#Characteristics of an Algorithm

No matter what kind of algorithm you're writing, it's likely to have some basic components that you should be able to discuss succinctly. This lesson will cover the basic building blocks of an algorithm, but we also want it to serve as a reminder: learning the right vocabulary can really help you on your developer journey.

As with many other topics in coding, it's possible to discuss algorithms in everyday language–but that's not the best idea. If you're entirely new to computer science and coding, you should strive to build the vocabulary necessary to appropriately discuss these topics, both for the sake of communicating with your peers and for seeking out answers online when you hit a dead-end. Sometimes the difference between finding the exact answer you need or getting the run-around for thirty minutes is a single key word that you forgot to use.

#The Basics

You don't need to make flashcards and memorize these necessarily, but you should make an effort to remember the broad details of the following algorithm vocabulary:

- **Input:** An algorithm generally has some input values. We can pass 0 or more input values to an algorithm.
- **Output:** All algorithms have some desired output. We will get 1 or more outputs when an algorithm finishes its work.
- **Unambiguity**: An algorithm should be unambiguous, meaning that the algorithm's instructions should be clear and simple.
- **Finiteness:** An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions. To put it simply: the instructions should be countable.
- **Effectiveness:** Every step in an algorithm should be effective. If you have extra steps that aren't influencing the overall process/outcomes of your algorithm, they should be eliminated.
- **Language-independent:** An algorithm must be language-independent so that the instructions in the algorithm can be implemented in any programming language in order to output the desired values. This is why we use **pseudocode** and **flowcharts** before attempting to write our code–they represent a language-independent version of our algorithms.

#Dataflow of an Algorithm

When describing an algorithm, we use the following to get into the details:

- **Problem:** A problem can be a real-world problem or any sub-task from a real-world problem for which we need to create a program or set of instructions.
- **Algorithm:** An algorithm is the set of instructions we design to solve our problem. It consists of step by step procedures according to the vocabulary in the above list.
    1. **Input:** After designing an algorithm, the required and the desired inputs are provided to the algorithm.

2. **Processing unit**: The input will be given to the processing unit, and the processing unit will produce the desired output.
3. **Output:** The output is the outcome or the result of the program.

#Why Do We Need Algorithms?

Some key terms that will be recurring frequently throughout the rest of the course are best introduced now, in light of why we need algorithms:

- **Scalability:** Scalability refers to our ability to grow a project to operate with more users, inputs, or required outputs. One major reason for writing algorithms is to ensure that we can **scale** our solutions. For example: using an array to contain ten data points is more efficient than making ten separate variables. Organizing for scalability is one of the chief concerns of algorithm design.
- **Performance:** No matter what our project does, it should do it well enough to meet user expectations. Good performance is a constant concern, and this is especially true of algorithms. Real-world problems aren't easily broken down into smaller steps, but the better we do so, and the better we tackle each of those steps, the more we can ensure that our algorithm remains **performant**.

#Summary

Algorithms represent our first deep-dive into the real complexity of computer programming. We touched on them briefly when we discussed concurrency and multithreading, but now we're going to get into the details. The first part of this week represents a good chance for students who have trouble breaking their code down into bite-sized steps to grow that skill. Moreover, we can all benefit from practicing breaking bigger problems down into smaller ones, as it's the best way to ensure that your path to being a professional programmer is a smooth one.

We've now discussed some tools (flowcharts and pseudocode) to use, as well as the vocabulary you'll need when discussing these topics with other devs or researching them for your own problem-solving needs. Feel free to come back to this page whenever you need a quick reminder, or simply search up the words yourself if you get confused.

#Extra Resources

This algorithm playlist from Bro Code is pretty great. There's a lot to learn there, though, and our course won't be covering everything that you see on the list. If you'd like to click through a few of the videos as we tackle the different algorithms this week, it'd be a good idea. Maybe give it a bookmark?

Understanding Search

We've discussed algorithms in a general sense and tried our hand at making some throughout the first few weeks at Academy. We understand how to use pseudocode and flowcharts to get our brains going in the right direction, and how to write code

using conditional and flow control statements to make them work. Now, it's time to talk about the big two: **search** and **sort**. For this lesson, we're focusing on **search** algorithms in particular.

