something that one would associate with an equal sign. For exammple,

$$P(n) : n! > 2^n$$

And we could prove that $P(n)\ \forall n \geq 4$.

There are two different (but equivalent) principles of induction, the first and second. The second appears to assume more than the first: the inductive hypothesis in the second principle is that the property is true for all cases up to and including the k^{th} case.

The second principle is useful when you need more ammunition than the first principle provides. So if you need to assume more of the $P(n)$ than simply the $P(k)^{th}$, you'll be using the 2nd principle. An example was the

Both principles of induction are equivalent to the principle of well-ordering, which asserts that every non-empty set of positive integers has a smallest element.

Sometimes more base cases than 1 are needed: you'll need enough base cases, sometimes closely related to the order of the recurrence (see next sections). So if you have a second order recurrence relation, you may need two base cases.

At some point in your proof you should **invoke the inductive hypothesis** (e.g. $P(k)$, in the first principle proofs). You've assumed it, presumably because doing so is going to be useful! Point out where you are using it.

At the end, you should restate what you've proved, and how you've proved it. Frequently it looks like

$$\therefore P(n) \forall n \in \mathbb{N}$$

2 3.1

Recursion in section 3.1 looks very much like induction: the idea is that we have a base case (or cases), and from there we generate additional cases. Unlike induction, the set of things we generate may not be easily indexed to the integers. For example,

- the palindromes on binary strings, or
- the construction of wffs using the logical connectives.

All kinds of things are defined recursively. In the course of this and the next few sections, you saw the Factorials, Fibonacci numbers, the Euclidean algorithm, and BubbleSort, and Binary Search defined recursively.

The Factorials illustrated an interesting situation: by defining them recursively we avoided an awkward problem with how to think about 0!. It's simply 1 – you needn't be bothered about thinking of it as the product of the first 0 counting numbers!?????

The Fibonacci numbers illustrated how recursion may be a horrible way to compute something, even when it's a nice way to define something. We were able to get around the debilitating computation with a different recursive scheme, however. So it may be that the scheme we've chosen to implement is simply bad.

3 3.2

Frequently a "guess, check, verify" strategy is helpful to deduce the form of the terms in a recurrence relation. You may be able to deduce a form for the closed form solution, which you can then prove with induction.

In this section we see how to solve one particular recurrence relation, the linear, first-order, constant-coefficient recurrence relation, to obtain what we hope is a closed-form solution which we can use to calculate more efficiently.

Once we have this formula, we needn't ever solve another linear, first-order, constant-coefficient recurrence relation from scratch: we can just invoke the formula. This is our quest, the holy grail!

We were also able to generalize from that to a formula for the "divide and conquer" algorithms, that cut a problem in half (say). And we have a formula for the closed form solutions of those, too.

4 3.3

In the analysis of algorithms we are interested in efficiency, and will count operations in order to compare competing algorithms. We can sometimes count operations directly, but may resort to recursion to count. We're interested in the worst-case scenario.

A different variety of recurrence relation occurs in the analysis of algorithms, when we consider "divide and conquer" algorithms (such as *BinarySearch*).

We also saw that, even if we can't count exactly, we may be able to provide bounds on the worst case behaviour. We looked at the Euclidean algorithm for the computation of the greatest common divisor ("gcd") of two numbers this way, and were able to come up with a crude bound; then refine it considerably, using what we knew about the Fibonacci numbers.