WEEK-08
Learning Objectives

Binary Trees and Binary Search Trees

The objective of this lesson is for you to become comfortable with implementing common data structures. This is important because questions about data structures are incredibly likely to be interview questions for software engineers from junior to senior levels. Moreover, understanding how different data structures work will influence the libraries and frameworks that you choose when writing software.

When you are done, you will be able to:

1. Explain and implement a Binary Tree.
2. Identify the three types of tree traversals: pre-order, in-order, and post-order.
3. Explain and implement a Binary Search Tree.you to become comfortable with implementing common data structures. This is important because questions about data structures are incredibly likely to be interview questions for software engineers from junior to senior levels. Moreover, understanding how different data structures work will influence the libraries and frameworks that you choose when writing software.

When you are done, you will be able to:

1. Explain and implement a Binary Tree.
2. Identify the three types of tree traversals: pre-order, in-order, and post-order.
3. Explain and implement a Binary Search Tree.

Graphs and Heaps

The objective of this lesson is for you to become comfortable with implementing common data structures. This is important because questions about data structures are incredibly likely to be interview questions for software engineers from junior to senior levels. Moreover,

understanding how different data structures work will influence the libraries and frameworks that you choose when writing software.

When you are done, you will be able to:

1. Explain and implement a Heap.
2. Explain and implement a Graph.table with implementing common data structures. This is important because questions about data structures are incredibly likely to be interview questions for software engineers from junior to senior levels. Moreover, understanding how different data structures work will influence the libraries and frameworks that you choose when writing software.

When you are done, you will be able to:

1. Explain and implement a Heap.
2. Explain and implement a Graph.

Network Models Objectives

The objective of this lesson is for you to get a basic understanding of network models. This lesson is relevant because any network-connected software that you write will implementations of these models to communicate with other computers. Questions about network models are popular interviewing topics, too.

When you finish, you should be able to

1. Describe the structure and function of network models from the perspective of a developer.a basic understanding of network models. This lesson is relevant because any network-connected software that you write will implementations of these models to communicate with other computers. Questions about network models are popular interviewing topics, too.

When you finish, you should be able to

1. Describe the structure and function of network models from the perspective of a developer.

Internet Protocol Suite Objectives

The objective of this lesson is for you to understand the different parts of the Internet Protocols. This lesson is relevant because knowledge of internet protocols is expected for all

Software Engineers that write code that connects to a network. Moreover, internet protocols are popular interviewing topics.

When you complete this content, you should be able to do the following.

1. Identify the correct fields of an IPv6 header.
2. Distinguish an IPv4 packet from an IPv6.
3. Describe the following subjects and how they relate to one another: IP Addresses, Domain Names, and DNS.
4. Identify use cases for the TCP and UDP protocols.
5. Describe the following subjects and how they relate to one another: MAC Address, IP Address, and a port.
6. Identify the fields of a TCP segment.
7. Describe how a TCP connection is negotiated.
8. Explaining the difference between network devices like a router and a switch.u to understand the different parts of the Internet Protocols. This lesson is relevant because knowledge of internet protocols is expected for all Software Engineers that write code that connects to a network. Moreover, internet protocols are popular interviewing topics.

When you complete this content, you should be able to do the following.

1. Identify the correct fields of an IPv6 header.
2. Distinguish an IPv4 packet from an IPv6.
3. Describe the following subjects and how they relate to one another: IP Addresses, Domain Names, and DNS.
4. Identify use cases for the TCP and UDP protocols.
5. Describe the following subjects and how they relate to one another: MAC Address, IP Address, and a port.
6. Identify the fields of a TCP segment.
7. Describe how a TCP connection is negotiated.
8. Explaining the difference between network devices like a router and a switch.

Network Tools

The objective of this lesson is for you to have a basic understanding of commonly-used network analysis utilities. This lesson is relevant because interacting with networks via different tools is essential for every software developer on the web. When you are done, you should be able to use `traceroute` to show routes between your computer and other computers. You should also be able to use `Wireshark` to show inspect network traffic.erstanding of

commonly-used network analysis utilities. This lesson is relevant because interacting with networks via different tools is essential for every software developer on the web. When you are done, you should be able to use `traceroute` to show routes between your computer and other computers. You should also be able to use `Wireshark` to show inspect network traffic.

WEEK-08 DAY-1
White-Boarding Tips and Tricks

White-Boarding Tips

Why do companies whiteboard?

Steps to solving white-boarding questions are identical to the steps necessary to solve real-world practical problems.

Companies want to see how you perform as a team member in a real team.

Things the interviewer will be looking for:

- Can you think algorithmically? Can you think about efficiency?
- Are you a good communicator? Are you someone I want to think through a new feature with?
- Can you code? Can you code neatly and correctly?

Correct steps

1. Clarify the problem & test I/O and edge cases
2. Formulate your approach(es)
3. Pseudocode best approach
4. Code it
5. Walk through an example input
6. Determine the Big O time and space complexity

Clarification

Before you jump into coding, ask questions. Don't rush, it's a problem-solving demo, not a speed-coding test.

- What are we coding?
- Are there any constraints on the input/output?
- What edge cases can we expect?

Test I/O

You should start with simple inputs and then slowly build up size or complexity of inputs. Look for patterns and things that remind you of problems you know how to solve.

If you draw a blank, use the following strategies to get started.

- Make up a sample input and compute it. Do this two or three times.
- Go through data structures in your head. Go through algorithms you know simultaneously. If you find one that works for this problem, run with it.
- If not, find a naive solution, any solution, to get started. Then you can optimize.
- Come up with a simpler version of the problem, solve it, then progressively add complexity.
- Think aloud about the likely bounds on efficiency for your solution.
    - This is an easy way to score points.
    - "What's sure is I'll have to iterate through all the points, so it's at least linear time."
    - "The problem is trivial when the set is sorted. So it can definitely be done in nLOGn time. Let's see if we can do better than n Log n."

Pseudocode

Never skip this step! Companies expect you to be able to pseudocode because it is the English version of your approach to the problem. You need to lay out your strategy step by step, so that when you code, you have something to refer back to. It's easy to be lost in the intricacies of implementation, so you as well as the interviewer need pseudocode to refer back to when you lose track of your thoughts.

- Be as detailed as possible
- Spend as much time as you want, within reason
- Make sure you can reason about implementation of every step

During the Problem

At this point, you should have spent at least 60% of your time. If you sprinted to coding, you most likely didn't spend enough time planning and will struggle here. If planned properly, this section should be the easiest part.

- The nice thing about white boards is you and the interviewer are facing the same direction.
- Convince yourself that you are solving the problem together. Say "we" instead of "I".
- Don't stop until they tell you to.

Your style definitely makes an impact. Here are tips to how to handle your bearing.

- Be confident; even if you don't know the answer, try to engage the problem, don't give up. If you keep telling an interviewer you don't know how to do something, they'll start to believe you.
- Talk through the problem; they want to see the process going on in your head. If you don't talk, the interviewer doesn't learn how you break-down and analyze a problem. If you can write and talk at the same time, great! If not, tell the interviewer what you're about to write, write it, and explain what you wrote.
- The interviewer may give you hints. They will ask questions to keep you on track. Don't be flustered or think you're failing; this is normal.
- If they ask you "does this work", take a moment to think. Walk through the steps; out loud is fine. If you say yes, say it like you believe it; interviewers don't like to think people are just praying they'll get the answer right.
- Listen to the interviewer. They are trying to help you. No one likes someone who doesn't listen.

Walk through an example input

- Initiate this step - don't wait to be prompted.
- Track all of your variables.
- Draw stacks if you're using a recursive method.
- Follow each iteration of your loops.
- Reason through your code from input to output.

Time and space complexity

- You should know how to do this - refer back to Big O reading if you need to.
- Remember time is expensive and space is cheap.
- If your approach is naive, attempt to optimize.
- If not, ask your interviewer if they'd like you to optimize.

Strategies

Keep a mental list of general strategies you can turn to. Here are a few:

1. Bucketizing with a hash: If the input set is bounded, try organizing it into a hash.
    - Ex: Sort an array of 100,000 integers that are all in the range 1-100
2. Dynamic programming, or "divide and conquer": Divide into smaller and smaller but equal subproblems.
    - Ex: See this solution for a Google interview question.
3. Look for useful mathematical properties.
    - Sometimes you have individual values when really what you care about is their sums.
    - Ex: For an array of integers 1 - 100 where all elements appear once except one that appears twice, find the repeat.
4. Amortized analysis: it's ok to do something memory- or space-intensive if you can prove that this cost comes with a greater payoff.
    - Ex: Implement a queue using two stacks.
5. Keep a stack or a queue on the side to track values as your algorithm goes through the problem.
6. Keep two pointers for the same iteration.
    - Ex: Reverse a string in place (ie. using no more memory space than the length of the string).
7. Perform an operation twice.
    - Ex: Reverse the word order of a string, but not the letters within the words.
8. Sort the input.
    - Ex: Finding anagrams.
9. Approach the problem from the other end.
    - Ex: see this solution for a Fog Creek interview question.
10. Use binary numbers instead of decimal numbers.
    - Ex: see the famous Bad King problem.
11. For efficiency, use binary search instead of incrementation. Esp. good for implementing math operators.
    - Ex: Implement division without using "/", in less than O(n) time.

Don't be Sly

If you don't understand the problem, ask for clarification. A well-formulated question is as impressive as a good answer. If you don't know something, don't make it up. Tell the interviewer you don't know and then try your best guess. Many interviewers will really like this. Same thing when you hit a snag. Don't try to cover things up and make it look like you were

on the right track. Explain to the interviewer why you think the current hypothesis actually won't work.

How to handle questions you've seen before.

What if you get a whiteboard problem you already know the solution to? Obviously the ethical thing is to tell your interviewer you've seen the problem before and you'll get points for that. Put on a sad face to show you were excited about solving a new problem.

Furthermore, you won't necessarily perform better on a question you've seen. You'll probably solve it faster, but speed is overrated. On the other hand, it's much harder to show your interviewer how you think when you're really just recalling a solution. Note that the easiest parts of a problem are also the hardest to remember. So even if you confound your interviewer, they'll be left wondering why you breezed through the hardest part of the problem while stumbling on the rest. That's a failure to demonstrate how you think, which is the real purpose of a whiteboard problem.

White Boarding Exercises

Work together to "white board" answers to the following problems. Your pair will present one of them at the end of the day.

You may not have real white boards on which to draw. If not, use the Zoom white board or just pen and paper. Talk through the answers. Determine if you are right.

When white boarding, you don't have to necessarily draw .. You can write code or pseudocode or draw circles and arrows. Just something to indicate that you understand the problem and what it would take to solve it.

Mirror image trees

Google asks in its interview process for you to draw an algorithm on a board that would return true if a binary tree is a mirror image of another binary tree.

