What is an Algorithm?

[(https://github.com/learn-co-curriculum/phase-1-algorithms-what-is-an-algorithm)](https://github.com/learn-co-curriculum/phase-1-algorithms-what-is-an-algorithm) [(https://github.com/learn-co-curriculum/phase-1-algorithms-what-is-an-algorithm/issues/new)](https://github.com/learn-co-curriculum/phase-1-algorithms-what-is-an-algorithm/issues/new)

Learning Goals

- Define "algorithm"
- Outline the steps for solving problems using algorithms

Introduction

One of the main focuses of this section is writing **algorithms** to solve problems. While the word **algorithm** may sound intimidating, its definition is actually quite straightforward:

> An **algorithm** is simply a well-defined procedure, or set of instructions, for accomplishing a specific task.

In the real world, you utilize algorithms all the time. If you had to describe how you wash your hair, you would likely outline the following instructions:

```
lather
rinse
repeat
```

Some other real-world algorithms that you might use all the time are recipes. For instance, if you want to make some delicious chocolate chip cookies, you'd want complete the following steps:

```
1. Preheat the oven to 350 F.
2. Microwave the butter for about 40 seconds. Butter should be completely melted
   but shouldn't be hot.
3. In a large bowl, mix butter with the sugars until well-combined.
```

```
4. Stir in vanilla and egg until incorporated.
5. Add the flour, baking soda, and salt.
6. Mix dough until just combined. Dough should be soft and a little sticky but
   not overly sticky.
7. Stir in chocolate chips.
8. Scoop out 1.5 tablespoons of dough and place on baking sheet.
9. Bake for 7-10 minutes, or until cookies are set. They will be puffy and still
   look a little under baked in the middle.
```

Any set of instructions or procedure describing how to complete a task could also be categorized as an algorithm. If you're wondering if that's any different from the work you've already been doing when working on labs, we've got good news: It's not! You've already had plenty of practice writing your own algorithms to solve problems based on specific sets of instructions. From this point on, however, we'll be focusing more deliberately on *how* to solve those problems and developing a process for writing your own algorithms.

How to Solve Problems

When given any problem to solve with code, two mistakes many programmers make are to jump into code too fast or to start thinking about code optimization too early. Both of these mistakes can greatly increase the amount of time it takes to solve a problem and increase frustration.

In general, any time you write code, your priorities are to:

- Make it work
- Make it right (i.e., readable)
- Make it fast

Here are the steps we recommend taking:

1. Rewrite the Problem in Your Own Words

Before you dive into solving the problem, take the time to read it and describe it in your own words. You might find it useful to rewrite the problem before moving on.

Pay attention in particular to the problem's **inputs** and **output**: what kind of data will you be working with? What kind of data should be returned?

If you have been given test cases, look at each one, apply your understanding of the problem to them to determine what the answer is, and then check if your answer matches the actual answer (e.g. work it out on paper or in your head, no code necessary here). If your answer doesn't match, you need to spend more time understanding the problem.

2. Write Your Own Test Cases

Now that you understand the problem and why the answers to the test cases are what they are, you're ready to write your own test cases! Writing your own test cases doesn't mean that you need to write entire test suites in Rspec or Jest. The important thing is to identify what inputs your code should handle and what outputs it should return, and then to find a quick way to check your work once you've written the solution.

One way to do this is to simply print the result of calling your solution method and compare it to the answer you expected. For example, if you were writing an algorithm to find all the elements in an array that are numbers:

```
// I expect calling getNumbers with an array [1, "5", 3] will return [1, 3]
console.log("Expecting: [1, 3]");
console.log("=>", getNumbers([1, "5", 3]));

// I expect calling getNumbers with an array [1, 3] will return [1, 3]
console.log("Expecting: [1, 3]");
console.log("=>", getNumbers([1, 3]));
```

Be aware that algorithm problem descriptions rarely provide all of the test cases you need to account for, so it's incredibly important that you also come up with your own. This is true when using online platforms, such as Leetcode and HackerRank, as well as during interviews.

3. Pseudocode

Remember how we asked you to check your understanding of the problem by going through the test cases and then writing your own tests? Congratulations! This means that on some level, you know how to solve the problem. Before you start coding, write pseudocode, which is just a plain description of how to solve the problem. For example, the pseudocode for copying only numbers from one array to another might look like this:

```
initialize empty array called result

iterate over each item in the input array:
  if element is a number:
    push item onto result

return result
```

Note that different people write pseudocode differently. The key is to make it easy to understand yourself and explain to others — this is the map to the code you're about to write! It's helpful to copy your pseudocode into your workspace as comments, and then code each piece alongside the matching comment.

You can also test your pseudocoded procedure against the test cases before writing any code. Validating and rewriting pseudocode will likely save you time. You might also wish to think about additional solutions: there's always more than one way to solve a problem.

4. Code

Now that you have a map, convert it to code!

At this point, the goal is to make it work: pass those test cases! If you're having a hard time getting all of the test cases working, check that your pseudocode actually solves for all of those cases and then check that your code does what the pseudocode says it should.

5. Make It Clean and Readable

Once your solution is well, a solution, it's time to refactor your code so that it's easy to read, not just for you but also for others. Use well-named variables and convert blocks of code to functions when necessary. If you find any unneeded or redundant code, delete it.

Here are some things to look for when writing clean code:

- Is it easy to read?
  - Make sure you use indentation correctly!
  - Use meaningful variable names. `s` is less meaningful than `substring` .

- Add comments when your code needs an explanation. Don't add comments to every line. But if your code is doing something out of the ordinary, or if you wouldn't understand the code yourself after looking at it in the future, it's good to add a comment to explain your intent.
- Don't try and cram everything into one line of code, just because you can (unless you're doing **code golf** **(https://en.wikipedia.org/wiki/Code_golf)**).

- Does it follow the **Single Responsibility Principle** **(https://en.wikipedia.org/wiki/Single-responsibility_principle)** ?
  - Keep functions focused on one specific task.
  - Create helper functions when needed.
- Does it have unnecessary lines?
  - Review your code for redundant/unnecessary lines. Are there more elegant ways of expressing your code?
  - Avoid unnecessary comments. Use well-named variables and helper functions rather than adding a comment to every line. Good code is its own best documentation!
- Is it DRY?
  - Avoid repeating yourself. Write helper functions to abstract away repetitive logic.

Don't forget to test your code again!

6. Optimize

As a last step, think about how to optimize your code for time or space complexity (e.g. how long it takes to run or how much memory it's using). In an interview setting, don't optimize unless you need to; it's better to have a working solution that you can discuss with the interviewer than to run out of time in the middle of optimizing your code. There are three major situations that call for optimization:

- The solution is hanging on certain test cases, and therefore cannot pass since it's taking too long
- You were asked to do so
- You think it would be fun to try

We'll spend more time later in this section discussing how to identify the time and space complexity of a function, so save this step for last.

Conclusion

We hope these steps help you solve the problems you're about to encounter. Remember, they can be applied to all types of problems, including building web apps. **Don't be afraid to spend more time thinking and planning than coding**. Take it from those of us who have been coding for years: we often spend more time thinking, talking, and writing than we do coding.

Before we go, we'd like to leave you with some final tips:

- Talk to yourself while you code: think out loud
- Consider recording your screen and voice as you solve a problem so you can review your performance
- Take your time and be patient with yourself!