Revisiting the First Four Steps

Step One: Work an Example Yourself

The first step to devising an algorithm is to work an instance of the problem yourself. As we discussed earlier, if you cannot do the problem, you cannot hope to write an algorithm to do it—that is like trying to explain to someone how to do something which you yourself do not understand how to do. However, you have to not only be able to do the problem, but also do it methodically enough that you can analyze what you did and generalize it.

Often, a key part of working the problem yourself is drawing a picture of the problem and its solution. Drawing a clear and precise picture allows you to visualize the state of the problem as you manipulate it. Having a clear idea of the state of the problem, and how you are manipulating it will help you with the next step, in which you write down precisely what you did on this instance of the problem.

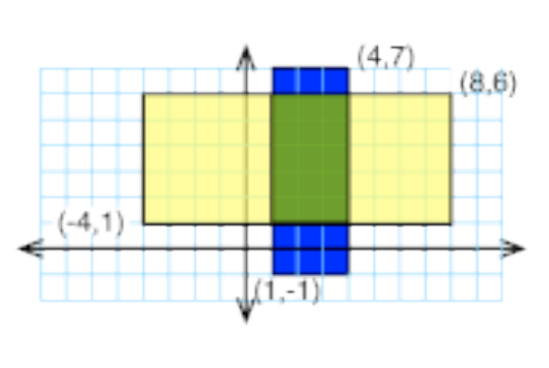
We will use the following problem as an example to work from for the rest of this chapter:

Given two rectangles, compute the rectangle which represents their intersection.

You may assume the rectangles are vertical or horizontal.

The first thing we should do here is to work *at least* one instance of this problem (we may want to work more). In order to do this, we need a bit of domain knowledge—what a rectangle is (a shape with 4 sides, such that adjacent sides are at right angles), and what their intersection is (the area that is within both of them).

What instance we pick is really up to us. For some problems, some instances will be more insightful than others, and some will expose *corner cases*—inputs where our algorithm needs to behave specially. The most important rule in picking a particular instance of the problem is to pick one that you can work completely and precisely by hand.



Working an Example of the Rectangle Intersection Problem

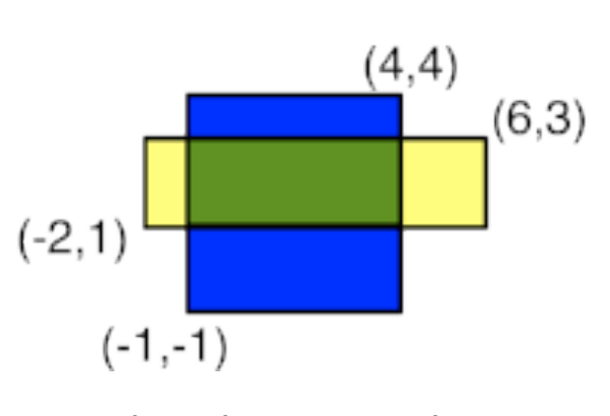
The figure above shows the results of Step 1 for the rectangle intersection problem. We picked an instance of the problem—here the yellow-shaded rectangle from (-4,1) to (8,6) intersecting with the blue-shaded rectangle from (1,-1) to (4,7). The resulting intersection is the green-shaded rectangle from (1,1) to (4,6).

You should note a few things about this example. First, while the yellow/blue/green coloring is not truly a part of the problem, there is nothing wrong with adding extra information to your diagram to help you understand what is going on. Second, note that the diagram is done precisely—we drew a Cartesian coordinate grid, and placed the rectangles at their correct coordinates. This precision ensured that any information we obtained from analyzing our diagram was correct and not a result of sloppy drawing (though whether some relationship is generally true, or a consequence of the specific case we chose is not guaranteed by a careful drawing). You typically do not need to draw things with draftsman-level precision, but the more precise you can be, the better.

In this case, we can tell the answer just by looking at the picture and seeing where the green region is. However, to write a program to do this, we need to figure out the math behind it—we need to be able to work the problem in some way other than just looking at the diagram and seeing the answer. Sometimes trying work things mathematically is hard when you can just see the answer. Learning to put the obvious aside and think about what is going on is a key programming skill to learn, but takes some time.