Reverse a linked list

Amazon and Microsoft ask you to show an algorithm that will reverse a singly-linked list, that is, a list that is made of nodes between which there is a unidirectional association as in the following image.

The missing value

Amazon and Microsoft ask you to derive an algorithm that will inspect an array of of numbers that contains the values between 0 and the length of the list, inclusive, and find the missing value. For example, you may be given an array that of length 6 that contains

```
[0, 2, 3, 4, 5, 6]
```

It is your job to determine that the missing value from the array is 1.

Stack min

Google and Apple ask you to design a stack that, in addition to the `push` and `pop` functions, has a function `min` that returns the minimum element in the stack without removing it. All three functions `push`, `pop`, and `min` should operate in O(1) time.

Test a retractable ballpoint pen

Facebook asks you to write the tests cases for testing a ballpoint pen. What would you consider to be good tests for the pen? Try to be as exhaustive as possible.

OOParking Lot

Amazon and Microsoft ask you to specify the classes that it would take to write software to manage a paid parking lot. It should know where cars are parked, be able to identify the cars, know where the keys are hanging, how many cars are in the lot, what time the cars come and go, and how much it costs someone when they leave the parking lot based on the following schedule:

Time	Rate per hour
8pm - 6am	$3
6am - noon	$10
noon - 6pm	$8
6pm - 8pm	$6

WEEK-08 DAY-2
Binary Trees

Binary Trees and Binary Search Trees

The objective of this lesson is for you to become comfortable with implementing common data structures. This is important because questions about data structures are incredibly likely to be interview questions for software engineers from junior to senior levels. Moreover, understanding how different data structures work will influence the libraries and frameworks that you choose when writing software.

When you are done, you will be able to:

1. Explain and implement a Binary Tree.
2. Identify the three types of tree traversals: pre-order, in-order, and post-order.
3. Explain and implement a Binary Search Tree.you to become comfortable with implementing common data structures. This is important because questions about data structures are incredibly likely to be interview questions for software engineers from junior to senior levels. Moreover, understanding how different data structures work will influence the libraries and frameworks that you choose when writing software.

When you are done, you will be able to:

1. Explain and implement a Binary Tree.
2. Identify the three types of tree traversals: pre-order, in-order, and post-order.
3. Explain and implement a Binary Search Tree.

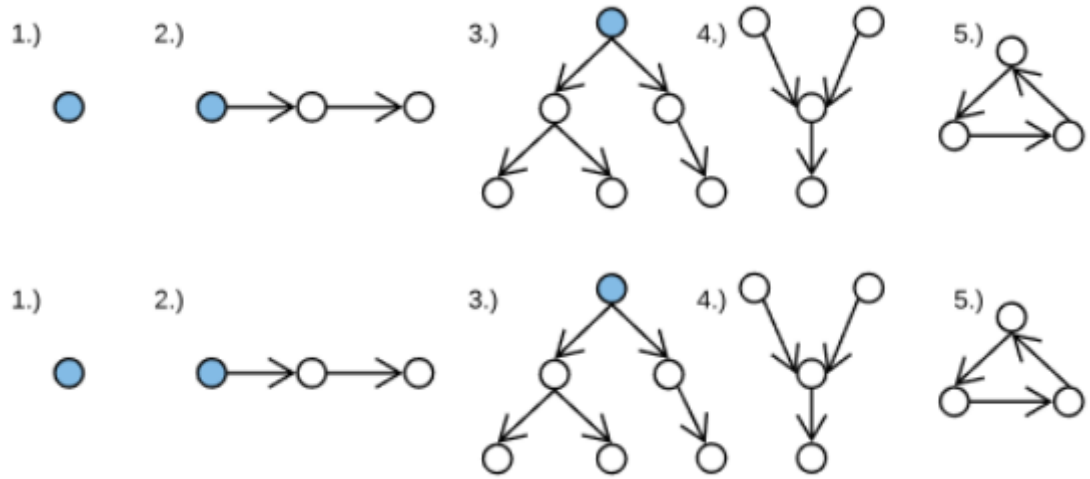
Binary Trees

Binary Trees are perhaps the most pervasive data structure in computer science. Let's take a moment to go over the basic characteristics of a Binary Tree before we explore algorithms that utilize this structure.

What is a Graph?

Before we define what a **Tree** is, we must first understand the definition of a **Graph**. A graph is a collection of **nodes** and any **edges** between those nodes. You've likely seen depictions of graphs before, they usually exist as circles (nodes) and arrows (edges) between those circles.

Below are few examples of graphs:



For now, you can ignore the blue coloring. Notice how the graphs above vary greatly in their structure. A graph is indeed a very broad, overarching category. In fact, **linked lists and trees are both considered subclasses of graphs**. We'll cover algorithms that operate on a general graph structure later, but for now we want to focus on what graphs are trees and what graphs are not. It's worth mentioning that a single node with no edges (image 1) is considered a graph. The empty graph (a graph with 0 nodes and 0 edges, not pictured 😐) is also still a graph. This line of thinking will help us later when we design graph algorithms.

What is a Tree?

A Tree is a Graph that does not contain any cycles.

- A cycle is is defined as a path through edges that begins and ends at the same node. This seems straightforward, but the definition becomes a bit muddled as Mathematicians and Computer Scientists use the term "tree" in slightly differentways.

- To a Mathematician, graphs 1, 2, 3, and 4 in the above image are trees.
- To a Computer Scientist, only graphs 1 ,2, and 3 are trees.

Well, at least both camps agree that graph 5 is most certainly not a tree! This is because of the obvious cycle that spans all three nodes. However, why is there disagreement over graph 4? The reason is this: In computer science, we use to the term "tree" to really refer to a "rooted tree." A "rooted tree" is a "tree" where there exists a special node from which every other node is accessible; we call this special node the "root". Think of the root as ultimate ancestor, the single node that all other nodes inherit from. Above we have colored all roots in blue. Like you'd probably suspect, in this course we'll subscribe to the Computer Scientist's interpretation. That is, we won't consider graph 4 a tree because there is no such root we can label.

You've probably heard the term "root" throughout your software engineering career: root directory, root user, etc.. All of these concepts branch† from the humble tree data structure!

What is a Binary Tree?

A **Binary Tree** is a **Tree** where nodes have **at most 2 children**. This means graphs 1, 2, and 3 are all Binary Trees. There exist ternary trees (at most 3 children) and n-ary trees (at most n children), but you'll likely encounter binary trees in your job hunt, so we'll focus on them in this course. Based on our final definition for a binary tree, here is some food for thought:

- an empty graph of 0 nodes and 0 edges is a binary tree
- a graph of 1 node and 0 edges is a binary tree
- a linked list is a binary tree

Take a moment to use the definitions we explored to verify that each of the three statements above is true. We bring up these three scenarios in particular because they are the simplest types of Binary Trees. We want to eventually build elegant algorithms and these simple scenarios will fuel our design.

Representing a Binary Tree with Node Instances

Let's explore a common way to represent binary trees using some object oriented design. A tree is a collection of nodes, so let's implement a `TreeNode` class. We'll use properties of `left` and `right` to reference the children of a `TreeNode`. That is, `left` and `right` will reference other `TreeNode`s:

```
class TreeNode {
  constructor(val) {
    this.val = val;
    this.left = null;
    this.right = null;
  }
}
```

Constructing a tree is a matter of creating the nodes and setting `left` and `right` however we please. For example:

```
let a = new TreeNode('a');
let b = new TreeNode('b');
let c = new TreeNode('c');
let d = new TreeNode('d');
let e = new TreeNode('e');
let f = new TreeNode('f');

a.left = b;
a.right = c;
b.left = d;
b.right = e;
c.right = f;
```

The visual representation of the tree is:

To simplify our diagrams, we'll omit the arrowheads on the edges. Moving forward you can assume that the top node is the root and the direction of edges points downward. In other words, node A is the Root. Node A can access node B through `a.left`, but Node B cannot access Node A.

We now have a data structure we can use to explore Binary Tree algorithms! Creating a tree in this way may be tedious and repetitive, however it allows us to decide exactly what nodes are connected and in what direction. This is will be useful as we account for edge cases in our design.

Basic Tree Terminology

- tree - graph with no cycles
- binary tree - tree where nodes have at most 2 nodes
- root - the ultimate parent, the single node of a tree that can access every other node through edges; by definition the root will not have a parent
- internal node - a node that has children
- leaf - a node that does not have any children
- path - a series of nodes that can be traveled through edges - for example A, B, E is a path through the above tree

† Pun Intended

Traversing trees

Unlike linked lists, arrays and other linear data structures, which are usually traversed in linear order from front to back or back to front, trees may be traversed in multiple ways. They may be traversed in depth-first or breadth-first order. There are three common ways to traverse them in depth-first order: in-order, pre-order and post-order.

Breadth-first search

Trees can also be traversed level-by-level, where you visit every node on a level before going to a lower level. This search is referred to as breadth-first search (BFS), as the search tree is broadened as much as possible on each depth before going to the next depth.

Depth-first searches

These searches are referred to as depth-first search (DFS), since the search tree is deepened as much as possible on each child before going to the next sibling.

Pre-order traversal

In the above image, the pre-order is noted by when the dotted line touches the red dot and yields F, B, A, D, C, E, G, I, H. The algorithm is as follows and is "pre-order" because you access the value of the node before recursively descending.

- Access the data of the current node
- Recursively visit the left sub tree
- Recursively visit the right sub tree

In-order traversal

In the above image, the in-order is noted by when the dotted line touches the yellow dot and yields A, B, C, D, E, F, G, H, I. The algorithm is as follows and is "in-order" because you access the value of the node after descending to the left but before descending to the right.

- Recursively visit the left sub tree
- Access the data of the current node
- Recursively visit the right sub tree

Post-order traversal

In the above image, the post-order is noted by when the dotted line touches the green dot and yields A, C, E, D, B, H, I, G, F. The algorithm is as follows and is "post-order" because you access the value of the node after descending to both branches.

- Recursively visit the left sub tree
- Recursively visit the right sub tree
- Access the data of the current node

Binary Search Trees

Now that we have a solid grasp of **Binary Trees**, let's add another constraint to the data structure. A Binary **Search** Tree (BST) has an additional criteria where:

- given any node of the tree, the values in the left subtree must all be strictly less than the given node's value.

- and the values in the right subtree must all be greater than or equal to the given node's value

BST Definition

We can also describe a BST using a recursive definition. A **Binary Tree** is a **Binary Search Tree** if:

- the left subtree contains values less than the root
- AND the right subtree contains values greater than or equal to the root
- AND the left subtree is a Binary Search Tree
- AND the right subtree is a Binary Search Tree

It's worth mentioning that the empty tree (a tree with 0 nodes) is indeed a BST (did someone say base case?).

