As a general rule in this course, you should imagine writing proofs for a "skeptical peer." That is, somebody who knows roughly the same amount of background material as you, but who needs to be convinced that what you say is true. Unluckily, this skeptic will typically do their best to poke holes in your argument, so you need to make sure that your logic is sound, precise, and clear enough to defend against such probes. Here are some accumulated suggestions to consider when writing proofs.

1 General

- One style point regarding induction proofs (and which can arise in equation-style proofs in general): be careful that you do not accidentally assume the claim you wish to show! One way to ensure this, when the desired property is an equation, is to start with one side of the equation and manipulate it until you reach the other side.

  As an explicit example of such an error, consider the "phony proof:"

  > We show that $1 + 2 + \cdots + n = 0$ for all n. Base case: $n = 0$, and the claim is trivial. For $n = k + 1$:
  >
  > $$\begin{aligned} 1 + 2 + \cdots + k + 1 &= 0 \\ 1 + 2 + \cdots + k + 1 + (1 + 2 + \cdots + k - 1) &= (1 + 2 + \cdots + k - 1) \\ 1 + ((1 + 1) + (1 + 2) + \cdots + (1 + k)) + (1 + 2 + \cdots + k - 1) &= (1 + 2 + \cdots + k - 1) \\ 1 + (1 + 2 + \cdots + k) + k + (1 + 2 + \cdots + k - 1) &= (1 + 2 + \cdots + k - 1) \\ (0 + 1) + (0 + 1 + 2 + \cdots + k) + (0 + 1 + 2 + \cdots + k) &= (0 + 1 + 2 + \cdots + k - 1) \end{aligned}$$
  >
  > Each of the grouped terms is a sum of integers from 0 up to a value at most k, so the induction hypothesis applies and we can substitute:
  >
  > $$0 + 0 + 0 = 0$$
  >
  > Since $0 = 0$ is true, our initial hypothesis holds, so the induction step is valid!

  The primary logical error here is the fact that the conclusion is true does not mean that the assumptions are (remember, false implies anything). In many cases, the manipulations we make happen to be "if and only if" statements, which makes this less of an issue: one could simply reverse the entire argument. But there are situations where this is not the case (and where this will not be as obvious as in the above example), so it's generally cleaner to start with what we know (or are assuming) to be true and work towards your "want to show."

- For induction proofs, it is also important to start from the "$k+1$ case" and work down. As an example of how this might go wrong, consider the claim "Any graph with n vertices and $n - 1$ edges that has no isolated vertices is a tree." This is not true (try to think of a counterexample!). But we may try to prove this as follows:

  Base Case: $n = 1$, and the only graph with 1 vertex and 0 edges is a single isolated vertex, so the claim vacuously holds. For $n = 2$, there is only one graph that has 2 vertices and 1 edge, and it is a tree.

  Now for the Induction Step, we assume that the claim holds for $n = k$ and try to show that it holds for $n = k + 1$.

  Let G be a graph with k vertices and $k - 1$ edges. To obtain a graph with $k + 1$ vertices, add a new vertex v to G to obtain G'. If v is an isolated vertex, then G' doesn't satisfy the premise, so we are done. Otherwise, v must have an edge to another vertex in G. Since G is a tree and we only added a single edge to v, G' is still connected and we could not have introduced a new cycle. Thus, G' is a tree, as desired.

  The issue here is that we haven't actually proven the claim for arbitrary graphs with $k + 1$ vertices, k edges, and no isolated vertices. You should be able to give an explicit example of such a graph for which there is no way to remove a vertex to obtain a size-k version of a graph satisfying the premises. If we had instead started with $n = k + 1$ and removed a vertex, rather than adding one to a graph of size k, this issue would have been more apparent.

