Handout 1: Good Writing, Induction

One of the objectives of this course is to practice good writing in the domain of algorithms, computing, and mathematics. This skill is highly valued in both industry and academia. Thus, a substantial fraction of the points on each assignment will be allocated for good writing. Here are the elements we'll be looking for:

Give an Overview: When a solution requires a few paragraphs of text, please start with a one sentence overview to give the reader a sense of what follows. The summary might be as simple as "We'll describe a greedy algorithm and then prove its correctness. The key idea is that selecting an edge of smallest weight that does not introduce a cycle preserves the greedy-choice property."

Begin Paragraphs with the Key Points: When the solution is a bit long, break your prose up into relatively short paragraphs, just as you would break a long program up into constituent functions. And, just as you would provide short documentation for each function, start each paragraph with a sentence explaining what you're about to do (e.g., "Now we analyze the running time of our function," "Here we prove correctness of the main subroutine.").

Use Intuitive Notation: Just as you should use good variable names in your programs, use intuitive notation in your algorithm description and proof. While some notation is standard (e.g., G and H are common names for graphs, V and E are common names for the vertex and edge sets in graphs, and u, v, and w are widely-used generic names for vertices in graphs), in many cases you'll need to invent your own notation and variable names. It makes sense to choose notation and names that are easy for the reader to understand and remember. For example, in an argument about cats and mice, using the variable c for a cat and the the variable m for a mouse is much clearer than using x and y. For the value of a variable v after the ith iteration of a loop, v_i can be a good choice.

Read (and Re-Read) What You Wrote: Please read what you wrote and make an editorial pass (or two, or three!) before you submit your write-up. Check for unsound logic, sloppy organization, malformed mathematical expressions, spelling and grammatical errors, etc. Almost certainly, you'll find things you can edit to make it clearer, more concise, and better organized.

When You Know That Something's Wrong: If you know that something is not quite right in your solution (e.g., a logical gap, an unhandled case), write a caveat at the beginning of your solution that explains what you understand to be less-than-completely correct. This helps us give you good feedback and demonstrates that you understand that there is a problem. *The grading rubric will always give some additional points for thoughtful caveats.* Here's an example: "In the proof below, something isn't quite right in my induction step. The problem arises when ...."

Adapted from a handout by Ran Libeskind-Hadas.

1 Inductive Proofs

1.1 A First Example

Let's begin with a simple arithmetic result evidently discovered by Gauss when he was in primary school.

Theorem 1.1. *For all integers $n \geq 1$,*

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2}.$$

We'll begin with a correct and clearly written proof of this fact.

Proof. The proof is by induction on n.

Base case: Let $n = 1$. In this case, the left-hand side is $\sum_{i=1}^{1} i = 1$ and the right-hand side is $\frac{1(1+1)}{2} = 1$ as well, thus the claim holds for $n = 1$.

Inductive Hypothesis: Assume that the statement is true for $n = k$, for some positive integer k. In other words, assume that

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}.$$

Inductive Step: We must show that $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$. By the inductive hypothesis,

$$\sum_{i=1}^{k} i = \frac{k(k+1)}{2}.$$

Therefore, adding $k+1$ to both sides, we get

$$\begin{aligned}\left(\sum_{i=1}^{k} i\right) + (k+1) &= \frac{k(k+1)}{2} + (k+1) \\ \implies \sum_{i=1}^{k+1} i &= \frac{k(k+1)}{2} + \frac{2(k+1)}{2} = \frac{(k+2)(k+1)}{2} = \frac{(k+1)(k+2)}{2},\end{aligned}$$

as desired. □

Note that in this proof, we introduced the variable k in the inductive hypothesis. This is a stylistic choice, and is not really necessary for the reader's understanding, though it does contribute to the formal rigor of the proof. It is also OK to state the inductive hypothesis as "Assume that the statement is true for some n. In other words, $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$." Then, in the inductive step they would say, "We need to show that the statement holds for $n+1$, i.e., $\sum_{i=1}^{n+1} i = \frac{(n+1)(n+2)}{2}$." Either way of writing the proof is entirely acceptable, if it is presented clearly.