#Search Algorithms

Searching for information is an everyday activity, but have you ever thought about how it actually works? You give a computer a search term, and then it magically returns the data you want–sometimes without even making you click a link. As you grow as a developer, you're going to need to understand how algorithms work and how to implement them in your code. You're not going to be asked to recreate a search engine overnight, but we do have to start somewhere:

#Linear Search

Linear search is a simple algorithm that involves looking for a given element in a list by iterating through the list element by element, until the desired element is found. The algorithm takes in an **array** and a **key** (the element we want to find) as **input**. It then iterates through the array, checking each element to see if it is equal to the key. If it is, the function returns the index at which the element was found. If the element is not found, the function returns `-1` to declare the absence of the desired element.

This is the equivalent of looking for your friend in a hotel by knocking on every door and asking if they're in there. It's not very efficient, but it will get the job done, assuming that you have the processing power (and the time) to devote to checking every single element.

Here's a simple Java code example that demonstrates how to perform a linear search in an ArrayList to find a specific element:

```
import java.util.ArrayList;

public class LinearSearchExample {
public static int linearSearch(ArrayList<Integer> list, int
target) {
for (int i = 0; i < list.size(); i++) {
if (list.get(i) == target) {
return i; // Return the index of the target element if found
}
```

```
}
return -1; // Return -1 if the target element is not found in
the ArrayList
}

public static void main(String[] args) {
// Create an ArrayList of integers
ArrayList<Integer> numbers = new ArrayList<>();

// Add some elements to the ArrayList
numbers.add(10);
numbers.add(20);
numbers.add(30);
numbers.add(40);
numbers.add(50);

// Define the target element to search for
int target = 30;

// Perform a linear search to find the target element in the
ArrayList
int index = linearSearch(numbers, target);

if (index != -1) {
System.out.println("Element " + target + " found at index " +
index);
} else {
System.out.println("Element " + target + " not found in the
ArrayList");
}
}
}
```

In this example:

- The `linearSearch` function takes an ArrayList of integers (`list`) and a target value (`target`) as parameters.
- It iterates through the ArrayList elements using a for loop.
- For each element in the ArrayList, it checks if the current element is equal to the target value.
- If it finds the target value, it returns the index of the target element.
- If the target value is not found in the ArrayList, it returns -1 to indicate that the target element is not present.
- In the `main` method, we create an ArrayList, define a target element (30), and use the `linearSearch` function to search for the target value in the ArrayList.
- The result of the search is printed to the console.

#Binary Search

A **binary search** algorithm works with a similar principle, but is much more effective, assuming that the array we're searching is an **ordered (aka sorted) list**. By ordered/sorted, we mean that the elements in the array have been stored in a sequential order, such as alphabetically for Strings or numerically for numbers. Binary search cannot be implemented if the elements are stored in a random manner.

Because binary search continuously breaks a larger problem down into smaller ones, it's closer to say, finding a word in a printed dictionary. Say you were searching for the word, "terrific." You'd open to the back half of the dictionary, because T is late in the alphabet, then adjust from there. This idea of splitting things into halves (over and over again) is what makes binary a more powerful option, so long as the collection is in order.

Think back to your lessons about Collections. Where have you heard the term "binary" in reference to a Collection? What kinds of data structures will work well with binary search, based on what you know?

Save Response

Reset to template

Here's a simple Java code example demonstrating binary search in an ArrayList:

```
import java.util.ArrayList;

public class BinarySearchExample {
public static int binarySearch(ArrayList<Integer> list, int
target) {
int left = 0;
int right = list.size() - 1;

while (left <= right) {
int mid = left + (right - left) / 2;

if (list.get(mid) == target) {
```

```
return mid; // Element found, return its index
}

if (list.get(mid) < target) {
left = mid + 1; // Target is in the right half
} else {
right = mid - 1; // Target is in the left half
}
}

return -1; // Element not found in the ArrayList
}

public static void main(String[] args) {
// Create a sorted ArrayList of integers
ArrayList<Integer> numbers = new ArrayList<>();
numbers.add(10);
numbers.add(20);
numbers.add(30);
numbers.add(40);
numbers.add(50);

// Define the target element to search for
int target = 30;

// Perform a binary search to find the target element in the
sorted ArrayList
int index = binarySearch(numbers, target);

if (index != -1) {
System.out.println("Element " + target + " found at index " +
index);
} else {
System.out.println("Element " + target + " not found in the
ArrayList");
}
}
}
```

In this example:

- The `binarySearch` function takes a sorted ArrayList of integers (`list`) and a target value (`target`) as parameters.
- It uses a while loop to repeatedly divide the search range (`left` to `right`) in half.
- The algorithm calculates the middle index (`mid`) and compares the value at that index to the target value.
- If the target is found, it returns the index where it was found.