Here are a few examples of BSTs:

Take a moment to verify that the above binary trees are BSTs. Note that image 2 has the same chain structure as a linked list. This will come into play later.

Below is an example of a binary tree that is **not** a search tree because a left child (35) is greater than it's parent (23):

A BST is a Sorted Data Structure

So what's the big deal with BSTs? Well, because of the properties of a BST, we can consider the tree as having an order to the values. That means the values are fully sorted! By looking at the three BST examples above, you are probably not convinced of things being sorted. This is because the ordering is encoded by an in-order traversal. Let's recall our previous `inOrderPrint` function:

```
function inOrderPrint(root) {
  if (!root) return;

  inOrderPrint(root.left);
  console.log(root.val);
  inOrderPrint(root.right);
}
```

If we run `inOrderPrint` on the three BSTs, we will get the following output:

```
BST 1: 42
BST 2: 4, 5, 6
BST 3: 1, 5, 7, 10, 16, 16
```

For each tree, we printed out values in increasing order! A binary search tree contains sorted data; this will come into play when we perform algorithms on this data structure.

Once you create a binary search tree class `BST`, you can call `insert` to build up the `BST` without worrying about breaking the search tree property. Here are two different trees built with the `BST` class that you'll write.

```
let tree1 = new BST();
tree1.insert(10);
tree1.insert(5);
tree1.insert(16);
tree1.insert(1);
tree1.insert(7);
tree1.insert(16);

let tree2 = new BST();
tree2.insert(1);
tree2.insert(5);
tree2.insert(7);
tree2.insert(10);
tree2.insert(16);
tree2.insert(16);
```

The insertions above will yield the following trees:

Are you cringing at `tree2`? You should be. Although we have the same values in both trees, they display drastically different structures because of the insertion order we used. This is why we have been referring to our `BST` implementation as **naive**. Both of these trees are Binary Search Trees, however not all BSTs are created equal. A worst case BST degenerates into a linked list. The "best" BSTs are **height balanced**, we'll explore this concept soon™.

A special traversal case

In a binary search tree, in-order traversal retrieves the keys in **ascending sorted order**. Please review the image that you saw before about tree traversal.

Note that the in-order sort represented by where the dotted line touches the yellow dots results in node visiting in the order A, B, C, D, E, F, G, H, I. In-order traversal always visits the nodes in sequential order in a binary search tree.

Binary Tree Project

This project contains a skeleton for you to implement a binary tree. This is a test-driven project. Run the tests and read the top-most error. If it's not clear what is failing, open the **test/test.js** file to figure out what the test is expecting. Make the top-most test pass.

Keep making the top-most test pass until all tests pass.

The algorithms for the traversal functions are in the files and are reproduced here.

```
procedure in order array (node)
  parameter node: a tree node

  if the tree node is null, return an empty array

  // get the array for visiting the left node of node
  // get the array for visiting the right node of node

  // return the left array concatenated with the node value
  //   concatenated with the right array
end procedure in order array
```

```
procedure post order array (node)
  parameter node: a tree node

  if the tree node is null, return an empty array

  // get the array for visiting the left node of node
  // get the array for visiting the right node of node

  // return the left array concatenated with the right array
  //   concatenated with the node value
end procedure post order array
```

Instructions

- Clone the project from https://github.com/appacademy-starters/data-structures-binary-tree-starter.
- `cd` into the project folder
- `npm install` to install dependencies in the project root directory
- `npm test` to run the specs
- You can view the test cases in `test/test.js`. Your job is to write code in
  - **lib/tree_node.js** to implement the `TreeNode` class
  - **lib/tree_order.js** to implement the `inOrderArray` and `postOrderArray` functions to traverse a tree

- **BONUS: lib/leet_code_105.js** as a scratch pad to work on the LeetCode.com problem at https://leetcode.com/problems/construct-binary-tree-from-preorder-and-inorder-traversal/

Binary Search Tree Project

This project contains a skeleton for you to implement a binary search tree. This is a test-driven project. Run the tests and read the top-most error. If it's not clear what is failing, open the **test/test.js** file to figure out what the test is expecting. Make the top-most test pass.

Keep making the top-most test pass until all tests pass.

Instructions

- Clone the project from https://github.com/appacademy-starters/data-structures-binary-search-tree-starter.
- `cd` into the project folder
- `npm install` to install dependencies in the project root directory
- `npm test` to run the specs
- You can view the test cases in `test/test.js`. Your job is to write code in
  - **lib/bst.js** to implement the `BST` class
  - **BONUS: lib/leet_code_108.js** as a scratch pad to work on the LeetCode.com problem at https://leetcode.com/problems/convert-sorted-array-to-binary-search-tree/
  - **BONUS: lib/leet_code_110.js** as a scratch pad to work on the LeetCode.com problem at https://leetcode.com/problems/balanced-binary-tree/

WEEK-08 DAY-3
Graphs

Graphs and Heaps

The objective of this lesson is for you to become comfortable with implementing common data structures. This is important because questions about data structures are incredibly likely to be interview questions for software engineers from junior to senior levels. Moreover,

understanding how different data structures work will influence the libraries and frameworks that you choose when writing software.

When you are done, you will be able to:

1. Explain and implement a Heap.
2. Explain and implement a Graph.table with implementing common data structures. This is important because questions about data structures are incredibly likely to be interview questions for software engineers from junior to senior levels. Moreover, understanding how different data structures work will influence the libraries and frameworks that you choose when writing software.

When you are done, you will be able to:

1. Explain and implement a Heap.
2. Explain and implement a Graph.

Graphs

It's time to generalize our knowledge! We've explored binary trees and the fundamental algorithms that accompany them. Naturally, we implemented these algorithms assuming the constraints of a binary tree. To review, these assumptions include the lack of cycles, a maximum of two children, and a single root node. However, what if we take away these constraints? How can we modify the algorithms to operate on general graphs?

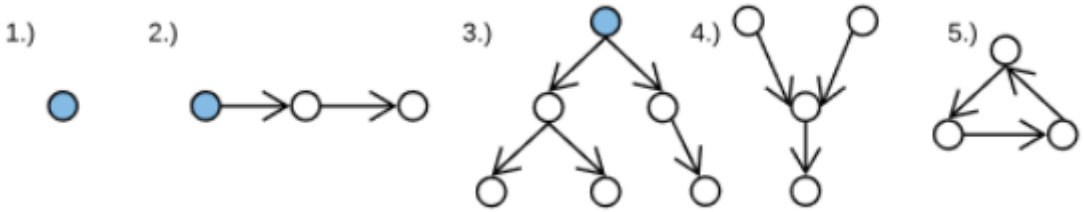
What is a Graph?

A **graph** is **any** collection of nodes and edges. In contrast to our previous trees, a graph is much more relaxed in it's structure. A graph may:

- lack a root node
- have cycles
- have any number edges leaving a node

In this section, we will draw heavily from our tree algorithms. The adjustments we will make to those algorithms will be motivated by these core differences.

Below are a few examples of graphs that don't agree with our CompSci definition of a binary tree:



Here are some highlights:

- `Graph 1` lacks a root. This means there is no single node that can access all other nodes in a path through edges. This is important because we previously referenced "entire" trees by referring to the ultimate root. We can no longer do that in a graph. If we provide just `T`, you can't access `U`. If we provide just `U`, you can't access `T`. If we provide just `V`, you can't access `T` or `U`.
- `Graph 2` has a cycle. This means there is no longer a parent-child relationship. Choose any node in `Graph 2`, its grandchild will also be its parent. Wait - what? From now on we'll have to use less specific language such as "`X` is a neighbor of `Y`." Perhaps even more deadly, imagine we ran a "simple" Depth-First traversal on this graph. We could get trapped in an infinite loop if we are not careful.
- `Graph 3` features nodes that have more than 2 edges. Anarchy!

Graph Implementations

There are many ways to represent a graph programmatically. Let's take a moment to explore each and describe the tradeoffs we make when choosing among them. We will use `Graph 3` from above as our candidate. Bear in mind that our graph is directed. For example, this means that `C` can access `D`, but `D` cannot access `C`.

GraphNode Class

This implementation is most similar to how we implemented binary trees. That is, we create a node class that maintains a value and an array of references to neighboring nodes. This easily solves the problem that a node can have any number of neighbors, no longer just a left and right.

```
class GraphNode {
    constructor(val) {
        this.val = val;
        this.neighbors = [];
    }
}

let a = new GraphNode('a');
let b = new GraphNode('b');
let c = new GraphNode('c');
```

```
let d = new GraphNode('d');
let e = new GraphNode('e');
let f = new GraphNode('f');
a.neighbors = [b, c, e];
c.neighbors = [b, d];
e.neighbors = [a];
f.neighbors = [e];
```

This implementation is great because it feels familiar to how we implemented trees. However, this implementation is clunky in that we have no easy way to refer to the entire graph. How can we pass this graph to a function? Recall that there is no root to act as the definite starting point.

Adjacency Matrix

This is the often the mathematician's preferred way of representing a graph. We use a 2D array to represent edges. We'll first map each node's value to an index. This means `A -> 0`, `B -> 1`, `C -> 2`, etc.. Below is the mapping for `Graph 3`:

From here, the row index will correspond to the source of an edge and the column index will correspond to its destination. A value of `true` will mean that there does exist an edge from source to destination.

```
let matrix = [
/*          A       B       C       D       E       F   */
/*A*/    [true,  true,   true,   false,  true,   false],
/*B*/    [false, true,   false,  false,  false,  false],
/*C*/    [false, true,   true,   true,   false,  false],
/*D*/    [false, false,  false,  true,   false,  false],
/*E*/    [true,  false,  false,  false,  true,   false],
/*F*/    [false, false,  false,  false,  true,   true]
];
```

A few things to note about using an adjacency matrix:

- when the edges have direction, `matrix[i][j]` may not be the same as `matrix[j][i]`
- it is common to say that a node is adjacent to itself, so ☑=== true for any `x`

An advantage of the matrix implementation is that it allows us to refer to the entire graph by simply referring to the 2D array. A huge disadvantage of using a matrix is the space required. To represent a graph of n nodes, we must allocate n^2 space for the 2D array. This is even more upsetting when there are few edges in graph. We will have to use n^2 space, even though the array would be sparse with only a few `true` elements.

Adjacency List

An adjacency list seeks to solve the shortcomings of the matrix implementation. We use an object where keys represent the node labels. The values associated with the keys will be an array containing all adjacent nodes:

```
let graph = {
    'a': ['b', 'c', 'e'],
    'b': [],
    'c': ['b', 'd'],
    'd': [],
    'e': ['a'],
    'f': ['e']
};
```

An adjacency list is easy to implement and allows us to refer to the entire graph by simply referencing the object. The space required for an adjacency list is the number of edges in the graph. Since there will be at most n^2 edges in a graph of n nodes, the adjacency list will use at most the same amount of space as the matrix. You'll find adjacency lists useful when attacking problems that are not explicitly about graphs. We'll elaborate more on this soon.