Another Instance of the Rectangle Problem, with the Cartesian Grid Removed

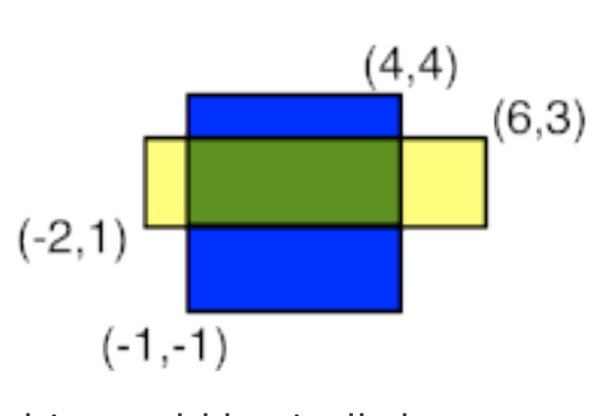
If you struggle with this, it may be useful to work another instance of the problem, but eliminate extra information that lets you jump straight to the answer without understanding. The figure below shows a different instance of the rectangle problem with the Cartesian grid removed (note that it was still drawn such that the rectangles are the right size and in the correct relative positions). We can still precisely work the problem from this diagram, but it is a little harder to just look at the Cartesian grid and see the answer. Take a second to work out the answer before you continue.



Note that there is nothing wrong with working a few instances of the problem, taking different approaches as you do it, and including/excluding various extra information as you do so. In general, it is better to spend extra time in an earlier step of programming than getting stuck in a later step (if you do get stuck, you might want to go back to an earlier step and redo it with another instance of the problem). For Step 1, doing a few different instances of the problem is preferable to moving into Step 2 and only being able to come up with "I just did it—it was obvious."

Step Two: Write Down What You Just Did

Now, you are ready to think about what it was *exactly* that you just did, and write down a step-by-step approach to solve one specific instance of the problem. Using the figure we just looked at:



this would basically be a set of steps anyone could follow to find the intersection of the rectangle from (-2,1) to (6,3) with the rectangle from (-1,-1) to (4,4). Note that you are not trying to generalize to any rectangles here, just writing down what you did for *this particular* pair of rectangles.

There are actually two important pieces to think about here. The first is how you represented the problem with numbers. Remember the key rule of programming: Everything is a number. Since a rectangle is not a number (in the way, for example, that the price of bread is a number) we will have to find a way to properly *represent* a rectangle using a number (or several). If you go back and read the descriptions of the instances of the problems we worked, you will find that we already have been representing each rectangle with 4 numbers—two for the bottom left corner, and two for the top right corner. Now that we have assured ourselves that rectangles are numbers, we know that we can happily compute on them—we also have an idea of what information we should think of a rectangle as having (each of which is just a number): a bottom, a left, a top, and a right.

The second thing we need to think about is what exactly it was that we did to come up with our answer, and write it down. These steps can be anywhere in the spectrum of pseudocode—notation that looks like programming, but does not obey any formal rules of syntax—to pure natural language (e.g., English) that you are comfortable with. The important thing here is not any particular notation, but to have a clear idea of what you did in a step-by-step fashion before you try to generalize the steps.

For example, we might write the following:

I found the intersection of

- **left: -2**
- **bottom: 1**
- **right: 6**
- **top: 3**

and

- **left: -1**
- **bottom: -1**
- **right: 4**
- **top: 4**

by making a rectangle with

- **left: -1**
- **bottom: 1**
- **right: 4**
- **top: 3**

In this case, we do not have many steps, but it is still crucial for us to write them down.

Step Three: Generalizing Values

Now that we know what we did for this particular instance, we need to generalize to all instances of the problem. We will note that this step is often the most difficult (you have to think about why you did what you did, recognize patterns, and figure out how to deal with any possible inputs) and the most mistake prone (which is why we test the algorithm in Step 4 before we proceed).