Next, let's look at a "bad" proof of this same theorem. After the proof, we'll explain what's "bad" about it.

Bad "Proof". The proof is by induction on n.

Base case and inductive hypothesis: [identical to above]

Inductive Step: We must show that $\sum_{i=1}^{k+1} i = \frac{(k+1)(k+2)}{2}$.

$$\begin{aligned}
\sum_{i=1}^{k+1} i &\stackrel{?}{=} \frac{(k+1)(k+2)}{2} \\
(\sum_{i=1}^{k} i) + (k+1) &\stackrel{?}{=} \frac{(k+1)(k+2)}{2} \\
\frac{k(k+1)}{2} + (k+1) &\stackrel{?}{=} \frac{(k+1)(k+2)}{2} \quad \text{by the Inductive Hypothesis} \\
\frac{k(k+1)}{2} + \frac{2(k+1)}{2} &\stackrel{?}{=} \frac{(k+1)(k+2)}{2} \\
\frac{(k+2)(k+1)}{2} &\stackrel{?}{=} \frac{(k+1)(k+2)}{2}
\end{aligned}$$

Since the left-hand side and the right-hand side of the last expression are equal, the proof is complete. □

What's wrong with this proof? First of all, there is no mathematical symbol $\stackrel{?}{=}$. Using that notation is analogous to writing an ungrammatical or nonsensical sentence. Second, the proof does not ever say that all the left-hand sides are supposed to be equal; though an attentive reader might figure this out, it is a key part of the argument and should be stated explicitly. Third and mostly importantly, though, the proof goes "backwards": it starts with what we want to show, and deduces something true. This is risky, since it relies on an implicit "if and only if" between each line above. Although that happens to be work out here, imagine if we had done an algebraic manipulation that was not "if and only if." For example, if we had multiplied both sides of one of the lines by zero, we would have deduced in the end that $0 = 0$, which is true. However, we cannot conclude from this that the original statement was true.

It's fine to use this kind of reasoning in the privacy of your own notes as you are trying out ideas and developing a potential proof. However, when you write up a proof for others to read, the exposition should be clear, and the reasoning should be sound. This second proof doesn't cut it.

1.2 Induction for Reasoning about Algorithms

Not all inductive proofs will be arithmetic or algebraic in nature. In fact, most proofs in this course will be about algorithms, including ones that work on data structures like arrays or graphs or other mathematical objects. Let's see some examples of theorems and proofs about algorithms.

First recall the notion of a binary tree, which has a *recursive* (or inductive) definition, and is therefore a great fit for inductive proofs. Henceforth, all trees will be binary, so for brevity we will usually drop that descriptor. A particular (binary) tree is one of the two following possibilities:

1. The "nil" or "empty" tree, which has no content. This is analogous to an empty (zero-length) string, an empty list, a graph with no vertices, etc.

2. A *node* called the "root," together with a left and a right tree, called the "child" subtrees.

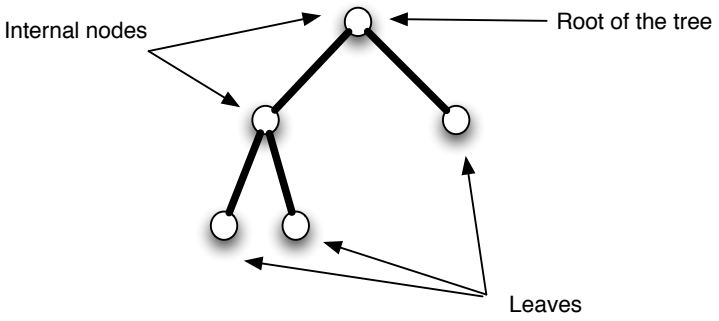

Figure 1: A binary tree.