Graph Traversal

Let's explore our classic Depth-First, but for **graphs** this time! We'll be utilizing the `GraphNode` and `Adjacency List` implementations of the following graph:

Since we already discussed the differences between Depth-First and Breadth-First, we'll focus just on Depth-First here. We'll leave the Breadth-First exploration in the upcoming project.

Graph Traversal w/ GraphNode

Let's begin by assuming we have our candidate graph implemented using our `GraphNode` class:

```
class GraphNode {
    constructor(val) {
        this.val = val;
        this.neighbors = [];
    }
}

let a = new GraphNode('a');
```

```
let b = new GraphNode('b');
let c = new GraphNode('c');
let d = new GraphNode('d');
let e = new GraphNode('e');
let f = new GraphNode('f');
a.neighbors = [e, c, b];
c.neighbors = [b, d];
e.neighbors = [a];
f.neighbors = [e];
```

One thing we'll have to decide on is what node to begin our traversal. Depending on the structure of the graph, there may not be a suitable starting point. Remember that a graph may not have a "root". However in our candidate, `F` is like a root. It is the only valid choice because it is the only node that may access all other nodes through some path of edges. We admit, the choice of `F` is somewhat contrived and in a practical setting you may not have a nice starting point like this. We'll cover how to overcome this obstacle soon. For now we'll take `F`.

We want to build a recursive `depthFirstRecur` function that accepts a node and performs a Depth-First traversal through the graph. Let's begin with a baseline solution, although it is not yet complete to handle all graphs:

```
// broken
function depthFirstRecur(node) {
    console.log(node.val);

    node.neighbors.forEach(neighbor => {
        depthFirstRecur(neighbor);
    });
}

depthFirstRecur(f);
```

Can you see where this code goes wrong? It will get caught in an infinite cycle `f, e, a, e, a, e, a, e, ...` ! To fix this, simply store which nodes we have visited already. Whenever we hit a node that has previously been visited, then return early. We'll use JavaScript Sets to store `visited` because they allow for constant time lookup.

```
// using GraphNode representation

function depthFirstRecur(node, visited=new Set()) {
    // if this node has already been visited, then return early
    if (visited.has(node.val)) return;

    // otherwise it hasn't yet been visited,
    // so print it's val and mark it as visited.
    console.log(node.val);
```

```
    visited.add(node.val);

    // then explore each of its neighbors
    node.neighbors.forEach(neighbor => {
        depthFirstRecur(neighbor, visited);
    });
}

depthFirstRecur(f);
```

This code works well and will print the values in the order `f, e, a, c, b, d`. Note that this strategy only works if the values are guaranteed to be unique.

If you are averse to recursion (don't be), we can write an iterative version using the same principles:

```
function depthFirstIter(node) {
    let visited = new Set();
    let stack = [ node ];

    while (stack.length) {
        let node = stack.pop();

        // if this node has already been visited, then skip this node
        if (visited.has(node.val)) continue;

        // otherwise it hasn't yet been visited,
        // so print it's val and mark it as visited.
        console.log(node.val);
        visited.add(node.val);

        // then add its neighbors to the stack to be explored
        stack.push(...node.neighbors);
    }
}

depthFirstIter(f);
```

Graph Traversal w/ Adjacency List

Let's now assume our candidate graph in the form of an Adjacency List:

```
let graph = {
    'a': ['b', 'c', 'e'],
    'b': [],
    'c': ['b', 'd'],
    'd': [],
    'e': ['a'],
```

```
    'f': ['e']
};
```

Bear in mind that the nodes are just strings now, not `GraphNode`s. Other than that, the code shares many details from our previous implementations:

```
// using Adjacency List representation

function depthFirstRecur(node, graph, visited=new Set()) {
    if (visited.has(node)) return;

    console.log(node);
    visited.add(node);

    graph[node].forEach(neighbor => {
        depthFirstRecur(neighbor, graph, visited);
    });
}

depthFirstRecur('f', graph);
```

Cool! We print values in the order `f, e, a, b, c, d`. We'll leave the iterative version to you as an exercise for later.

Instead, let's draw our attention to a point from before: having to choose `f` as the starting point isn't dynamic enough to be impressive. Also, if we choose a poor initial node, some nodes may be unreachable. For example, choosing `a` as the starting point with a call to `depthFirstRecur('a', graph)` will only print `a, b, c, d, e`. We missed out on `f`. Bummer.

We can fix this. A big advantage of using an Adjacency List is that it contains the full graph! We can use a surrounding loop to allow our traversal to jump between disconnected regions of the graph. Refactoring our code:

```
function depthFirst(graph) {
    let visited = new Set();

    for (let node in graph) {
        _depthFirstRecur(node, graph, visited);
    }
}

function _depthFirstRecur(node, graph, visited) {
    if (visited.has(node)) return;

    console.log(node);
    visited.add(node);

    graph[node].forEach(neighbor => {
```

```
            _depthFirstRecur(neighbor, graph, visited);
        });
    }

    depthFirst(graph);
```

Notice that our main function `depthFirst` is iterative and accepts the entire Adjacency List as an Argo. Our helper `_depthFirstRecur` is recursive. `_depthFirstRecur` serves the same job as before, it will explore a full connected region in a graph. The main `depthFirst` method will allow us to "bridge" the gap between connection regions.

Still fuzzy? Imagine we had the following graph. Before you ask, these are not two separate graphs. This is a **single** graph that contains two connected components. Another term for a graph of this structure is a "Forest" because it contains multiple "Trees", ha:

It is easy to represent this graph using an Adjacency List. We can then pass the graph into our `depthFirst` from above:

```
let graph = {
    'h': ['i', 'j'],
    'i': [],
    'j': ['k'],
    'k': [],
    'l': ['m'],
    'm': []
}

depthFirst(graph);
// prints h, i, j, k, l, m
```

Here's the description for how `depthFirst` operates above. We enter `depthFirst` and the for loop begins on `h`. This means we enter our `_depthFirstRecur`, which will continue to explore the "local" region as far as possible. When this recursion ends, we would have explored the entire connected region of `h, i, j, k` (note that we add these nodes to visited as well). Our recursive call then returns to the main `depthFirst` function, where we continue the for loop. We iterate it until we hit an unvisited node (`l`) and then explore it's local region as far as possible using `_depthFirstRecur`, hitting the last node `m`.

Graph Project

This project contains a skeleton for you to implement some graph functionality. This is a test-driven project. Run the tests and read the top-most error. If it's not clear what is failing, open

the **test/test.js** file to figure out what the test is expecting. Make the top-most test pass.

Keep making the top-most test pass until all tests pass.

After the instructions, there is an in-depth explanation of the "friends of" problem.

Instructions

- Clone the project from https://github.com/appacademy-starters/data-structures-graph-starter.
- `cd` into the project folder
- `npm install` to install dependencies in the project root directory
- `npm test` to run the specs
- You can view the test cases in `test/test.js`. Your job is to write code in
  - **lib/breadth_first_search.js** to implement the `breadthFirstSearch` function for graphs
  - **lib/max_value.js** to implement the `maxValue` function for graphs
  - **lib/num_regions.js** to implement the `numRegions` function for graphs
  - **lib/friends-of.js** to implement `friendsOf` and `friendsOfRecursion` to find connected nodes in a graph less than or equal to a specified distance away from the start node (please see the explanation after these instructions)
  - **lib/leet_code_207.js** to implement the `canFinish` function located at https://leetcode.com/problems/course-schedule/

Friends of

The set of tests in **test/friends-of-spec.js** asks you to write a function named `friendsOf` that finds the total set of friends a specified distance away from a person. It will take as parameters

1. The adjacency list (which will always be an object with keys that always have arrays as values)
2. The name of the person whose friends you need to return
3. The distance away from the person that you'll use to collect the friends (this value will always be greater than or equal to 1)

The following table interprets the distance parameter:

Distance	Meaning
1	Immediate friends
2	Immediate friends and friends of friends
3	Immediate friends, friends of friends, and the friends of friends of friends

Distance	Meaning
n	All the people accessible n steps away from the indicated person

For example, say you had the following dependency graph.

```
const graph = {
  'carrie':  ['humza', 'jun'],
  'farrah':  ['humza'],
  'humza':   ['carrie', 'farrah', 'jun', 'silla'],
  'jun':     ['carrie', 'silla'],
  'ophelia': ['travis'],
  'silla':   ['humza', 'yervand'],
  'travis':  ['ophelia'],
  'yervand': ['silla'],
};
```

Then, the following table shows the expected results for the person **jun** at different distances.

Distance	List of people returned by `friendsOf`
1	carrie and silla
2	carrie, silla, humza, yervand
3	carrie, silla, humza, yervand, farrah
4	carrie, silla, humza, yervand, farrah

At distance 1, your traversal algorithm will find the friends of **jun**, carrie and silla and return them.

At distance 2, your traversal algorithm will find carrie and silla, then find their friends, humza and jun for carrie, and humza and yervand for silla. But, jun is the person that you started with, so you don't include them in the return value. Humza is both carrie's and silla's friend, but you only include that name once.

At a distance 3, you find carrie and silla, then humza and yervand. Then, looking at humza's friends, you see that humza knows carrie, farrah, hun, and silla. Only farrah is new, so that name will end up in the return value. When your traversal looks at yervand, it sees that silla is that person's friend, but is not a new value and does not end up getting added again to the return value.

At a distance four, you find carrie and silla, then humza and yervand, then farrah. From there, you look at farrah's friends which is just humza. You already have that name, so it doesn't get duplicated in the return value.

All distances 3 and greater will return the same list because you've exhausted all of the distinct names of people. You've captured the entire circle of friends.

The order in which you return the names is not important.

The tests also define edge cases that you also have to handle that are not in this explanation.

WEEK-08 DAY-4
Network Knowledge

Network Models Objectives

The objective of this lesson is for you to get a basic understanding of network models. This lesson is relevant because any network-connected software that you write will implementations of these models to communicate with other computers. Questions about network models are popular interviewing topics, too.

When you finish, you should be able to

1. Describe the structure and function of network models from the perspective of a developer.a basic understanding of network models. This lesson is relevant because any network-connected software that you write will implementations of these models to communicate with other computers. Questions about network models are popular interviewing topics, too.

When you finish, you should be able to

1. Describe the structure and function of network models from the perspective of a developer.

Internet Protocol Suite Objectives

The objective of this lesson is for you to understand the different parts of the Internet Protocols. This lesson is relevant because knowledge of internet protocols is expected for all Software Engineers that write code that connects to a network. Moreover, internet protocols are popular interviewing topics.

When you complete this content, you should be able to do the following.

