How to write excellent proofs

Comp 160: Algorithms

Like any other kind of writing, it takes practice to write a good proof. In fact, it is actually difficult to define what is a good proof. Indeed, a good proof must have the following properties:

- Correct
- Each statement clearly follows from the previous one
- Easy to follow
- . . .

Think on it for a bit: can you find other requirements? You will notice that those are necessary requirements but not sufficient (as in, we can have a proof that satisfies those constraints but for some reason we still would not consider it a good proof). This is true beyond the lecture environment: it sometimes happens that proofs that were considered to be correct by hundreds of mathematicians turned out to be wrong. Just to name an example: around the late 19th century, there were two different proofs of the Four Color theorem (https://en.wikipedia.org/wiki/Four_color_theorem). Both proofs were incorrect, but the mistake was not shown until 11 years after their publication. The result is still considered correct, but the only proof known to the day is via exhaustive computer search.

Fortunately, in Comp 160 we will not go through this level of scrutiny, nor the statements we are looking to prove be so difficult. Nevertheless, we still want proofs to be rigorous (and of course, correct). In the following we will give you some guidelines that can help you writing good proofs.

1 General guidelines

State clearly what you want to prove Remember that a proof is meant to be read by others (indeed, in this class your proofs will often be graded by others). Clarity of intention is good for both you and the reader. For example,

- If the question specifically says prove $X$ then you can write Claim: $X$.
- If $X$ is verbose and there is a way to shorten/summarize it, consider doing so.
- If the question is how would you solve problem $Y$?, then start with a punchline: a summary of your approach or statement of a claim you will prove.
State clearly how you will prove it Closely related to the previous one, remember to guide the person reading your statement. Rather than simply starting with the math, spend a few lines say something like We will show this statement by induction on k (where k is the number of prime numbers). Then you can launch into the proof: For k = 0 the statement is true because.....

Know your audience This sometimes comes as a surprise, but your audience is not the TAs or professor of the class, though they are certainly the ones who will be evaluating your work. They already know the correct solution and do not need to be convinced. Instead, you should imagine your audience is a reasonably skeptical classmate who does not already know the solution, and your job is to convince them that your answer is correct.

Clearly guide from line to line Try to explain what you did from line to line. For example, line 3 to line 4 is “by induction”, line 4 to line 5 is “by properties of logarithms”, and so on. Only when you are rearranging terms or doing something very simple you can ignore the rule.

Avoid large walls of text When a proof is not well structured, the description tends to repeat the same ideas several times in hopes that at least one of the sentences sounds convincing. No one likes to see long paragraphs of text that go over the same idea over and over.

Review your own writing Once you finish writing a proof, let it rest for 1-2 days. Only then take another look and ask yourself: Does this make sense? Is there any sentence that is not needed? Should I have explained where this comes from? This has the additional bonus that forces you to do the homework early. Never rush for a deadline!

Start from the beginning and walk forward It may sound a bit obvious but: a proof should end with the thing you’re trying to prove, it should not begin with it. Do not start from the goal and walk backwards to reach the initial conditions. Note that you can (and should) clearly state what your final goal is, just give the breadcrumbs that are needed to get there in order.

2 General example

Here are two different solutions to the same problem.\(^1\) Which one is easier to understand?

\[
(0 - 3)^2 + (x - 2)^2 = 25 \\
3^2 = 9 + (x^2 - 4x + 4) = 25 \\
x^2 - 4x - 12 \\
(x - 6)(x + 2) \Rightarrow x = -2, 6 \quad x > 0 \quad x = 6
\]

\(^1\)From Francis Su at Harvey Mudd College, [https://www.math.hmc.edu/~su/math-writing.pdf](https://www.math.hmc.edu/~su/math-writing.pdf)
WHY THIS IS POORLY WRITTEN:
- You don’t know what problem the writer was solving.
- You can’t tell what’s an assumption and what’s a conclusion.
- Where does one thought end and another begin? There are no sentences!
- In the 2nd line: combining two thoughts can create untruths (3^2 is 9 but it is not 25).
- The 3rd line dangles; what’s being asserted here? It’s not a sentence.
- What’s the relationship between all these phrases? Connective phrases would help!