Note that by definition, any child subtree is itself a tree, so it can be empty, or not. A non-empty tree whose two children are both empty trees is called a *leaf*; otherwise, its root is called an *internal* node. (We won't use these terms below, though.) For example, a binary tree is shown in Figure 1.

Let's write an algorithm `numNodes` that counts the number of nodes in a given tree, and write an inductive proof that it is correct. Because trees are recursive/inductive data structures, it is most natural to write the algorithm recursively as well. Remember that every recursive algorithm should have a *base case* and a *recursive case*. It is important to understand that these are *not the same* as the base case and inductive case in an inductive proof—the algorithm actually performs steps, while the proof analyzes what those steps do—but they are *closely related*: the proof's base and inductive cases will typically be "about" the algorithm's base and recursive cases, respectively.

`function numNodes(tree` T`):`

1. If T is the empty tree, return 0.
2. Otherwise, return $1 +$ `numNodes`(T.left) + `numNodes`(T.right).

Observe that we have written the algorithm in a human-friendly style, as "pseudocode." It is a mixture of English and math notation (mostly the former). But the meaning of each line should be clear and unambiguous to a reader from the intended audience (in this case, EECS 376 students), and such a reader should be able to straightforwardly and correctly translate this into real code in their favorite programming language.

We now state and prove a theorem about this algorithm.

Theorem 1.2. *`numNodes` returns the number of nodes in any input tree.*

We first give a "bad" proof of this theorem, and describe what is bad about it.

Bad "Proof". The proof is by induction on the number of nodes n in the tree (in this case, starting from zero).

Base case: When $n = 0$, the tree is empty, so the condition in the first line of the algorithm is met, and the algorithm returns zero, as desired.

Inductive Hypothesis: Assume that `numNodes`, when given any tree with n nodes, outputs the number of nodes in that tree.

Inductive Step: We need to show than when `numNodes` is given any tree T with $n+1$ nodes, it outputs $n+1$. So, take an arbitrary tree T' with n nodes. Now, build a tree T with a root node, a left-child subtree of T.left $= T'$, and an empty right-child subtree T.right. By construction, this tree T has $n+1$ nodes: the root, plus the n nodes in T', plus none in the right subtree.

Now observe that when `numNodes`(T) is run, the condition in the first line is not met (because T is not empty), so it proceeds to the second line. By the inductive hypothesis, `numNodes`(T.left) returns n, and by the base case, `numNodes`(T.right) returns zero, so `numNodes`(T) returns $1+n+0 = n+1$, as desired. □

What's wrong with the above proof? The problem here is that in the inductive step, we must show that `numNodes` returns $n+1$ on **any** binary tree T that has $n+1$ nodes. But all we did was show that `numNodes` returns $n+1$ on **some specific** tree T with $n+1$ nodes—namely, the one whose left-child subtree has n nodes, and whose right-child subtree has zero nodes. Not every tree with $n+1$ nodes has this "shape," so we did not prove what was required. **This is a common error called the "induction pitfall." Watch out for it!**

How to fix the proof. The key is that we must begin the inductive step by considering some *arbitrary* tree T with $n+1$ nodes, without assuming anything about its "shape." We could then hope to appeal to the inductive hypothesis, which says that `numNodes` returns n on *any* tree with n nodes, and try to apply that hypothesis to the algorithm's recursive calls. However, a moment's thought reveals a problem: what if T.left and/or T.right don't have exactly n nodes? In fact, they can't *both* have exactly n nodes (unless $n = 0$), because T itself has $n+1$ nodes! So, the inductive hypothesis does not say anything about how `numNodes` behaves on T.left and/or T.right.