1. Identify the correct fields of an IPv6 header.
2. Distinguish an IPv4 packet from an IPv6.
3. Describe the following subjects and how they relate to one another: IP Addresses, Domain Names, and DNS.
4. Identify use cases for the TCP and UDP protocols.
5. Describe the following subjects and how they relate to one another: MAC Address, IP Address, and a port.
6. Identify the fields of a TCP segment.
7. Describe how a TCP connection is negotiated.
8. Explaining the difference between network devices like a router and a switch.u to understand the different parts of the Internet Protocols. This lesson is relevant because knowledge of internet protocols is expected for all Software Engineers that write code that connects to a network. Moreover, internet protocols are popular interviewing topics.

When you complete this content, you should be able to do the following.

1. Identify the correct fields of an IPv6 header.
2. Distinguish an IPv4 packet from an IPv6.
3. Describe the following subjects and how they relate to one another: IP Addresses, Domain Names, and DNS.
4. Identify use cases for the TCP and UDP protocols.
5. Describe the following subjects and how they relate to one another: MAC Address, IP Address, and a port.
6. Identify the fields of a TCP segment.
7. Describe how a TCP connection is negotiated.
8. Explaining the difference between network devices like a router and a switch.

Network Tools

The objective of this lesson is for you to have a basic understanding of commonly-used network analysis utilities. This lesson is relevant because interacting with networks via different tools is essential for every software developer on the web. When you are done, you should be able to use `traceroute` to show routes between your computer and other computers. You should also be able to use `Wireshark` to show inspect network traffic.erstanding of commonly-used network analysis utilities. This lesson is relevant because interacting with networks via different tools is essential for every software developer on the web. When you are done, you should be able to use `traceroute` to show routes between your computer and other computers. You should also be able to use `Wireshark` to show inspect network traffic.

The OSI Network Model

One challenge with mental models is that everyone thinks about things differently! While we've discussed the TCP/IP reference model at length, we haven't introduced any others. Let's take a look at one other very well-known reference model for networks: the OSI Model.

We'll cover:

- the origin of the OSI reference model,
- each OSI layer and its properties,
- and how to choose between OSI & TCP/IP.

More layers, ~~more~~ fewer problems?

Around the same time computer scientists in the United States were hammering out the layers of the TCP/IP reference model, a similar discussion was happening a world away in the United Kingdom. Researchers in the UK decided that a clear reference model needed to be available to others worldwide, so they began working with the International Standards Organization (ISO). The ISO published a document called **The Basic Reference Model for Open Systems Interconnection** (or **ISO 7498**), including a seven layer reference model for networking, in the early 1980s.

The Open Systems Interconnection (OSI) reference model differs from the TCP/IP model by its focus on standardization. The TCP/IP model is mostly focused on practical networking concepts and isn't tightly tied to particular protocols (other than those for which it's named). The OSI model, however, has both conceptual layers **and** suggested protocols for each. This idea was well-intentioned: make these protocols the standard so that computer scientists have less to think about! This standardization could help prevent vendor lock-in as well, since all major vendors would (hopefully) follow the standards.

The layers of the OSI model

Here's an overview of the seven layers of the OSI model, along with some well-known protocols for each layer:

Let's dig into each layer, starting from the top.

Application

The OSI Application Layer includes information used by client-side software. Data transmitted on this layer will interact directly with applications, as the name suggests, and can be

displayed to the user with limited translation. **HTTP** is an example of a common Application Layer protocol.

Presentation

The OSI Presentation Layer is where data gets translated into a presentable format. This is often called the syntax layer since data is converted between machine-readable & human-readable syntax here as well. As a result, the Presentation Layer may include data compression, encryption, and character encoding. Many image formats, including **JPEG** and **GIF**, use well-known Presentation Layer protocols.

Session

The OSI Session Layer includes protocols responsible for authentication and data continuity. Session Layer protocols may authorize a client with the server or re-establish a dropped connection. An example protocol you may find on this layer is **RPC (Remote Procedure Call)**, a mechanism for one device to initiate a command on another.

Transport

Now we're on familiar turf! The OSI Transport Layer, like the layer of the same name in the TCP/IP reference model, utilizes transport protocols. Processes here are focused on data integrity and connectivity. Our old friends **TCP** and **UDP** are the two most-used transport protocols.

Network

The OSI Network Layer mirrors TCP/IP's Internet Layer. This layer manages connections between remote networks, transferring packets across intermediary devices. The best-known protocol at the Network Layer is **IP**.

Data Link

Protocols at the OSI Data Link Layer deal with connections directly from one machine's network interface to another. Frames targeting different MAC addresses are transferred here, and the Data Link Layer is primarily used by machines in local networks. The most recognizable protocol on this layer is **Ethernet**.

Physical

OSI's Physical Layer goes a little deeper than the TCP/IP reference model. Physical Layer protocols have to do with translating from raw electrical signals to bits & bytes of data. You may recognize **Wi-Fi** (technically known as **802.11**) and **DSL** as common Physical Layer protocols.

Which model do I use?

That's a lot of layers! It can be a little overwhelming to think of networks from the two complementary but differing perspectives of the TCP/IP and OSI reference models. Let's discuss when we might want to use each.

The OSI model is conceptual, meaning its practical uses are limited. We can see this when we look at protocols that cross layers. For example, HTTP primarily works on the OSI Application Layer, but includes the ability to manage character encoding, a Presentation Layer concern. Uh-oh! This makes OSI good for **understanding concepts**, but too restrictive for **building new protocols**.

The TCP/IP reference model, on the other hand, is almost purely practical. It was extracted from real, functional networks used by DARPA in the 1970s. Instead of concerns with minutiae like signal-to-data conversions, TCP/IP focuses on the core of networking: getting data from one place to another. For this reason, it's most often used when **building new systems** or **analyzing real networks**.

Of course, two popular models means most engineers will flip-flop between them! You'll often hear both models used in the same conversation. We'll discuss some techniques to differentiate between these two models in an upcoming lesson.

What we've learned

We've examined two ways of thinking about network design & functionality: first, the TCP/IP reference model, and now the OSI model. Next up, we'll compare these two models in greater detail.

After this lesson, you should feel comfortable:

- describing the layers of the OSI reference model,
- giving examples of protocols used at each layer,
- and explaining where one model may be more applicable than the other.

TCP/IP: Four Layers

We've investigated TCP/IP in great detail, and we've seen how broad a scope it covers. Now let's step back and think about the whole networking process. We're breaking the TCP/IP stack down and categorizing our protocols - are you ready?

We'll cover:

- The TCP/IP four-layer reference model,
- separation of concerns in networking,
- and data encapsulation.

A layered approach

Remember that when TCP/IP was first being crafted, researchers felt it was too large and separated the Transmission Control Protocol from the Internet Protocol. This separation was a boon; it made the protocols easier to implement individually and led to the Internet we know and love today!

We sometimes refer to this approach as separation of concerns. We divide up complex processes so that many connected concepts can work independently. This makes it easier to consider each concept in detail on its own and means each concept can grow at its own unique pace.

The developers of TCP/IP took this separation even farther in 1989 when they published RFC 1122. This spec, also titled "Requirements for Internet Hosts -- Communication Layers", provided a new way of thinking about the whole TCP/IP process. According to the RFC, we can separate the connection out into four distinct layers, or separate areas of interest. These are:

- Application
- Transport
- Internet
- and Link

We refer to this as a reference model: a high-level overview of a complex topic provided by an organization that manages it. The four-layer model presented in RFC 112 is often called the TCP/IP reference model, simply TCP/IP model, or even the Department of Defense (DoD) model, referring to the original research being done at DARPA.

Layers of the TCP/IP model

Here's a visual summary of the four layers of the TCP/IP reference model, along with some well-known protocols for each layer:

Let's look at what each layer of the reference model includes.

Application

The Application Layer includes protocols related to user-facing data. Some of the protocols that's we've discussed, like HTTP and FTP, operate in this layer. The TCP/IP model doesn't care what type of application data is used; **whatever** is transmitted from the Transport Layer is considered Application Layer data.

Transport

The Transport Layer includes (you guessed it) transport protocols! We've already discussed the two best-known: TCP and UDP. This layer focuses on connectivity between clients and servers, and relies on the lower layers to establish network connectivity.

Internet

The Internet Layer is where IP lives. Data is processed in packets on this layer, and routing is primarily handled with IP addresses. The Internet layer focuses mostly on connecting separate networks together.

Link

The Link Layer includes our lower-level communication standards. Link Layer protocols aren't concerned with the type of data being transported, but instead focus on getting data from one local network resource to another. We jump up to the Internet layer when dealing with resources on other networks.

> #### Fifth layer?
>
> Despite the RFC specifying four layers for the TCP/IP model, you may encounter resources detailing a five layer model for TCP/IP instead! The "fifth layer" is usually the Physical layer. This helps us separate electrical concepts like transmission across wires from data-oriented concepts like MAC addresses, but isn't an official reference model from the IETF. TCP/IP doesn't explicitly include any physical mediums, so thinking of this as a fifth layer can be helpful.

Translating layers to data

Layers provide a mental model we can use to think about how interactions across networks occur. It's important to remember that these are "best fit" models, though: they translate loosely to our actual data. Some protocols may cross layers, and some companies will adjust these models to fit their own internal implementations. Ultimately, these layers provide a form of shared communication between professionals. You can count on another engineer understanding what an "Application Layer issue" means, even if your own definitions differ slightly!

We often refer to encapsulation when describing how layers map to our data. This means higher layers are encapsulated, or wrapped, in lower layers. For example, a Transport Layer segment includes Application Layer data in its payload, and a Link Layer frame includes the whole stack! Here's an example:

As we'll see, alternative networking reference models may define layers as beginning/ending at different points in our data. However, the general idea is shared across models: lower layer data units include data for higher layers in their payloads.

What we've learned

When it comes to technical concepts, it's "reference models all the way down"! These new ways of thinking about topics we've already explored will help you communicate the concepts more clearly, and help you navigate problems that may be deeper than your own code.

You should feel confident:

- naming each of the four canonical TCP/IP reference model layers,
- explaining why these layers were defined,
- and relating these layers to the data being transferred on a network.

A Crash Course in Binary and Hexadecimal Notation

As we study networking we are going to be investigating a very low level of computing compared to the JavaScript programming we've been doing so far. We are moving closer to hardware, and so we will be exposed to some new numbers, formatted as binary or hexadecimal.

Binary

At the lowest level, a computer just speaks two values, `1` and `0`. These are usually represented by two different voltages inside an integrated circuit known as a CPU (Central Processing Unit). Everything that you do on a computer, writing and running your JavaScript code, watching videos online, chatting or posting on social media, or playing a video game, are at a fundamental level just 1's and 0's being processed by an integrated circuit.