**Problem 1:** find a point in the plane on the positive x-axis that has distance 5 from the point (2, 3).

**Solution:** The desired point is (6, 0).

To find this, we note if (x, 0) is a solution, then x must satisfy the equation \((x - 2)^2 + (0 - 3)^2 = 25\), which follows from the planar distance formula between the points (x, 0) and (2, 3). It follows that \(x^2 - 4x + 13 = 25\). Then

\[ x^2 - 4x - 12 = 0. \]

Factoring, we obtain

\[ (x - 6)(x + 2) = 0, \]

satisfied by either \(x = -2\) or \(x = 6\). Since we assumed \(x > 0\) and \(y = 0\), we see (6, 0) is the desired point.

WHY THIS IS WELL WRITTEN:
- The writer described the problem, and strategy for solution.
- Every thought is a full sentence with subject and verb (the “equals” sign is a verb).
- The question is answered right at the beginning. (Boxing answers is customary.)
- Even the equations have punctuation (comma, periods) as they are part of sentences.
- Important ingredients are highlighted; important equations are displayed.
- Trivial algebra is avoided: the intended audience (a Comp 160 student) should be comfortable with the missing steps.

3 Proof styles for Comp 160

Now we will focus on specific tricks and tips that we think will be particularly useful for 160. During the course we will see many assignments. Naturally, they will all be different, but we identify three common themes:

3.1 Massaging an equation

Although Comp 160 does not have many straight calculation exercises, some math formulations appear. Because of this, we may have exercises of the type prove that (some formula of the runtime) is \(O(\text{another expression})\). The answer to these questions are relatively easy to
structure: you want to show that the initial expression is equal to something else, or finding
an upper bound (when we look for a big O bound). For these type of exercises the most
important thing to do is to go step by step and justify each line.

**Problem 2:** prove that $3n^4 + 7n^3 + 100 = O(n^4)$.

*Proof.* We will show that there exists $n_0$ and $c$ such that $3n^4 + 7n^3 + 100 \leq cn^4$, $\forall n > n_0$.
In our case, we will use $n_0 = 10$ and $c = 11$. Since $n > 0$ we can upper bound $7n^3$ by $7n^4$, and thus we get:

$$3n^4 + 7n^3 + 100 < 3n^4 + 7n^4 + 100 = 10n^4 + 100$$

Now we use the fact that for $n > n_0 = 10$ we have $100 < n^4$:

$$10n^4 + 100 < 11n^4$$

That is, for $n > 10$ we have upper bounded the initial expression by a constant times $n^4$.
This is exactly the definition of big O, and thus the statement is shown.

Note the cute ☐ symbol on the previous line (right side). This square denotes that the
proof has ended and that discussion will focus in a different topic. Although we can often
deduce when a proof ends, it can be helpful for the reader. As such, we encourage you to
use it in all of your proofs. Fun fact: if you use \LaTeX, the program will automatically put
the symbol for you. Isn’t this a good motivation for using such a nice program?

### 3.2 Designing an algorithm

This kind of question tries to simulate the difficulty of transforming a real-life problem into
an expression that computers can help. Often they are given in the context of a story.

**Problem 3:** You encounter $k$ hungry dragons, and you fortunately have $k$ hamburgers to
feed them so they do not eat you. The golden rule in the dragon world is that you respect
size: if any dragon sees a second dragon that is smaller in size but is given a bigger ham-
burger than the one given to the first one, then it will get jealous and you will be eaten.
What algorithm would you use to feed the dragons?