One aspect of generalizing your algorithm is to scrutinize each value you used, and contemplate what it is in the general case. Is it a constant that does not change depending on the inputs? Does it depend on one (or more) of the parameters? If it does depend on some of the parameters, what is the relationship between them?

Going back to the rectangle example on which we did Step 2, we came up with **-1** for the left value of the answer rectangle. We can quickly rule out the idea that this is a constant—surely not all rectangles have -1 as the left side of their intersection (counterexamples would be easy to come by if we needed to convince ourselves).

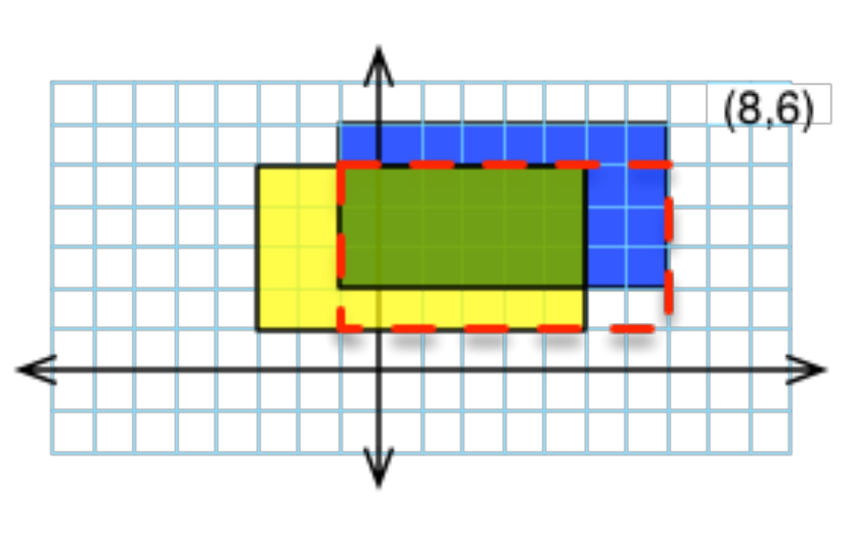
Now we are left figuring out how -1 relates to the input parameters. It could be that the left value of the answer rectangle matches one of the values of the input rectangles—both the left and the bottom of the second rectangle are -1. It could be the case that it has some mathematical relationship to another value—maybe the left of the first rectangle divided by 2, or plus 1, or maybe the negative of the bottom of the first rectangle. Any of these would yield -1, and work in this case, but we need to think about *why* the answer is -1 to figure out the correct generalization.

Sometimes this analysis is quite difficult. Whenever you get stuck on generalization, it can help to repeat Steps 1 and 2, to give us more information to work with and more insight. For example, looking back at the other example we worked first in Step 1, we can rule out some of the ideas we pondered in the prior paragraph. From these two examples, we might draw the conclusion that the left value of the intersection is the left value of the second rectangle. We might proceed similarly to generate the following generalized algorithm (as with Step 2, notational specifics do not matter as long as you are precise enough that each step has a clear meaning):

To find the intersection of two rectangles, r1 and r2:
Your answer is a rectangle with

- **left: r2's left**
- **bottom: r1's bottom**
- **right: r2's right**
- **top: r1's top**

While these generalized steps accurately describe the two examples we did, they are in fact not a correct generalization.



This figure shows a pair of rectangles where our algorithm gives the wrong answer—shown a red dashed rectangle. If we make an incorrect generalization such as this, we *should* catch it in Step 4 (or if not, then when we test the code at the end of Step 5). In such a case, we must return to Step 3 before proceeding, and fix our algorithm.

When you detect a mis-generalization of your algorithm, you have the advantage that you have already worked at least one example which highlights a case you need to analyze carefully. In this case, we can see that we want r1's right (*not* r2's right) for the right side of the answer, and r2's bottom (*not* r1's bottom) for the bottom side of the answer. Note that r1's right and r2's bottom did not work for the earlier cases, so we cannot simply change our algorithm to use those in all cases. Instead, we must think carefully about when we need which one and why.