So, what is binary? To understand this, we need to talk about bases.

Bases

What's a base? It turns out there's not just one number system. The number system human beings have used for thousands of years is **base 10**. This means we count using the Arabic digits 0 through 9. The reason we use base 10? It should be pretty obvious. The majority of human beings have ten fingers. So naturally we invented a counting system that used 10 digits. If we ever meet aliens from another planet that only have say, six digits, they might indeed use **base 6**.

As it turns out, the base that computers speak is **base 2**. This derives directly from the fact that transistors (which is what all integrated circuits are made of) have two voltage states. You can use **base 2** to perform mathematical calculations and because of this all computing at a fundamental level is base 2.

Base 2

So what does base 2 look like? Well if base 10 contains the digits 0 through 9, then base 2 contains the digits 0 through 1.

If you remember from your early math education, decimal (another word for base 10) numbers can be divided into *places*.

For example, given the number 42, the number breaks down into the following *places*.

Place	1000	100	10	1
Digit	0	0	4	2

```
(4 * 10) + (2 * 1) = 42
```

For binary, we instead have the following places, and the number 42 breaks down this way:

Place	128	64	32	16	8	4	2	1
Digit	0	0	1	0	1	0	1	0

```
(128 * 0) + (64 * 0) + (32 * 1) + (16 * 0) + (8 * 1) + (4 * 0) + (2 * 1) = 42
```

Or since we can simplify the zeros to 0 and the ones to 1, we can calculate this much more simply by just adding up the places that contain a 1.

```
32 + 8 + 2 = 42
```

This is a good shorthand way of calculating a binary number in your head as long as you memorize the bases.

Bits and Bytes

So inside computers we often call a single digit a **bit**. A bit can be either **on** (1) or **off** (0).

A sequence of 8 bits is known as a **byte**.

So our 42 example is a single byte since it contained 8 bits:

`00101010`

There are also some multiples of bytes computer science borrowed from the metric system, although with confusing results since the metric system is Base 10, while computing is Base 2.

Unit	Value
Kilobyte	1000 bytes
Megabyte	1000^2 bytes
Gigabyte	1000^3 bytes
Terabyte	1000^4 bytes
Petabyte	1000^5 bytes
Exabyte	1000^6 bytes
Zettabyte	1000^7 bytes
Yottabyte	1000^8 bytes

This system worked fine until computer storage got very large. Manufacturers of hard drives would use Base 10, while Operating Systems would often use Base 2. The discrepancy between something like the gigabyte in base 2 vs base 10 was very large.

> 1 Gigabyte in base 10 = 1,000,000,000 bytes 1 Gigabyte in base 2 = 1,073,741,824 bytes

That's 73.7 Megabytes of difference! With a Terabyte it got even worse.

> 1 Terabyte in base 10 = 1,000,000,000,000 bytes 1 Terabyte in base 2 = 1,099,511,627,776 bytes

For a whopping 99.5 Gigabytes of difference.

Something had to be done. So now we have two different sets of terminology one for Base 10 and one for Base 2.

Base 10	Abbr	Value	Base 2	Abbr	Value
Kilobyte	kB	1000 bytes	Kibibyte	KiB	1024 bytes
Megabyte	MB	1000^2 bytes	Mebibyte	MiB	1024^2 bytes
Gigabyte	GB	1000^3 bytes	Gibibyte	GiB	1024^3 bytes
Terabyte	TB	1000^4 bytes	Tebibyte	TiB	1024^4 bytes
Petabyte	PB	1000^5 bytes	Pibibyte	PiB	1024^5 bytes
Exabyte	EB	1000^6 bytes	Exbibyte	EiB	1024^6 bytes
Zettabyte	ZB	1000^7 bytes	Zebibyte	ZiB	1024^7 bytes
Yottabyte	YB	1000^8 bytes	Yobibyte	YiB	1024^8 bytes

Still to this day, you will hear people refer to the base 2 versions as Kilobyte or Megabyte. Often it's hard to determine what unit is being used when manufacturers advertise the size of hard drives or memory. Worse, Operating Systems often display inconsistent numbers throughout their many displays of how big disks or files are.

Another useful base

Another useful base in computing is **Base 16** also known as *hexadecimal*.

Why is this useful? This is use because hexadecimal can provide a shorter, more human-readable version of binary.

So if base 10 goes from the digits 0 through 9, what are we going to do? There aren't 16 digits...

The letters A through F are here to rescue us from this. The available digits for hexadecimal are *0 through F*, where `A` is `10` decimal and `F` is `15` decimal.

```
hexadecimal: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,  A,  B,  C,  D,  E,  F
decimal:     0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15
```

It's hard to think of letters as being numbers, but in hexadecimal it's perfectly normal.

So how does this help us write binary numbers in a shorter form? It's because there are 8 bits in a byte, which means one byte can be expressed as *two* hexadecimal digits.

Let's look at the *places* for hexadecimal for the decimal number 42. There's a sixteenth's place and a one's place in this example.

Place	**16**	**1**
Digit	2	A

This might seem confusing at first but just like our other base examples, this is:

```
(16 * 2) + (A * 1)
```

However, since `A` is really `10` in decimal this resolves to this in decimal.

```
(16 * 2) + (10 * 1) = 42
```

Using this you can see that the maximum value for a two digit hexadecimal number is `FF`, which is `255` in decimal and `11111111` in binary.

This happens to be the maximum value of one byte.

```
FF = 255 = 11111111 = 1 byte
```

So instead of typing out an entire string of 1s and 0s we can write bytes as a sequence of 2 digit hexadecimal numbers. We usually separate these numbers by spaces or some other delimiter to make it clear they are a sequence of numbers:

For instance this sequence of numbers in decimal:

```
4 8 15 16 23 42
```

is the following in binary:

```
00000100 00001000 00001111 00010000 00010111 00101010
```

But in Hexadecimal it's only this:

```
04 08 0F 10 17 2A
```

I hope you can see that this is a really compact and convenient way to represent binary numbers. The numbers are always the same length, which is good for data storage, and they can be easily translated back to binary. It might also help rescue you from Mars.

The `0x` Notation

Sometimes you will see hexadecimal numbers represented by prepending a `0x` to the front of them. So our 42 would be expressed like this:

```
0x2A
```

In JavaScript

In JavaScript, we can use `toString` on `Number` objects to convert different bases to decimal. We can supply a base as an argument to `toString`.

```
Number(42).toString(16) // 2a
Number(42).toString(2) // 101010
```

We can also use our old friend `parseInt` with an optional second argument to convert a binary or hexadecimal string to a decimal number.

```
parseInt('101010', 2) // 42
parseInt('2A', 16) // 42
```

In conclusion

Binary and hexadecimal are often used in computing because computers are fundamentally base 2. As you move closer to the hardware, you encounter these more and more often. In networking, you will see them used for IP Addresses and MAC Addresses. We'll learn more about that when we look next at how IP Networking works in the next section.

Internet Protocol

As we dive deeper into understanding how computers communicate, a common question keeps coming up: what is "the Internet", exactly? To answer this, let's discuss the Internet Protocol, also known as IP.

We'll cover:

- What IP is and why it matters to us,
- different protocol versions and when to use them,

- and how to identify IP data by its formatting.

History of IP

To understand where we are today, we need to look back to where we came from. Picture yourself in the United States in the late 1960s. The country is nearing the end of the "space race" and technology is booming. There are numerous technical teams and physical networks created as a result of recent research, but communication between them is limited. There's also a rise of different proprietary standards which are hampering growth: not every team can afford a separate terminal of each available type! How can we facilitate better collaboration with less investment required?

By 1974, two researchers working for DARPA think they have the answer. They propose something called the Transmission Control Program. It's a complex process that defines exactly how multiple networks can communicate with each other. This protocol stands out because it is:

- *fault tolerant*: data transmitted between networks can be cached and re-sent if it fails the first time.
- *end-to-end*: there are no single central systems that can take the whole network down; each host can send/receive to others.

These highlights are critical because DARPA is a military organization. They're looking for technology that could theoretically withstand a nuclear attack - and the Transmission Control Program fits the bill!

The great divide

It became quickly apparent that that Transmission Control Program was too dense. The process was complicated and involved many moving parts, and other engineers raised concerns that it should be extracted into separate parts. Soon, the Transmission Control Program was divided into two separate sections: Transmission Control Protocol (TCP), which was responsible for the fault-tolerance of joined networks, and Internet Protocol (IP), which was responsible for the end-to-end nature of joined networks.

The protocols we use today have been improved over time, but still carry those names and general purposes. It's amazing to think that modern social media, video gaming, and streaming content is all dependent on 50+ year old technology!

So what is the Internet, exactly?

The Internet can be loosely defined as "a series of internetworked systems". Here's a practical example of what we mean by that:

> Imagine every building in your town has a parking garage. Each garage is owned by a different company and uses custom entry sensors. These sensors prefer certain vehicle types: one garage for luxury sports cars only, another that will only allow motorcycles. You may need to park in the garage for the vehicle you own, then walk a lot. Otherwise, you'd have to own a different vehicle for each garage! Oh no! We'd consider this an "isolated" model: each garage works fine by itself, but patrons of one garage may not park elsewhere, and none of the garages are really meeting their full potential.
>
> Now imagine the garages remove their sensors. Suddenly you can park anywhere you'd like. Your motorcycle and your neighbor's school bus can exist harmoniously in any garage, and you can travel from garage to garage in any vehicle you'd like. We'd consider this an "internetworked" model: each garage's patrons may travel to any other garage and while the particular entrance/exit policies may differ, drivers can rest assured knowing they'll fit anywhere.

The Internet Protocol opened the door for this internetworked model in computing. Now a network in New York City using one vendor's computers could seamlessly communicate with a network in London from a different vendor! This connectivity led to the birth of the Internet, which is itself a series of interconnected networks sharing data.

Packet-Switching

IP data is transmitted in a format known as a packet. A packet uses a data format we've seen before: metadata in headers, and a body with content. The headers are used to get the packet to its destination, while the body contains the information we'd like to transfer.

We refer to IP's communication style as packet-switching. This is when a message is split up into separate "packets", delivered to a destination, and reassembled as appropriate. Remember that IP's primary responsibility as part of the Internet's "double threat", TCP/IP, is maintaining an end-to-end state. For this reason, IP isn't concerned about whether packets are received by the client in sequential order, and may sometimes even lose packets altogether while in transit!

> ### A crash course in bits & bytes
>
> Most of the protocols we'll cover measure their data in bits. We represent a bit as a single binary digit, either 1 or 0. Since data gets long, we have some larger units available as well! We'll sometimes describe data sizes in bytes. A byte is eight bits.
>
> For example, "01001" is five bits, and a piece of data that's 32 bits long could also be described as "4 bytes".