Naturally, a computer cannot handle hamburgers or dragons. So the key part of the
assignment is to relate the story to the processes that a computer can recognize. In the
previous example the fact that you want to respect size gives you some kind of ordering.
Can we use that to our advantage? if so, how?

For this kind of problem, having a good structure in the proof is a must. Although you
are free to use any good system you like, we generically recommend a structure with three
parts: the punchline, the justification and extra details, described as follows:

**Punchline** Summarize your algorithm in a single sentence. Note that you are just describing
the algorithm, which could be as simple as *use this or that technique*. If there is any
modification needed, then just describe the differences.
**Justification**  Here again we look for a short justification of why your algorithm is correct. No need to go into all details, but mainly specify the tricks that you extracted from the text and how you formalized them.

**Details**  Depending on the problem you may need to address extra details. For example, did you make any extra assumptions? Is there any case that might need special consideration? For extra clarity we recommend that you list all cases/assumptions using bullet points.

**Runtime**  How fast is your algorithm? Try to give a bound that is as tight as possible. If amount of memory needed by the algorithm is not obvious, it never harms to also give bounds on that.

Let’s see a sample answer following this format:

<table>
<thead>
<tr>
<th>Problem 3:</th>
<th>describe an algorithm for feeding dragons respecting their relative size.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm:</strong></td>
<td>Sort the dragons by size from largest to smallest. Likewise, sort the hamburgers by size from largest to smallest. Now you can feed the largest hamburger to the largest dragon, the second largest hamburger to the second largest dragon, and so on.</td>
</tr>
<tr>
<td><strong>Justification:</strong></td>
<td>This algorithm works because if we label the dragons $d_1 \ldots d_k$ from largest to smallest and the hamburgers $h_1 \ldots h_k$ from largest to smallest, then $d_i$ is fed hamburger $h_i$. And if $d_i &gt; d_j$, then we must have $h_i \geq h_j$ (they are both sorted in the same order). Thus, dragon $d_i$ will not get jealous (and you will not be eaten).</td>
</tr>
<tr>
<td><strong>Details:</strong></td>
<td></td>
</tr>
</tbody>
</table>
  - We assume that we can compare two hamburgers and determine which one is bigger (and the same for dragons!).  
  - We must make sure that our sorting algorithm can handle ties (i.e., having two hamburgers or dragons of the same size)  
  - Note that ties can be handled arbitrarily since a dragon will only eat you if a strictly smaller dragon gets a strictly bigger hamburger, which cannot happen with a good sorting algorithm. |
| **Runtime:** | We need to sort the $k$ dragons and hamburgers. If we use MERGE_SORT, this will take $\Theta(k \log k)$ time. The pairing of dragons and hamburgers is afterwards done in $\Theta(k)$ time. This makes the overall runtime equal to $\Theta(k \log k)$. |

### 3.3 Auxiliary Lemma

An auxiliary Lemma is a minor result that is used as a stepping stone to solve a larger result. When you are facing a difficult problem, we encourage you to split it by introducing 1 (or more) Lemmas so that the overall explanation becomes easier to read.

For example, in Comp 15 you probably learned about binary search and how we can use it to find an element fast in a sorted array. Now try to justify why this technique runs in $O(\log(n))$ time. When you think about the fundamental reason that this algorithm works, you realize there is a key idea, and sometimes you may want to focus on this key idea with a separate lemma. For the binary search algorithm, the auxiliary lemma we are looking for
could be the following one:

**Lemma:** Each step of a binary search eliminates at least half of the values.

*Proof.* Let $a_1, \ldots, a_n$ be a sequence of integers such that $a_i < a_{i+1}$ for all $1 \leq i < n$, and let $t$ be another integer. Compare $t$ with $a_{\lceil \frac{n}{2} \rceil}$. If $t = a_{\lceil \frac{n}{2} \rceil}$ we are done. If $t < a_{\lceil \frac{n}{2} \rceil}$, then it holds that $t < a_j$ for any $j \geq \lceil \frac{n}{2} \rceil$, so we eliminate $a_{\lceil \frac{n}{2} \rceil} \ldots a_n$ which is at least half the values. Similarly, if $t > a_{\lceil \frac{n}{2} \rceil}$, then it holds that $t > a_j$ for any $j \leq \lceil \frac{n}{2} \rceil$, so we eliminate $a_1 \ldots a_{\lceil \frac{n}{2} \rceil}$ which is at least half the values. \hfill \Box