Careful scrutiny will lead us to conclude that we need the minimum of r1's right and r2's right, and the maximum of r1's bottom and r2's bottom. We may also realize that we should do something similar for the left and top (if not, we should find that out when repeating Step 4). We could then come up with the following correctly generalized steps:

To find the intersection of two rectangles, r1 and r2:

- **Make a rectangle (called ans) with**
- **left: maximum of r1's left and r2's left**
- **bottom: maximum of r1's bottom and r2's bottom**
- **right: minimum of r1's right and r2's right**
- **top: minimum of r1's top and r2's top**

That rectangle called ans is your answer.

We will note that in the case of rectangles that do not intersect, this algorithm will produce an illogical rectangle as the answer (its top will be less than its bottom and/or its left will be greater than its right). For the purpose of this problem, we will say that giving such an invalid rectangle in these cases is the intended behavior of the algorithm—in part because we have not learned how to represent "no such thing" easily.

Generalizing Repetitions

Another important part of generalizing an algorithm is to look for repetitions of the same (or similar) steps. When similar steps repeat, you will want to generalize your algorithm in terms of how many times the steps repeat (or until what condition is met). To examine this aspect of generalizing, we will deviate from our rectangle example (which does not have this type of repetition), and consider a slightly different problem for a moment:

Given an integer N (>0), print a right triangle of *s, with height N and base N.

For example, if N = 4, you would print

- *
- **
- ***
- ****

We might work an example with N=5, and end up with the following result from Step 2:

- **Print 1 star**
- **Print a newline**
- **Print 2 stars**
- **Print a newline**
- **Print 3 stars**
- **Print a newline**
- **Print 4 stars**
- **Print a newline**
- **Print 5 stars**
- **Print a newline**

Here, we are doing almost the same thing (Print i stars; Print a newline) 5 times. Once we observe the repetition, we can take one step towards generalizing the algorithm by re-writing the algorithm like this:

- **Count (call it i) from 1 to 5 (inclusive)**
  - **Print i stars**
  - **Print a newline**

Notice that the way we have re-written the algorithm here gives us two new constants to scrutinize: the 1 and the 5 in the range that we count from/to. Careful consideration of these would show that 1 is truly a constant (we always start counting at 1 for this algorithm), but 5 should be generalized to N:

- **Count (call it i) from 1 to N (inclusive)**
  - **Print i stars**
  - **Print a newline**

This algorithm is correct for the triangle-of-stars problem. Sometimes it takes a little more work to make the steps of your algorithm match up so that you can describe them in terms of repetition. For example, consider the following problem:

Given a list of numbers, find their sum.

We might work this problem on the list of numbers 3, 5, 42, 11, and end up with the following result from Step 2:

- **Add 3 + 5 (= 8)**
- **Add 8 + 42 (= 50)**
- **Add 50 + 11 (= 61)**
- **Your answer is 61**

Scrutinizing each of these constants might lead us to the following more general steps:

- **Add (the 1st number)   + (the 2nd number)**
- **Add (the previous total) + (the 3rd number)**
- **Add (the previous total) + (the 4th number)**
- **Your answer is (the previous total)**

Here, we almost, but not quite, have a nice repetitive pattern. We can, however, make the steps match up:

- **previous_total = 0**
- **previous_total = Add previous_total + (the 1st number)**
- **previous_total = Add previous_total + (the 2nd number)**
- **previous_total = Add previous_total + (the 3rd number)**
- **previous_total = Add previous_total + (the 4th number)**
- **Your answer is previous_total**

Note that mathematically speaking, what we did was exploit the fact that 0 is the additive identity—0 + N = N for any number N. We will also note that starting with the identity element as our answer before doing math to the items in a list is typically a good idea, since the list may be empty. Often, the correct answer when performing math on an empty list is the identity element of the operation you are performing. That is, the sum of an empty list of numbers is 0, the product of an empty list of numbers is 1 (the multiplicative identity). Now that we have re-arranged our steps, we can generalize nicely:

- **previous_total = 0**
- **Count (call it i) from 1 to how many numbers you have**
  - **previous_total = Add previous_total + (the ith number)**
- **Your answer is previous_total**

In this example, we also did something that will make Step 5 (translating to code) a bit easier—naming values that we want to manipulate. In particular, we gave a name to the running total we compute, which means that not only is it clear exactly what we are referencing when we say **previous_total**, but also that when we reach Step 5, this will translate directly into a variable.