IP Versions

The Internet Protocol has evolved over time, but two versions stand out as the most used & important to us: version 4 and version 6. We often refer to these as IPvX, where X, is the version number.

IPv4

The best known version of the Internet Protocol is IPv4. This version was used when TCP/IP was finalized by DARPA in 1983, and it's still the most-used protocol version online.

An IPv4 packet's header consists of at least 13 fields, or sequences of binary bits. These fields start with a version identifier (`0100`, or "4" in binary), continue with 10 sequences that define things like the length of the header and protocol type contained in the body, and wrap up with source and destination addresses for the packet. IPv4 also includes an allowance for a 14th optional header field called "Options" that can contain extra metadata about the packet's content, but it's not often used. An IPv4 header without options will be 20 bytes (160 bits).

It's hard to visualize a header since it's essentially just a long string of 1s and 0s. Instead of trying to cram it all into one line to study, we can split it into specified widths and stack them. Here's a stacked diagram of the IPv4 header:

IPv4 Addresses

IPv4 addresses are composed of 4 octets, or 8-bit binary numbers. We usually represent them like this:

```
192.18.1.1
```

This is the same as `11000000.00010010.00000001.00000001` in binary notation, but that's a lot harder to read! IPv4 supports around 4 billion unique addresses.

IPv6

The "4 billion unique address" limit seemed almost infinite in the earliest days of the Internet, but today it's easy to see how we might use up that few addresses! Seeing this address exhaustion on the horizon, Internet researchers began concocting a new protocol version, one that would allow more addresses, in the mid 1990's. By 2017, the new protocol was an official standard: IPv6.

IPv6 uses a totally different packet header format than IPv4, though they share a few fields. It only uses 8 header fields, and supports optional "extension headers" that come after these 8 fields, as opposed to IPv4's large "Options" block.

The 8 fields IPv6 uses, in order, are:

- Version
  - `0110`, or "6" in binary notation
- Traffic Class
  - used to identify different types of packets, like video or phone data
- Flow Label
  - an experimental option used for adding packet sequencing into IP
- Payload Length
  - lets the reciever know how large the data in the packet will be
- Next Header
  - Usually identifies the protocol type of the packet's data, but may indicate the first extension header (if present)
- Hop Limit
  - A means of preventing packets from being passed around routers forever, this field will be decremented by 1 every time the packet passes through an intermediary (like a router)
- Source Address
  - Where the packet originated
- Destination Address
  - Where the packet is heading

These headers have a fixed length of 40 bytes (320 bits).

IPv6 Addresses

Notice that IPv6 packets have fewer headers, but are double the length of IPv4 headers! This is primarily due to IPv6 addressing, which allows **dramatically** more address space than IPv4. How many more addresses?

IPv4 supports **~4 billion ($4x10^9$)** addresses.

IPv6 supports **~350 undecillion ($3.5×10^{38}$)** addresses.

That's a billion times a billion more addresses! It's even more addresses than grains of sand in all the world's beaches & deserts ($7.5x10^{18}$, according to the University of Hawaii).

It handles this by quadrupling the number of bits in an address. IPv4 uses 32 bits, while IPv6 uses 128 bits. Remember that these bits are binary, so adding additional bits exponentially increases the number of permutations.

This new address space also required a new notation. Instead of the "four dotted decimal" notation of IPv4, IPv6 uses "eight colon-ed hexadecimal". Here's an example IPv6 address:

```
2600:6c5e:157f:d48c:138f:e0ba:6fa7:d859
```

The same address in binary is:

```
0010011000000000:0110110001011110:0001010101111111:1101010010001100:0001001110001111:1110000010111010:
```

It's easy to see how we added so many extra addresses! That said, IPv6 is much more difficult to read by humans. You can read some neat rules for making IPv6 addresses easier to read on Wikipedia.

Special addresses

Both popular versions of the Internet Protocol include space for some special addresses that you should quickly recognize. The main one you'll encounter is called the loopback address. This is the identifier for your current machine. You'll see it repeatedly while developing because you'll navigate your browser there to access your own servers! You may also hear this referred to as localhost.

In IPv4 the loopback address is `127.0.0.1`. In IPv6, the loopback is `::1`.

There's also an "all interfaces" address. This address is used to catch any incoming requests regardless of intended destination. It's only used by the local machine: you would never send a packet to the "all interfaces" address, but a server that is listening on that address would see all incoming packets.

For IPv4, the "all interfaces" address is `0.0.0.0`. For IPv6, it's simply `::`.

Remember that the loopback and "all interfaces" address are not interchangeable! This is a common misconception you may encounter in tutorials online, and might be a trick question during a technical interview. If you're ever asked to connect to `localhost`, make sure you use the loopback.

What we've learned

Whew! The Internet Protocol is a dense topic, but a little familiarity will go a long way when you're debugging server connections. After reading this lesson, you should be able to:

- provide a rough history of where the Internet Protocol came from,
- compare/contrast the two most popular IP versions (4 & 6),
- identify an IP packet's version by its headers,
- and name the localhost addresses for both versions.

Transport Protocols

Between HTTP and IP, we find an extra layer of information. We often refer to this as the transport layer of communication, and protocols used in it are referred to as transport protocols.

There are myriad of transport protocols available, but we're going to cover the two biggest: TCP and UDP. We'll dive into:

- why we need transport protocols,
- the differences between TCP and UDP,
- and where each protocol is used today.

What exactly are we transporting?

We've already briefly mentioned the Transmission Control Protocol (TCP) when discussing the history of the Internet Protocol (IP). Both TCP & IP made up the original Transmission Control Program developed at DARPA in the 1970s. We've dug deep into IP now, and we have some understanding of HTTP. Why do we need more protocols?

Let's provide a practical example. Think about the process of delivering a package. Floor pickers take your package from a warehouse onto the back of a truck, and a dispatcher sends that truck to your house. There's a place on your porch just waiting for that package. How, then, does your package make it across the very last leg of its journey? Whoops - we forgot the delivery person!

Transport protocols act as our "delivery person". IP is concerned with machine-to-machine communication, and HTTP is designed for application-to-application communication. Transport protocols bridge the gap and help our data cover the last mile between the network and software.

Ports

Like every other part of the internetworking process, transport protocols use their own unique form of addressing. We call these ports. Ports are virtual interfaces that allow a single device to host lots of different applications & services. By lots, we mean a whole bunch - there are 65536 separate ports available to each transport protocol!

Ports are represented by numbers: `port 80`, `port 51234`, etc. If we know both the IP address and port we'd like to connect to, we can use a special notation where both are joined by a colon:

```
192.168.1.1:8080
```

This would point to port `8080` on a network interface with an IP address of `192.168.1.1`. We refer to an IP address & port written together in this way as a socket.

TCP

The most common transport protocol used is TCP. TCP is a connection-oriented protocol, meaning it establishes a connection between two sockets. This connection acts as safeguard from other error-prone protocols underneath it, including IP and Ethernet. Pieces of data sent via TCP (referred to as segments) respect a strict order and verify when they have been received. This means that data can't be "lost" across a TCP connection: if a segment is received out of order, the receiver will ask the transmitter to re-send the missing segment. This behavior makes TCP a reliable protocol.

We'll dive deeper into exactly how TCP verifies data & forms connections in a future lesson. For now, remember that "TCP == reliability". Any time it's critical that data arrives ordered and in full, TCP's the way to go! You'll see TCP used as the underlying connection for HTTP, file transfers, and media streaming. In all of these cases, missing data would result in corrupt files and unreadable data.

Because of everything TCP offers us, it's a relatively "heavy" protocol to use. Messages may take a bit longer to transfer than they would via other protocols, but you can be confident that your message has been received the way you intended it. This inherent slowness means applications using TCP may buffer data, or wait until a certain amount has been received before passing it to the user. You've probably seen this happen on your favorite video sharing sites!

UDP

The User Datagram Protocol (UDP) arrived on the scene a few years after TCP. Scientists working with TCP found that they sometimes didn't need all the order and reliability that TCP provided, and they were willing to trade that for raw speed. UDP is connection-less and provides no verification for whether data is received. Because of this, we refer to it as an unreliable protocol.

Hold on, though! By "unreliable", we certainly don't mean "useless". UDP is used in lots of familiar places: real-time video sharing, voice-over-IP phone calls, and DNS all rely on UDP as their transport protocol of choice. These services prioritize speed over reliability, so it makes sense that they would forego TCP's additional lag. If some data is lost along the way, that's okay - for example, you might just see lower-quality video for a moment.

Unreliable systems are valuable outside the world of transport protocols as well! Consider the postal service: most letters are sent without any sort of delivery confirmation or guarantee of

arrival. It's up to the sender and/or recipient to manage expectations of when a letter ought to have arrived. This is similar to UDP. Data will be transmitted, and most will arrive, but if either side needs more reliability than that they will have to implement it themselves.

What we've learned

Transport protocols fill a gap in our current understanding of networks. They help us get data up from the network to our applications, and they give us a few options for fault-tolerance versus performance.

After reading this lesson, you should feel comfortable:

- explaining what transport protocols are and why we need them,
- comparing & contrasting TCP & UDP,
- and discussing use cases for each of these two major protocols.

Surveying Your Domain

We've covered how connected devices communicate with each other, but we're missing a key piece: where humans fit into the equation! After all, the Internet would be a much more boring place if we had to remember the IP address of every website we chose to visit.

Let's look into the Domain Name System, a method of translating long numeric identifiers into friendly, human-readable addresses. We'll cover:

- how the Domain Name System came to be,
- how a URL gets translated to an IP address,
- and the different types of information stored by the system.

What is DNS?

The Domain Name System (often just referred to as DNS) is a distributed approach to providing easily-understood names for internetworked devices. Practically, it's similar to a phone book: DNS allows us to look up a specific IP address by its domain.

In the early days of computer networking, connecting to another computer was a manual, complex process. A user would need very specific addresses to find the networked resource they were looking for, and those addresses were difficult to read/remember! A scientist named Elizabeth Feinler, working with ARPA in the early 1970's, saw a way to help. She started out with a simple text file listing computer names by their numeric addresses. This file grew in

size as more systems joined ARPANET, and Elizabeth expanded her operation from a text file to a whole organization dedicated to keeping an up-to-date list of hostnames & IP addresses.

By the 1980's, it was clear that one organization wasn't enough to manage the growing Internet. The Domain Name System was invented as a way to distribute the work to numerous organizations, lightening the load and allowing much more rapid growth. The first DNS name server was written in 1984, and the rest is history!

DNS is one of the most important parts of allowing the Internet to grow so rapidly. It's perfect for quick scaling because it is both simple (relying on specifically-formatted text files) and distributed (redundant across numerous servers).

Domains?

We've mentioned domains before today, but without much detail. Let's dive a little deeper into that term. A website's domain refers to the "friendly" name for the website's host, or the server providing the site's content. A domain differs from a URL in that the domain is only the server's identifier, not other application or protocol-related data in the URL.

