# Section 4:Problem Solving Approach

## 17. PREREQUISITES

### Prerequisites For This Section

### It is strongly recommended you first complete the following sections before working through this section:

### Section 1: BIG O NOTATION

## 18. Introduction to Problem Solving

<https://cs.slides.com/colt_steele/problem-solving-patterns>

### PROBLEM SOLVING

#### Understand the Problem

#### Explore Concrete Examples

#### Break it Down

#### Solve Or Simplify

#### Look Back and Refactor

## 19. Step 1: Understand the Problem

1. Can I restate the problem in my own words?
2. What are the inputs that go into the problem?
3. What are the outputs that should come from the solution to the problem?
4. Can the outputs be determined from the inputs? In other words. Do I have enough information to solve the problem? (You may not be able to answer this question until you set about solving the problem. That’s okay; it’s still worth considering the question at this early stage.)
5. How should I label the important pieces of data that are a part of the problem?

## 20. Step 2: Explore Concrete Examples

Coming up with examples can help you understand the problem better

Examples also provide sanity checks that your eventual solution works how it should

**User Stories!**

**Unit Tests!**

### EXPLORE EXAMPLES

* Start with Simple Examples
* Progress to More Complex Examples
* Explore Examples with Empty Inputs
* Explore Examples with Invalid Inputs

## 21. Step 3: Break it Down

### Explicitly write out the steps you need to take.

This forces you to think about the code you’ll write before you write it , and helps you catch any lingering conceptual issues or misunderstandings before you dive in and have to worry about details (e.g. language syntax) as well.

## 22. Step 4: Solve Or Simplify

### Simplify

* Find the core difficulty in what you’re trying to do
* Temporarily ignore that difficulty
* Write a simplified solution
* Then incorporate that difficulty back in

## 23. Step 5: Look Back and Refactor

### Refactoring Questions

* Can you check the result?
* Can you derive the result differently?
* Can you understand it at a glance?
* Can you use the result or method for some other problem?
* Can you improve the performance of your solution?
* Can you think of other ways to refactor?
* How have other people solved this problem?

## 24. Recap and Interview Strategies

#### Understand the Problem

Ask questions of your interviewer and of course clarify the problem and make sure you understand it.

#### Explore Concrete Examples

Understand Inputs

Understand Outputs

Understand Edge cases

How to handle errors

What happens when a user enters something invalid

#### Break it Down

Break out a couple of steps to lay out a game plan for the code you need to implement

This is really helpful especially in an interview a few times and you know that you’re going to run out of time

So make sure you know where you’re going before you just start typing code

#### Solve Or Simplify

If you can’t solve the problem right away, solve a problem that you can even if it’s simpler. If you simplify you remove some core difficulty core challenge and try to solve something where you can then reincorporate that core difficulty back in.

If you’re not sure how to test for something being alphanumeric, ignore that part and just write the problem, write the solution ignoring the alphanumeric part so that you could plug it in later.

#### Look Back and Refactor

Most important for learning or improving as a developer.

Looking back on your code and sometimes refactoring it sometimes just looking back and analyzing it and patting yourself on the back. Most of the time there’s room to refactor even if you're a totally expert developer. It can be very useful to look at other people’s solutions or to scrutinize your own and figure out what you could have done better.