Generalizing Conditional Behavior

Sometimes when we are generalizing, we will have steps which appear sometimes, but not others. Such a situation may be a matter in which we perform a step for some parameter values, but not for others; or in which we have steps that are almost repetitive, but some actions which appear in some repetitions but not in others. In either case, we need to figure out under what conditions we should do those steps.

It may take some work and thinking to determine the patterns for what conditions we need to perform those steps under, and what conditions we do not. As with many things in generalizing, if it is not immediately apparent, it can be quite useful to work more examples—giving you more information to generalize from. You might also find it informative to make a table of the circumstances (parameter values, information specific to each repetition, etc.) and whether or not the steps are done under those circumstances.

Once you have figured out the pattern, you can express the step in the algorithm more generally by describing the condition that should be determined, and what to do if that condition is true, and what to do if it is false. Doing so makes your algorithm a little bit more general, and may help you express a large sequence of steps as repetition, since they will now be more uniform.

Generalizing Is an Iterative Process

Generalization is an iterative process—you take what you have, generalize (or rewrite it) a bit, and then try to generalize that result more. Sometimes one step of generalization opens up new avenues of generalization that were not visible before. We have already seen how recognizing repetitive patterns can lead to the opportunity to generalize in terms of how many times you do the repeated steps. You may also end up exposing the repetitive pattern of some steps only once you have figured out what the generalization of the values in those steps is.

Step Four: Test your Algorithm

Once you have generalized your Algorithm, it is time to test it out. To test it out, you should work it on *different* instances of the problem than the one(s) you used to come up with it. The goal here is to find out if you mis-generalized before you proceed. We have already seen one instance of mis-generalization in our rectangle problem, in which our algorithm was too specific to the examples from which we had built it (always using r1's bottom, r2's left, etc...). Testing on these same examples would not have revealed any problems.

In doing this testing, you want to strike a balance—enough testing to give you confidence that your algorithm is correct before you proceed, but not an excessive amount of testing. Note that in this testing, you perform your steps by hand, so it may be somewhat slow for a long or complex algorithm. You can do more extensive testing after you translate your algorithm to code. The tradeoff there is that the computer will execute your test cases (which is fast), but if your algorithm is not correct, you have spent time implementing the wrong algorithm.

Here are some guidelines to help you devise a good set of test cases to use in Step 4:

- Try test cases that are qualitatively different from what you used to design your algorithm. In the case of our rectangle example, the two examples we used to build the algorithm were both fairly similar, but the third example (which we used to show the flaw) was noticeably different—the rectangles overlapped in a very different way.
- Try to find *corner cases*—inputs where your algorithm behaves differently. If your algorithm takes a list of things as an input, try it with an empty list. If you count from 1 to N (inclusive), try N=0 (you will count no times) and N=1 (you will count only one time).
- Try to obtain *statement coverage*—that is, between all of your test cases, each line in the algorithm should be executed at least once. We will discuss various forms of test case coverage later in the course.
- Examine your algorithm and see if there are any apparent oddities in its behavior (such as it always answers “true”, it never answers “0” even though that seems like a plausible answer), and think about whether or not you can get a test case where the right answer is something that your algorithm cannot give as its answer

Revisiting the Intersection of Two Rectangles

This video reviews how to apply the first four steps to the problem discussed above (finding the intersection of two rectangles).

1.4.1 Intersection of Two Rectangles

Day 4, Part 1

Steps 1-4 Review
Please complete this review quiz on Steps 1-4