Reading Code

The answer to algorithms questions is some sort of code, and to write code you have to be able to "read" code.

Code in the context of a PM interview is different than real-world code. Much of the boilerplate machinery is stripped away, leaving the core logic plus whatever's needed for the specific language.

There are too many languages to cover, so we won't try. There is, however, the concept of "pseudocode". This is code written in no specific language but *looks* like real code and, more importantly, has the same key features as real code.

What are those features?

Variables

Variables represent things. For our purposes, they will almost always represent some sort of information, such as a number, string, or a more complex data object. A variable has a name (which you use to refer to it in your code) and a value (which will be known by the computer when the code runs).

Variables must be "declared" by the programmer. This means that every variable that is used must be known beforehand. Not the value of the variable, but just the fact that the variable needs to be there. For example, if you had to keep track of the biggest and smallest numbers in a list, you'd just need two variables: one to hold the biggest, and one to hold the smallest. However, if you needed to keep track of every even number that you saw, you couldn't use variables since you don't know how many you'll need. You can't declare an infinite number of variables beforehand. You can, however, use lists.

Lists

Lists are a series of values. The key fact here is that its values can be accessed by `index`. In other words, you're allowed to ask a list for its 4th or 1st or 103rd element. The elements held inside lists are arbitrary (aka it doesn't matter), but for our purposes are usually strings or numbers.

Lists are sometimes called "arrays". For our purposes, they are the same thing.

Functions

A function in code is like a function in math. Inputs go in, something happens, and a result comes out. The code you write inside the function provides the "something happens" part. Functions delineate logically separate chucks of code, and will have a name.

A simple problem will only require you to write the code for one function, but a more complex problem may require multiple functions.

Control Flow

Code runs from top to bottom. An earlier line will be run before a later line.

There are four exceptions:

- **If/Else:** Sometimes you don't want certain lines of code to run. Sometimes you want a different bit of code to run. You can control which lines of code run by specifying under which circumstances for those lines of code.
- **Loops:** Sometimes you'll have a large amount of data that needs the same thing to happen to every element. Similar to how you can't declare an infinite number of variables, you also can't write out the same line of code infinite times.
  However, you can tell your code to do something a given number of times.
  For example, `print this string 5 times` or `say hi to every person in this list`.
  Sometimes, the conditions are more elaborate, instead of just being counting based. `say hi to everyone until you see brad` or `say hi to every other person in this list`.
  Loops can do all of this.
- **Breaks:** Occasionally you'll need to exit a loop before it has completed. A `break` statement halts execution of the loop and returns to the main code.
- **Returns:** Remember, the purpose of functions is to have an output. Returns specify what that output is.
  If your code reaches a return, it stops, even if there's more code after it. This means returns are usually at the end of the function, or represent special cases and are placed within `if` statements.
  Returns are like breaks, but for the whole code/function, not just a loop.

At the risk of stating the obvious, code *flows* from top to bottom, and these mechanisms *control* changes to that flow, hence *"control flow"*.

Logical operators

In programming, the words `and` and `or` have very precise definitions.

"Something is true if `a` is true **and** `b` is true" means both `a` and `b` both have to be true.

"Something is true if `a` is true **or** `b` is true" means just one of them need to be true. Both *can* still be true, but only one *needs* to be.

To solidify this concept, the following example is commonly cited:

```
true and true = true
true and false = false
false and true = false
false and false = false

true or true = true
true or false = true
false or true = true
false or false = false
```

This concept can be nested and repeated within itself:

```
a or b
(a and b) or c
(a or b) and (c or d)
etc.
```

Actually Reading Some Code

Sorry that was so long, but we needed to establish some common vocabulary to move forward faster.

Remember earlier I mentioned pseudocode? That's what we'll end up writing for our algorithms, so that's what we should reduce any given code to (in our heads).

Let's take a look at how each of those concepts might manifest.

Variables

`int x = 2`

`x = 2`

`var x = 2`

`let x = 2`

`x` is the name of our variable, and its value is on the right (`2`). The equals sign is typically how we know something is a variable.

When coding, a single `=` represents *assignment*. We're *telling* the computer that the thing on the left is now equivalent to the thing on the right (and always in that direction). If we want to express *equality*, we use two equal signs, `==`. This is *asking* the code "are these two things equivalent?"

Different languages will have different syntax in front of the variables. For our purposes, they don't matter. The core concept in all four lines above is that `x` is equal to `2`.

Lists

`var foo = []`

`foo = [1,2,4]`

`int[] foo = [12,3,45]`

`arr[int] foo = [12,3,45]`

Lists are universally represented by square brackets, `[]`.

The notation is reused to refer to a given element within a list.

```
foo = ["hi", "hello", "hey"]
print(foo[1])
```

`foo[0]` refers to the "0th" element of the list `foo` (foo is a **variable** whose value is the list `["hi", "hello", "hey"]`)

Code is "zero-indexed". This means the first thing in a list is number 0, the second thing 1, etc. It's a quirk, and most people will know what you mean if you mess up, but just try not to. It's a n00b mistake.

In the above list, foo[0] = "hi", and foo[1] = "hello"

Functions

`func foo()`

`function foo()`

`def foo()`

`int function foo(int a, int b)`

`void func foo(string a, int b)`

Functions need to have names so they can be run. In the above examples, all three functions are named "foo".

Function have inputs, called parameters. Parameters are universally denoted with parenthesis at the end of the function name.

The names of the parameters are specified within the parentheses. Within the function, we refer to the inputs by the variable names set in the function name.

```
function add(a, b):
    sum = a + b
    return(sum)
```

The variables `a` and `b` are named in the first line, and used in the second.

On the other end, we use the same notation to *give* parameters to a function when we want to use it.

```
x = 1
y = 2
add(x,y)
```

We are telling our code to run the `add` function with the values of `x` and `y`.

Control Flow

If/Else

```
if (a > b):
    asldfjas
else:
    qruwqeou
```

```
if (a > b):
    asldfjas
else if (a < b):
    qruwqeou
else:
    poipoksf
```

The code in the parenthesis after `if` and `else if` specify the conditions under which the code that follows (represented by the keyboard mash) is run.

So if `a = 1` and `b = 2`, then we would run `qruwqeou`.

For-loops

`for (i = 0; i < 10; i++)`

The code will run 10 times. If you need to, you can even reference the variable `i` inside the code. With each iteration, `i` will update. The first time it's 0, then it's 1, then it's 2, up to 9.

`i++` is a common shorthand to mean "increment `i` by 1". It's equivalent to `i = i + 1`

`for item in list`

The code runs for as many times as there are items in the list. Just like `i` held the number it was assigned, `item` will hold the corresponding item in the list.

```
list = ["hi", "hello", "hey"]
for (item in list):
    print(item)
```

Will print "hi", then "hello", then "hey".

While-loops

`while x < 10`

`while x`

`while foo()`

While-loops run until the provided condition is false. Be careful that something ensures that `x` will eventually be `false`, otherwise your code will never stop running.