Notice that the text above does not show that binary search runs in $O(\log(n))$ time. How would you prove the original claim making use of this lemma?

### 4 Conclusion

Like driving, learning how to write a good proof takes practice. Whenever you get feedback on your work take the time to read and incorporate it. We hope you enjoy developing this important skill.
Appendix A  Frequent Mistakes

Many questions in 160 are on purpose open ended to simulate real life algorithmic choices that you will find in your career. Although we think it is great practice for students to learn, we understand that it leads to unclear situations for the students. The best guidance that we can give is that we try to be very consistent. Use the feedback you got in previous exercises and try to apply it to future ones.

In the following we give a few example solutions (inspired by what students submitted as answers in the past) so that you can learn from their answers.

Consider the following problem:

**Problem 4:** You are given an array of \( n \) numbers and an integer \( k \leq n \). Design an algorithm that can find the \( k \)-th largest element in the array

Note that in this course we will learn a technique that can almost directly solve this problem. For the sake of practice, assume that this technique has not been explained yet (and thus it is not available for students).

Let’s see a few answers and why we think they are good or bad:

Tip 1: be mindful when using techniques outside Comp 160

<table>
<thead>
<tr>
<th>Problem 4: given an array ( A ) of ( n ) numbers, find the ( k )-th largest element</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Algorithm:</strong> I found on internet an algorithm called the <em>Blum, Floyd, Pratt, Rivest, and Tarjan</em> algorithm that solves the problem in linear time.</td>
</tr>
<tr>
<td><strong>Justification:</strong> Wikipedia says that this algorithm runs in linear time</td>
</tr>
<tr>
<td><strong>Details:</strong> no assumptions needed.</td>
</tr>
</tbody>
</table>

The answer above is technically of correct (the algorithm mentioned is the one we will teach in class and can be used to solve the problem), but as you can imagine this is not the type of answer that we would like. Can you tell why?

Keep in mind that questions are meant to evaluate how much has the student understood the contents of the course. The response from the student only shows that they are capable of using an internet browser.

The answer to virtually all questions in this course will be a minor modification of a technique presented in the class. If your solution needs major changes and/or a completely new algorithm step back and ask yourself: am I going in the right direction? Can I maybe try a different approach? We encourage you to consult with a friend or TA to discuss your idea and why you think it is right (of course, this is not possible during an exam).

Although **we do not forbid using techniques outside this class**, be very mindful:

- Often the techniques have some requisites for when they can be used. For example, the Blum, Floyd, Pratt, Rivest, and Tarjan algorithm mentioned by the student will be covered in class. However, this algorithm computes the \( k \)-th smallest element (not the largest one as requested by the question).
- Indeed, one can modify the algorithm so that it computes the largest one, but the
answer is not as simple as plug in this answer. Sometimes the requirements are more convoluted (say, the algorithm only works if input values are positive). Have you looked at all requirements? Do all of them apply?

- Finally, a practical reason: say that you can find a very creative solution that uses cool techniques you learned somewhere else. The grader may not know those methods, and thus will not understand your explanation.

**Tip 1 (addendum): beware of too simple answers**

The same reasoning applies if your answer **only** uses techniques from prerequisite courses (say, you just try all possibilities). Say we are deep into the course and for some problem you think that it can be solved using a very simple technique, step back and think again. You may be right, but make sure to carefully prove so.

