Approaching Algorithmic Problems

Sam McCauley
Algorithms, Spring 2025

Goal for Today

One of the most difficult parts of algorithms is approaching a new problem for the first time.

This document is strategies to get past this. We'll talk about three things. First, tips to start the proof—a way to start making progress from the beginning. Second, we'll talk about how to approach the main ideas for the proof. Finally, we'll give some tips for getting unstuck when looking for a proof.

We'll see some examples for each of these, along with solutions. In all cases, these solutions aren't full solutions to the problems—they are examples of how to use these techniques to come up with proof ideas. (In all cases, these solutions are the first step towards a full answer to the question being posed.) These examples focus on topics from the beginning of class to try to make them more accessible; however, they work for all topics we'll see throughout the class.

The Beginning

Ask yourself: what does the *beginning* of your solution look like? In other words, you were given an algorithmic problem to solve. In this part, go over the details of the problem. What exactly what said? What assumptions were made? What do you already know about the problem being asked?

Let's be a little more specific. Let's say the problem involves a list of numbers. You want to ask yourself questions like:

- Are the numbers all positive?
- Can some number appear multiple times?
- Are the numbers integers? From a bounded range?

If the problem involves a graph, you can ask similar questions:

- Is the graph directed or undirected?
- Are the edges weighted? Can the weights be negative?

These questions are not only useful for getting started, they can help you avoid missing information in the problem that is key to the solution.

Once you have a good handle of the problem being asked, the question is: what can you say about the problem *immediately*? Phrased differently, what could the first few lines of your proof look like?

Tip 1. To begin a proof, you should:

- Read carefully over any assumptions made in the question. Ask yourself if there are implicit assumptions.
- Replace high-level concepts with formal definitions and name variables.

Let's look at a concrete example of how we can use this tip to start a proof.

Example. You are given an array A of numbers in sorted order and a number x. Give an algorithm to determine if x is stored in A and prove that it is correct.

Solution. The first thing I notice about the problem is that the array A is in sorted order. There aren't any other restrictions on A—so it could contain multiple copies of numbers, negative numbers, floating point numbers, etc.

Let's think about how to restate "A is in sorted order" more formally. In other words, what can we immediately say about A? Here's one good option. In English, I'd say that since A is in sorted order, each element is smaller than the next. Mathematically, I might write "Since A is in sorted order, for all i, $A[i] \leq A[i+1]$".

There can be multiple answers to these questions. Another good option would have been: "Since A is in sorted order, for all $i < j$, $A[i] \leq A[j]$".

The Middle

What might the *middle* of your proof—the part with most of the content—look like?

In particular, what you should ask yourself is: **is there a similar problem where you already know the solution?** There are several ways in which a problem can be similar. Let's discuss some of them.

Tip 2. When approaching the proof for a first time, ask yourself the following questions to help come up with ideas of what the proof might look like.

- Is there a similar problem you've seen the answer to?
  - Perhaps the problem has a similar setup (for example: maybe both problems are on sorted arrays of numbers).

  - Perhaps the problem is asking for a similar solution (for example: maybe both problems ask for a shortest path).
  - Perhaps we are looking for an algorithm that looks similar to one you've already seen.
- Are there techniques we've seen in class recently that one could use? Perhaps these techniques don't obviously apply—but if we do try to apply them, what happens? Is there an obstacle to getting them to work?

While this method is likely the most powerful, it may sometimes lead to dead ends. Try a few things and see what works!

It's worth mentioning that when first using this idea, students sometimes think that it's not a good method—in other words, one may think that it's better to come up with a solution from scratch rather than trying adapt something you've already seen. This is not the case. Not only is looking for similar strategies a good idea in this class, it is how the majority of algorithms research—perhaps all scientific research—is done.

Let's look at a real-world[1] example of a sorting algorithm question.

Example. Prove that the following algorithm correctly sorts an array a of length n.

```
for(i = 0; i < n; i++)
   for(j = 0; j < n; j++)
       if(a[i] <= a[j])
            swap(a[i], a[j]);
```

Solution. This algorithm's behavior seems reasonable at first glance, but in fact it is extremely odd. Note, for example, that in some cases $i < j$—in this case, the algorithm swaps $a[i]$ and $a[j]$ so that the *larger* element is first.

Let's think if there is an invariant the algorithm satisfies. As mentioned above, we can use similar algorithms as a starting point. Here are the two invariants we saw in class:

- **Selection sort:** after the outer loop runs i times, the i largest elements of the array are in the correct position in the array.
- **Insertion sort:** after the outer loop runs i times, the first i elements of the array are in increasing order.

[1] Here is a forum question asking this question: https://cs.stackexchange.com/questions/45348/why-does-this-sort-algorithm-work

Let's do some examples and see if either of these work. Let's start with an array consisting of $[2, 3, 1]$. After we run the outer loop once, the array consists of $[3, 2, 1]$. Neither the largest nor smallest element is in sorted order.

The insertion sort invariant seems to work so far. Let's do the second iteration of the loop, with $i = 1$. The array then consists of $[2, 3, 1]$. This is consistent with the insertion sort invariant.