- If the target is smaller than the middle element, it updates the `right` boundary to search the left half.
- If the target is larger than the middle element, it updates the `left` boundary to search the right half.
- The loop continues until the target is found or the search range is exhausted.
- In the `main` method, we create a sorted ArrayList, define a target element (30), and use the `binarySearch` function to search for the target value in the ArrayList.
- The result of the search is printed to the console.

#Summary

There are times when we will have to invent our own algorithms, but there are other times when we can use more off-the-shelf solutions. The above algorithms for searching are frequently used, especially in cases when our data sets are small. As our data sets grow, we'll need more complex (and more efficient) algorithms to ensure that we can achieve our goals in a timely manner, but these algorithms are a great place to start thinking about the ways in which our problems can be broken down into smaller pieces.

#Knowledge Check

In linear search, the inputs given to the function are:
an array or a collection
element to search for
a function

Check Answer

If an element is not found using linear search, what does the function return?
null
Zero.
For an array named array, it returns `array.length + 1`
-1

Check Answer

In order to use binary search, the given array must be:

Sorted

Unsorted

Reversed

Randomized

Check Answer

Problem Statement

Write a Java program to perform Linear Search on an ArrayList.

#Code Breakdown

1. Import the necessary classes:
   - `import java.util.ArrayList;`
   - `import java.util.List;`
2. Define the class `LinearSearchList`:[This code is provided in template]
   - This class will contain the method to perform a linear search on a `List` of integers.
3. Define the `linearSearch` method:
   - This method takes in a `List` of integers and an integer `target` as parameters.
   - The method uses a for-each loop to iterate over all of the elements in the `List`.
   - For each element, the method checks if the current value is equal to the `target`.
   - If the target is found, the method returns `true`.
   - If the loop completes and the target was not found, the method returns `false`.

4. Define the `main` method:
   - In this method, create an `ArrayList` of integers called `numbers` and add the following integers to it:

     InfoWarningTip

     You can add any integers you like into the list OR use the following one.

     ```
     List<Integer> numbers = new ArrayList<>();
     ```

```
numbers.add(3);

numbers.add(6);

numbers.add(2);

numbers.add(9);

numbers.add(11);
```

- Your target is an integer with a value of 6.
- Call the `linearSearch` method with the list and the target as arguments.
- Store the result of the search as a boolean variable called `result`.
- If the result is `true`, the message "`Target found in the List.`" should be printed to the console.
- If the result is `false`, the message "`Target not found in the List.`" should be printed to the console.

#Expected Input & Output

Given the following input:

```
List<Integer> numbers = new ArrayList<>();
numbers.add(3);
numbers.add(6);
numbers.add(2);
numbers.add(9);
numbers.add(11);
```

Your program should printout "Target found in the List".

```
import java.util.ArrayList;
import java.util.List;

public class LinearSearchList {

    // linearSearch() method that takes in a List and an integer target as par
ameters
    public static boolean linearSearch(List<Integer> list, int target) {
        // Iterate over all elements in the List using for-each loop
        for (int num : list) {
            // Check if the current value is equal to the target
            if (num == target) {
                // If so, return true
                return true;
            }
        }
        // If the target was not found, return false
```

```
        return false;
    }

    public static void main(String[] args) {
        // Create an ArrayList of integers and populate integer values
        List<Integer> numbers = new ArrayList<>();
        numbers.add(3);
        numbers.add(6);
        numbers.add(2);
        numbers.add(9);
        numbers.add(11);

        // Define the target to search for
        int target = 6;

        // Call the linearSearch method with the list and target as arguments
        // Store the result in a variable of type boolean
        boolean result = linearSearch(numbers, target);

        // Check the result and print the appropriate message
        if (result) {
            System.out.println("Target found in the List.");
        } else {
            System.out.println("Target not found in the List.");
        }
    }
}
```