Instructors design exercises so that the best answer uses the techniques explained in class. Most likely you are working in an incorrect direction.

**Tip 2: avoid code/pseudocode**

Let’s look at a different answer:

```plaintext
For j = 0 to k - 1 do
    m[j] ← A[0]
For i = 0 to n - 1 do
    if (A[i] > m[j] and A[i] < m[j - 1]) then m[j] ← A[i]
Return m[k - 1]
```

First, the obvious comment is that the solution does not follow our proposed format, so even if it is perfectly correct it will not get full credit. Even if we ignore this and look at the solution for a while, you will see what the student wanted to do: first find the largest element, then the second largest, and so on until we find the k-th largest element and return that value. Although in spirit the answer is correct, it is easy to find cases where this code will fail (for example, when $j = 0$ the program will access out of range. Even if this were fixed, it will have problems when $A$ contains repeated elements).

Still, it is good to keep in mind that **this class is not a programming course**. If you are taking this course, we already know you can handle minor details (such as the repeated values in $A$). Ask yourself: how many lines of code would the correct algorithm take? I would say that at least 10 lines are needed. And once you wrote them down, what would be simpler to understand? Those 10 lines of code or a couple of sentences in plain text?

Remember that the ultimate goal of the exercise is NOT to test your coding skills. Instead, we are interested in whether or not you can properly analyze the problem.

**Tip 3: limit description to what changed**

Now let’s look at a great solution:
Problem 4: given an array $A$ of $n$ numbers, find the $k$-th largest element

Algorithm: Use MergeSort to sort $A$. Then return the value stored in $A[n - k + 1]$.

Justification: After sorting $A[i]$ contains the $i$-th smallest element. We want the $k$-th largest element among $n$ numbers, which is equivalent to the $(n - k + 1)$-th smallest element.

Details: I assume that the array is 1-indexed (that is, the values are from $A[1]$ to $A[n]$). Also, I assume that $k \geq 1$

Runtime: MergeSort needs $O(n \log n)$ time. Other modifications only need $O(1)$ time.

Let’s discuss why we love it:

Great presentation Beautifully structured answer. Hopefully you will agree with us that this is very easy to read.

Incremental solution In class we explain MergeSort algorithm. The student clearly identifies what technique from class to use for the problem, and what to modify (in this case very little)

Assume previous knowledge Unless otherwise stated you can always assume content explained in lectures (so, no need to explain MergeSort). This makes the answer compact.

Focused justification You probably know that MergeSort ... well, sorts. However, in this algorithm we are using it for something else (finding a specific number). The justification focuses in the relationship between sorting and the goal of the algorithm (note again, that you do not need you to justify that MergeSort is correct. That will be done in class).

Appendix B Other rules to consider

When answering any question (be it in recitation, homework or an exam) we always assume you follow these guidelines:

- Justify your answers. A well justified incorrect answer is preferable over a good answer with no justification. When the question asks you to give an algorithm, you must justify that the algorithm is correct.
- We only care about asymptotic values. Suppose the question is how many comparisons does your algorithm do? The answer $42n + 15$ is as good as $\Theta(n)$ (as long as both are equally well justified)
- If a problem admits more than one solution and both can be equally well justified, the one with a faster algorithm is more desirable. Do not forget to state (and justify) the runtime of any algorithm you describe
- We are always interested in a general algorithm. Even if the question describes a particular example, give an algorithm that can work in all cases.
- Although we are not against pseudocode, it tends to make it difficult to create a well structured proof. Avoid using it whenever possible. Here is a general hierarchy of answer styles. Try to be as high on the list as possible:
  - Well-structured proof like the examples in this document
- Bullet points
- Pseudocode
- Wall of text (notice this is LAST on the list!)

- Every question that we ask can be solved with tools presented in this course. It is sometimes ok to use tools that we did not introduce, but it must be presented in a similar way (justify why it is correct, discuss runtime, and so on). If you’re not sure, ask.