2 Automata/TMs

- Make sure to *argue* correctness of your constructions! This typically entails a bidirectional argument to show that the language of an automaton/TM M satisfies $L(M) = A$: first, you show that every string in the language A is actually accepted by your automaton/TM. Then, you show that every string which is accepted is one that should be, i.e., it is in A.
- Remember to test edge cases, e.g., the empty string.
- Ensure that your constructions satisfy the necessary definitions. For example, the transition function of a DFA needs to be a function $\delta : Q \times \Sigma \to Q$.
- When arguing equivalence of two models of computation (e.g. DFA's and NFA's), you need to show *both* directions (any language computable by a DFA is computable by an NFA, and vice versa).
- When using the pumping lemma, your argument must work for *any* choice of x, y, z that satisfies $xyz = s$, $|xy| \leq p$, $y \neq \varepsilon$. Just showing that it works for one such choice is insufficient, because it corresponds to a "for all" quantifier. In the game sense, the latter condition would be equivalent to the opponent having *some* move where you can guarantee a win, while the former would be equivalent to the opponent *only* having moves where you can guarantee a win. A way to streamline this approach is to limit their choices via the structure of s, that is, choosing a string with very little "interesting information" in the first p characters, such as 0^p1 – then the opponent will only be allowed to choose how many zeroes each of x and y can contain, restricting their flexibility in string construction.
- When using the Myhill-Nerode theorem to prove a language nonregular, you need to show that there are an infinite number of equivalence classes. This requires (1) providing an infinite set of strings S and (2) showing that every pair of strings in S is distinguishable (i.e., S is pairwise distinguishable). This is *not* equivalent to S having infinitely many pairs of distinguishable strings. Any equivalence relation $\equiv_L$ with more than 1 equivalence class (so any L except for $\emptyset$ and $\Sigma^*\}$) will have infinitely many pairs of distinguishable strings, but some of these have only finitely many equivalence classes.

3 Reductions

- Be careful about the direction of your reduction. To show that A is "hard" (i.e., undecidable or unrecognizable), we start with a known hard problem B and reduce $B \leq A$ or $B \leq_m A$ (showing that A is "harder" than B). This entails assuming that a solution for A exists and showing how we can use it to produce a solution for B. This yields a contradiction because we already know B has no easy solution.
- Make sure that your reduction is of the correct type! For example, if you are reducing to $\mathsf{HALT_{TM}}$, a language whose elements are all of the form $\langle M, w\rangle$, you should make sure that the output of your reduction is also a pair of a machine description and a string.
- Remember that a reduction must show how to construct a new Turing machine to solve some problem. In particular, when writing the description, we can't include conditionals of the form "If M accepts w, then ... else, ..." It is specifically the "else" condition that is an issue here, because we can't implement the check "Does M not accept w?" in a Turing machine — this language is unrecognizable! The first "if" clause ends up being okay, because we can run M on w and then continue to the "then" portion *after* M accepts. In this case, the simulation terminates by assumption so it is okay to perform operations "afterwards." But since we can't recognize M *not accepting* w, this clause cannot be part of a Turing machine description. To avoid this, I recommend generally avoiding "else" clauses, and making sure that "if" conditionals are recognizable behaviors, such as a TM accepting a string, or halting on the string.
- Along these lines, students sometimes write reductions in the form:"Make M' such that if M accepts w, then M' accepts Σ^* and if M rejects w, then M' accepts no strings." However, this isn't actually a Turing machine description, because it doesn't describe how this behavior can be *implemented*. In

terms of software development, this sentence is analogous to a specification that describes the desired behavior of an M', whereas a complete proof needs to specify the actual implementation (the analogue of code).

- Note that Turing machines don't get to choose their input strings, so the behavior "accept a particular string x" is less correct than "check if the input is x and if so, accept." The latter is a behavior that can be encoded into a Turing machine, while the former can't be directly implemented (like above, it's more of a specification).

- Generally speaking, you should make sure any variables you use are bound. Some proposed reductions describe a new Turing machine M' which takes input x, and then give $\langle M', x\rangle$ as the output of the reduction. But outside of the description of M', the variable x is unbound – it doesn't have a value! This is analogous to writing a function with a local variable x and then trying to call x outside of the function, where it hasn't been assigned. One way to mitigate this risk is to use fixed constant strings (such as ε or 01 or 0^n) as the output of reductions, rather than arbitrary x.

4 Time/Space Complexity

- Be sure to argue that your algorithms actually run in the appropriate amount of time and space. This requires a step further than stating that "this procedure runs in polynomial time." In CSC-341, I will require that such claims are supported by an explicit big-O bound obtained by analyzing the steps of the algorithm.

- Time/space bounds are properties of the entire computation and not just the output. This means that for runtime/space usage analysis, you really do have to argue about many resources each step of the procedure takes. For example, consider an algorithm which takes input $\langle G, k\rangle$, a graph and integer, and searches every size-k subgraph for a clique, which it returns if one is found. The output has size $O(|V|)$, which is linear in the input size. However, the runtime of this algorithm is $O(|V|^k)$, which is exponential in the input size.

- Be specific about the contents of your certificate! You will find it difficult to give precise runtime/space bounds if you do not.

- Space usage rule of thumb: An integer x can be written using $O(\log |x|)$ space, while any other type of object F takes $O(|F|)$ space.