We now have an idea: perhaps the insertion sort invariant is true for this algoritm as well. We can proceed as we did for insertion sort in class. The next step will be to set up an induction where the inductive hypothesis is "after the outer loop runs i times, the first i elements of the array are in increasing order". As it turns out, this proof does work for this algorithm.

Getting Unstuck

Anyone who works with algorithms will, sometimes, find themselves stuck. Here are some ideas to help get unstuck.

Work Through Small Examples

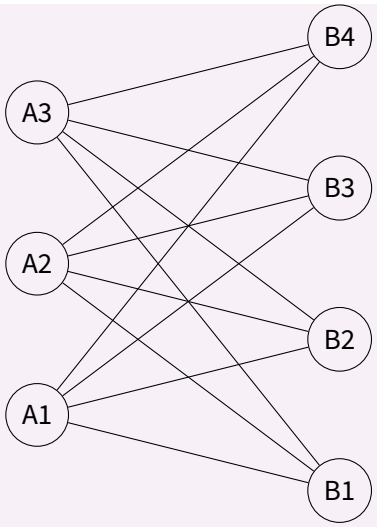
Tip 3. Try some small examples and see what happens. Is the claim true? Can we get more information about why it's true?

Example. In a *bipartite* graph, the set of vertices V can be partitioned into two subsets A and B, such that each edge is between a vertex in A and a vertex in B.

Prove that if a graph is bipartite, it cannot contain a cycle with odd length.

Solution. Let's draw a bipartite graph and see what happens.

Here's a bipartite graph:



Let's try to come up with an odd-length cycle. I'll start at A_1 and try to get back in an odd number of steps. For example, I may go $A_1, B_1, A_2, B_3, A_3, B_1, \ldots$. Pretty soon I'm noticing that my steps always alternate between the left side and the right side.

In fact, I'm just about done, except for a little formalization. I can prove the following statement (induction and contradiction both work here): any path in G alternates between a vertex in A and a vertex in B. Since any cycle begins and ends in the same set, and alternates between the two sets, it must have even length.

Try to Come up With a Counterexample

Let's say we tried some examples and they worked. Let's look back at what happened. Can we tweak the example so that the claim is no longer true?

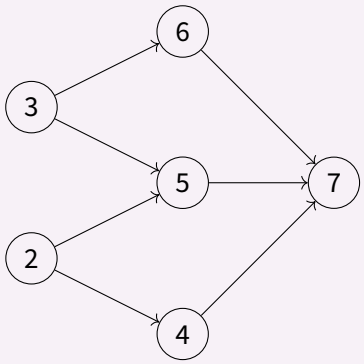
The answer is probably no: no matter what we do to the counterexample, the claim still holds. The thing to look for here is: why is it true? Does the solution change as we change the counterexample? These observations can give extremely valuable information about the proof.

Tip 4. Try to alter your small examples to *disprove* what you're trying to prove. Odds are good that your attempts won't work. Why not? See if this gives intuition to help you prove the claim.

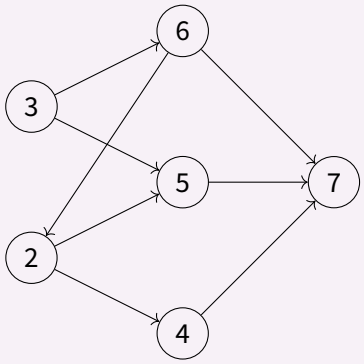
Let's look at a time when trying to come up with a counterexample can give valuable information about the problem. We have seen a proof of the following statement in class.

Example. If G is a directed graph, and G does not contain a cycle, then G has at least one vertex v with no incoming edges.

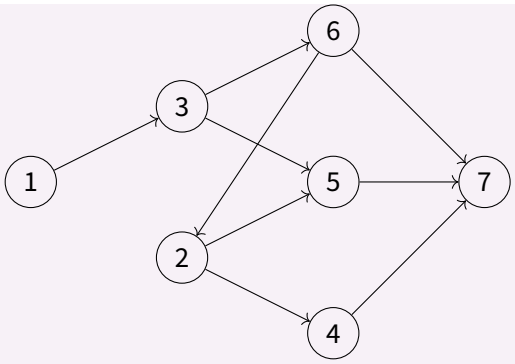
Solution. Let's start to build up a counterexample. Let's start with the following graph:



Vertices 2 and 3 have no incoming edge. We can add an edge from vertex 6 or 3 to vertex 2 without creating a cycle. Let's do that.



Vertex 3 still has no incoming edges. In fact, we can't add an edge from any vertex to 3 without creating a cycle; we have to add a new vertex.



Now, vertex 1 has indegree 0; again we need to add a new vertex to fix it. This seems like it will always be a problem: if there's only one vertex left with no incoming edges, we can't add an edge into it without creating a new vertex.

The final proof of this example has this observation at its heart. In short, the proof starts at an arbitrary vertex v, and traces backward through the incoming edges. Since there's no cycle, no vertex can repeat as we go backward—so eventually we must come to a vertex with no indegree.