The solution is to proceed by so-called *strong* induction, which is a minor variation on ordinary induction. The difference is that in strong induction, the inductive hypothesis is that the statement in question holds for *all* positive (or non-negative) integers up to n, not just for n itself. In the present case, the (strong) inductive hypothesis will say that `numNodes` is correct on any tree having *at most* n nodes, and hence both recursive calls return correct answers (regardless of the sizes of T.left and T.right). Strong induction is often needed when formally proving correctness of a recursive algorithm, because the algorithm might call itself recursively on smaller inputs of various sizes.

Proof of Theorem 1.2. The proof is by strong induction on the number of leaves n in the tree.

Base case: [same as above]

(Strong) Inductive Hypothesis: Assume that `numNodes`, when given any tree having *at most* n nodes, outputs the number of nodes in that tree.

Inductive Step: We must show that `numNodes`, when given *any* tree having $n+1$ nodes, outputs $n+1$. (Alternatively, we could show that `numNodes` is correct when given any tree having *at most* $n+1$ nodes, but since the case of at most n nodes is already covered by the inductive hypothesis, it suffices and is simpler to limit our attention to the case of exactly $n+1$ nodes.)

So, let T be an arbitrary tree with $n+1$ nodes. Since T is not empty, it has a root node, and left and right child subtrees. These two subtrees have a total of n nodes between them, because T's root is the one and only node in T that is not in either subtree. Hence, both T.left and T.right have at most n nodes each.

When `numNodes`(T) is run, the condition in the first line is not met, so it proceeds to the second line, which returns $1+$ `numNodes`(T.left) + `numNodes`(T.right). By the (strong) inductive hypothesis, `numNodes`(T.left) returns the number of nodes in T.left, and similarly for T.right; as noted above, the sum of these is n. Therefore, the output of `numNodes`(T) is $n+1$, as desired. □

Let's make a few important observations about this proof. First, we began with an *arbitrary* tree T with $n+1$ nodes, without assuming anything about its "shape," since we need to show that the claim holds for *every* such tree. By contrast, in the previous proof, there was no guarantee that we had considered every possible tree with $n+1$ nodes (and indeed, we hadn't). So, the second proof is better already!

Next, we proved some facts about T's left and right subtrees. Then, we stepped through the execution of `numNodes`(T), observing that the "otherwise" clause (on the second line) is triggered, which returns a certain value involving recursive calls. In order to conclude anything about these calls via the (strong) inductive hypothesis, we needed to ensure that they are made on trees having at most n nodes. We achieved this by observing that T's root is one of its nodes, leaving a total of only n nodes between its two child subtrees. In particular, each child subtree has at most n nodes—here it is important that a (sub)tree cannot have a negative number of nodes!—so the recursive calls are indeed covered by the (strong) inductive hypothesis, and hence their outputs are correct.

1.3 Final Thoughts

The above is a very detailed and rigorous proof of the correctness of a very simple algorithm. Indeed, you might have initially thought (and/or might still think) that correctness is "obvious," and does not require a proof! We don't really disagree. But as a counterpoint, consider that in our first-draft pseudocode, `numNodes`(T) returned just "`numNodes`(T.left) + `numNodes`(T.right)," without the "$1+$". It was not until we attempted to write the proof that we noticed the omission! (Testing would also have revealed the bug, but it is hard to automate testing of pseudocode.)

A main message of what we've done here is that thinking about how one *would* rigorously prove correctness can be very valuable in *organizing your approach* and *designing an algorithm in the first place*. For very simple algorithms like the one above, we do not expect or want you to give full correctness proofs. But you will soon encounter more complex problems, and be asked to devise and analyze non-obvious algorithms for them. Your pseudocode and reasoning should be presented in a way that conveys the key problem-specific ideas, and reasonably convinces us (and your fellow students) that all the details of such a proof *could* be given, following a standard approach like induction. This means, for example, arguing that all recursive calls are made on smaller inputs, and that the algorithm combines all the (assumed correct) answers to those calls to produce a correct output for the original input.