Reasoning and writing about algorithms: some tips

The suggestions below address common issues arising in student homework submissions. Absorbing these lessons should be worthwhile in this class and beyond.

Outline your writeup.

Here is a widely-applicable outline for the HW problems:

1. We define the following algorithm A: [blah blah]

2. We claim that on every input $x$, $A(x)$ halts with the desired output (as specified in the problem statement). Proof: [blah blah]

3. We claim that on any input $x$ with input size-parameter $n$, the runtime of $A$ is at most $F(n)$ [for some explicit function $F$]. Proof: [blah blah]

You may want to write a similar outline for these individual parts. For example, if you are writing a divide-and-conquer algorithm, you will probably want an algorithm with: a base case; a way of taking a problem instance and generating recursive calls to related, smaller instances; and a way of “combining” these smaller problems’ solutions to obtain a solution to the main problem. It may help to draw an outline of these parts, and then fill in the details of what should happen in each part. Similarly, the proof of correctness and efficiency will use induction and have a predictable structure, and it may help to diagram these.

The 3-part solution outline listed above might sound obvious, but student submissions often deviate from this format—and usually are worse as a result, for one of several reasons:
• *omission:* forgetting to provide a clear proof of correctness or efficiency;

• *redundancy:* writing down more thoughts and observations than are needed to solve the problem, e.g. explaining things once vaguely and then again with more details;

• *confusion:* the submission is poorly organized and it’s not clear what is being proved at any given point in the argument.

An important, closely-related point:

**Separate algorithm from analysis.**

This is so important and comes up often. It is almost always best to first state your algorithm in clear pseudocode, then start making and proving assertions about what it does. Students who fail to follow this approach may include some partial analysis observations while motivating or specifying the algorithm, and mistakenly believe this is sufficient to show correctness.

The only common exception to the separation rule is that you might want to first define a *subroutine*, prove that it correctly solves some sub-task, and then use that subroutine to specify a larger algorithm to solve your main problem. For example, when we use efficient data structures, we like to specify and analyze their implementations first, and then we can use them freely in various applications.

As with ordinary programming, you have the right to include *comments* in your pseudocode, indicating what it is supposed to be achieving at a given point in the code. But these are just helpful hints to the reader that may aid in verifying the actual analysis, which comes after.

**Separate program variables from the quantities we wish to compute.**

This is part of the previous point. Here is a simple example: suppose that on input an integer $N > 2$, our goal is to compute the $N^{th}$ Fibonacci number $f_N$.

**Algorithm:** Define the algorithm $A$ as follows:

*Input:* integer $N > 2$.

Create an integer array $a[1 : N]$ of length $N$;
Set $a[1] \leftarrow 1$, $a[2] \leftarrow 1$;

For $i = 2$ to $N$:

Set $a[i] \leftarrow a[i - 1] + a[i - 2]$;  //Comment: I will claim that $a[i] = f_i$...

Output $a[N]$;

Now to prove correctness, one can just prove that for any fixed input value $N$, each entry $a[i]$ equals $f_i$, which implies that the output $a[N]$ is the desired value. This is very easy, since the array entries obey the same base-cases and recurrence relation that one typically uses to define $f_i$ in the first place. But in more interesting examples, the algorithm may use a different method of computation from that in the definition, yet arrive at the same quantities.

In this class we need to prove that the program variables end up with the desired values that we have defined mathematically as our “targets”. For this reason, it is key that you give the program variables different names from the target values. Otherwise you will confuse the reader, and might confuse yourself into thinking that there is no need to prove correctness.

For example, it is typical to use $OPT$ to denote the maximum achievable value of some optimization problem, and then often we might define an $i$th subproblem whose optimum value is referred to as $OPT(i)$. You might observe that these values obey some recurrence relation, and it’s natural to write an iterative program to compute these quantities. Just give the program variables a different name, say $Y_i$, and prove by induction on $i$ that your computed values satisfy $Y_i = OPT(i)$.

**Use induction.**

It’s always worth emphasizing this. When analyzing an algorithm that takes many steps, we are always trying to show some kind of gradual progress over time. The additional progress we make at each execution step is reliant upon what we have accomplished in prior steps, so we should analyze step by step. It is usually best to formalize such an argument by induction (instead of something hand-wavy like “the pattern becomes clear”). Being explicit about this makes it clear to graders that you know how the proof is supposed to look, even if some element of your inductive proof ends up being flawed.
When using induction, clearly identify the induction variable \( k \) and the statement \( S(k) \) to be proved for a given value of \( k \).

In the Fibonacci algorithm I gave above, it might be tempting to state and inductively prove the statement \( S(k) : \) “algorithm A halts with the correct output \( f_k \) on input \( k \)”.

The difficulty with this is that A is an iterative algorithm, not a recursive one. A does not call itself on smaller inputs, so it’s not so clear how to use the inductive hypothesis to extend to the value \( k + 1 \). Instead, we fix an input \( N \); and we prove by induction on \( k \in \{1, 2, \ldots, N\} \) that during this execution \( A(N) \) each program variable \( a[k] \) eventually gets set to \( f_k \). This is easy.

In general, the two key features \( S(k) \) should have are:

1. \( S(k) \) should capture the essential progress we have made, that makes further progress possible (so, we need \( S(k) \) to contain the information that allows us to derive \( S(k + 1) \));

2. The “final” statement \( S(n) \) should imply that our algorithm has reached our goal.

These points suggest that we should make \( S(k) \) as strong and informative as we can. This is not completely true, however, because the stronger we make \( S(k) \), the more we also have to prove when deriving \( S(k + 1) \).

Avoid logical pitfalls by paying close attention to language.

Here is a proposal for a linear-time sorting algorithm, \textsc{ultrasort}: on input \( a_1, \ldots, a_n \), if \( n = 1 \) then do nothing, else split the list in half and apply \textsc{ultrasort} on each half.

I claim the algorithm is correct on any input \( a_1, \ldots, a_n \). Proof: induction on \( n \). The base case is easy—any 1-element list is sorted. Now suppose true for \( n = 1, \ldots, N \), and consider \( n = N + 1 \). Well by inductive hypothesis, the left half ends up sorted, and the right half ends up sorted; so the whole list is sorted!

Of course this is bogus. The source of the error is failing to analyze the meaning of “sortedness” carefully enough, and distinguishing between a list whose two halves are internally sorted, versus one that is globally sorted.

A similar but trickier issue appeared in one of our HW problems. Suppose we are trying to find a local minimum of a rectangular array of numbers. One approach is to just chop the array in half and find a local minimum on one of the two sides. But what does this mean? Values on the rightmost edge of the left may be considered
“local minima” when we restrict our attention to their adjacent cells within the left half, but they also have neighbors on the right half and we need to consider these too when trying to argue that what a recursive program call gives us is a solution to our larger problem. To obtain a correct efficient algorithm for this problem, it turns out to be helpful to think about the more subtle notion of an internal local minimum within a grid—one that does not live on the outer boundary. This distinction should get us thinking about making a stronger inductive hypothesis about what a recursive call on a subgrid returns, so that we do not get back a faulty “local minimum” that lives on the edge and may fail to be a local minimum in the larger grid.