Here's a breakdown of an average URL. We've highlighted the domain in green and labelled each part of the URL underneath:

A domain name can be split into a few parts:

- The top-level domain (TLD) is the last part of the domain, appearing just before the URL begins pointing at application routes (usually indicated with `/`'s) or query parameters (indicated with a `?` and `&`'s). The best known TLDs are `.com`, `.net`, and `.org`. TLDs are managed by special organizations that have demonstrated the ability to handle the immense workload involved, often known as domain registries. These registries may be government entities (for example, `.gov` is managed by the General Services Agency) or private companies that were awarded the privilege by ICANN.

- To the left of the TLD, separated by a dot, is the second-level domain. You'll often hear the TLD & second-level domain lumped together as "the domain". This is the name most people associate with the website. Through domain registrars, consumers can purchase second-level domains. The registrar maintains a listing of each purchase.

- Some websites will have additional domains to the left of the second-level domain. These can be referred to by their formal names (third-level domain, fourth-level domain, etc.) but are often informally referred to as subdomains. The best-known subdomain is `www`, though this is less-used on newer sites. Subdomains can usually be freely created by the consumer.

How The Magic Happens

DNS does one thing really well: identifying connected devices by friendly names. How exactly does this work?

It all goes back to the magic word: domain. Each individual domain is represented by a set of name servers, which store information about the domain's registered subdomains. Name servers will direct a client where they need to go - even if that's another name server! We refer to this process of working out which name server we need as resolution. Eventually, we'll reach a name server that can tell us the specific IP address for the full domain. We refer to this as the authoritative name server for our domain. It has the final say!

When trying to resolve a domain name, we start from the rightmost part (the TLD) and work our way to the left. We'll stop once we've reached an authoritative server that can give a direct address for the domain we're seeking. Intermediate servers should be able to point us to the most-relevant name server to continue our search (usually, the next domain to the left). We can think of this as a conversation between the client and the available name servers for our domain, each one moving us closer to our goal.

Here's a practical example of how DNS is used to discover the authoritative name server for the fictional URL `https://students.appacademy.io`:

Looking for something a little more whimsical? DNSimple has a fantastic webcomic detailing the journey of a domain resolver. Check it out!

DNS Records

We can see how DNS works, but what does it actually look like? It's not much different than it was at the very beginning! Each name server maintains a zone file: a text file containing host names, IP addresses, and resource types. Here's an example of a simple zone file:

```
$TTL 299
my-site.com.    IN  SOA     ns1.cloudflare.com. dns.cloudflare.com. 2032032092 10000 24
my-site.com.    IN  NS      ns1.my-site.com.
my-site.com.    IN  NS      ns2.my-site.com.
my-site.com.    IN  A       104.28.31.159
my-site.com.    IN  A       104.28.30.159
my-site.com.    IN  AAAA    2606:4700:30::681c:1f9f
my-site.com.    IN  AAAA    2606:4700:30::681c:1e9f
www             IN  CNAME   my-site.com.
ns1             IN  A       104.28.31.150
ns2             IN  A       104.28.30.150
my-site.com.    IN  MX      10 mail.google.com.
```

Each line in a zone file includes the affected domain, type of record on that line, and the data for that record. Let's discuss some of the most common DNS record types in the order we see them above:

`SOA`

The SOA record represents the **S**tart **O**f **A**uthority. This record lets us know which name server is the master, or primary authority, for the domain we're querying. The SOA record is the minimum requirement in a zone file - every name server will return this record, if nothing else!

`NS`

NS records point to name servers for the zone. Most zones will have at least two NS records for redundancy. Remember that one of DNS's strengths is that it's distributed. If one name server loses power or becomes disconnected, we don't lose access to the zone.

`A` / `AAAA`

A records are the most important DNS records present. They map a resource directly to an IP address. This is the core of what DNS is for: connecting the domain directly to a machine. `A` records are used for IPv4 addresses, while `AAAA` records perform the same function for IPv6.

`CNAME`

The CNAME record acts as an alias, linking one domain to another. In our example above, we're saying that `www.my-site.com` should point at the same resource as `my-site.com`. Notice that the `www` doesn't have a `.` after it. This means it's a relative reference, and the additional parts of the domain for this zone (`my-site.com.`) are implied. When a domain in zone file ends in a `.` we can treat it as an absolute reference with no unwritten subdomains.

`MX`

DNS: It's not **just** for websites! MX records, short for **M**ail E**x**changer, are used by e-mail clients to direct messages to the appropriate mail servers. These records let you send messages to "friend@gmail.com" instead of having to remember "friend@123.45.67.89"!

Metadata

There's one piece we've overlooked in our example zone file above: the first line. `$TTL 299` refers to the Time to Live (TTL) for our records. This is a measure of how long a record should be cached by a DNS name server.

We cache DNS queries because reading from a file can be slow! When a query comes in for a particular domain, the name server will cache the result in memory so that subsequent

requests are much faster. However, this in-memory copy won't be updated if the zone file changes - yikes! The TTL lets us set how often a cached record should be discarded and read from the zone file again. This is especially important if we are pointing at a service where the IP might change frequently, like a local development environment or shared hosting service.

In our example, we've set the TTL for all records in the zone file to 299 seconds. This means that if your current web host goes offline and you have to point your domain at a new server, the downtime won't last more than approximately five minutes. This **also** means that you'll be re-checking your zone file for that domain at least once every five minutes. If you're confident in your hosting and aren't making infrastructure changes, longer TTLs can result in slightly increased performance.

What we've learned

The Domain Name System is a great example of a simple process (linking names to locations) evolving over time to support greater and greater needs. It's frightening to think of how difficult navigating around the Internet would be if we didn't have DNS to make websites easily accessible!

Before we move on, here's a quick tip: **DNS questions are popular fodder for technical interviews.** You may be asked to define a particular record type or to walk through a rough outline of what happens when you type a URL into your browser and click "Go". Try thinking through this process with your new knowledge!

After reading this lesson, you should have a better understanding of:

- the history and intent of DNS,
- how to read & break down a domain name,
- and what's involved in translating domains into IP addresses.

Networking Hardware: Getting Physical

We've discussed a lot of data- and internal-communication protocols, but what supports these? Let's examine some of the most important hardware you'll see while examining computer networks!

We'll cover:

- networking hardware devices and how they differ,
- use cases for each type of device,
- and specialized cases with integrated devices

Three levels of control

Network protocols mean very little if we don't have a physical way of connecting computers together! Whether it's via copper cables, fiber optics, or wireless networks, we need ways of managing communications to put those protocols into action. A quick search for "networking hardware" will yield a slew of results, but don't get overwhelmed! We can boil many of these devices down to three types: hubs, switches, and routers.

Hubs: keeping it simple

A hub is the simplest networking device you're likely to find in service. It performs no network management and might be better known as a "signal splitter". When a hub receives data, it duplicates that data and broadcasts it to all connected devices. That's it!

Hubs tend to be cheap and are often found in older networks. They are usually small metal boxes with a handful of physical connectors. You can get hubs with lots of connectors, but they're usually a little smaller - think 5 or 10 instead of 30 or 40. This is due to the natural limitations a hub possesses.

> Heads up! We'll refer to the physical sockets that cables plug into as connectors, but you'll often hear them called ports instead. This can get **very** confusing, so be sure you're clear about the difference between virtual ports used by transport protocols and physical ports used by hardware. When in doubt, use a clearer term, like "connector" or "jack".

For one, a hub can't do any sort of filtering. This means every single data packet is sent to every single device, all the time. This creates a lot of unnecessary load on the network. Imagine if every time you called a friend, all of your other friends' phones rang too. Yikes! Additionally, hubs share bandwidth across devices, so heavy traffic can result in lower speeds. We'll sometimes see this problem on overloaded networks with other devices, but on a hub it's guaranteed.

Hubs were a helpful and necessary piece of hardware for a long time, but today there's little reason to use them. They may still be slightly cheaper than a switch, but the limitations outweigh most cost concerns. The best use for a hub now is as a temporary replacement while replacing a broken device.

Switches: traffic control

A step up from the hub, we find a network switch. Switches are "intelligent" hubs: they track devices connected to them, help manage network load, and can manage separate internal

networks with ease! The biggest thing that separates a switch from a hub is the MAC address table.

A network switch maintains an internal address book containing the MAC addresses of the devices connected to it. Remember that data frames contain both a source and destination MAC address? This is how the switch stays up to date! It uses this data to perform one of three actions with each piece of data it receives:

- flood: When a destination address is unknown, the switch will flood received data out to all connected devices **except where the data came from**. When the intended recipient responds, the switch will update its MAC address table accordingly. This is how switches learn about connected devices, and it's significantly more efficient than a hub's behavior of flooding all the time!

- forward: When a switch already has the destination MAC address in its internal table, it can send data directly to that device. This is called forwarding the data. No other devices connected to the switch are made aware of this data.

- filter: Sometimes a switch will receive data on the same connector the data is destined for. In these cases, the switch will filter, or drop, the data entirely. This can be a little confusing to think about! If data arrives on the same connector it would later be sent out of, then we can assume the data was handled by some other part of our network, and the receiving switch can't do anything to help. Remember that this is very specifically related to the physical connector the data is received on: a switch will never act on data that comes in and goes out the same connector.

Switches often look just like hubs, but come in a much larger range of sizes. They can be chained together to cover large networks, or a 5-connector switch might be used for a small home network. Switches have improved on hubs' limitations: they don't share bandwidth, so you'll see less impact on speed through a switch than you might through a hub.

If you're building a home or single-location office network today, switches are your friend. They provide lots of simple management functionality for not much cash, and they're easy to keep around.

Routers: thinking globally

Here's a thought experiment: let's think about what a switch for a national network might look like. Since switches can be chained together, we wouldn't need thousands of connectors - but we would need lots of memory! The MAC address table would need to hold entries for every computer in the country. No way!

Instead of trying to solve this problem with switches, we have a higher-level device: the router. Routers connect separate networks with each other. Instead of identifying devices via MAC address, they use IP addresses to make decisions about data.

A router, like a switch, maintains an internal table of addresses. This routing table is used to pass received data on through the network. Data may move on to another router, or the router may recognize the data and pass it to an internal switch.

Routers also participate in an important process called NAT, short for Network Address Translation. NAT helps minimize IP address overload by giving the router a single IP address to use for all external communication. The router then uses IP ports to map incoming data to internal device IP addresses in its routing table. Imagine living in an apartment complex with a mail office. The postal service could bring packages with your apartment number to the front desk, and the mail officer would dole those packages out, but the sender would never actually have to know your name or exactly where your apartment is located! NAT provides a tiny bit of security and allows significantly more computers to coexist on the Internet simultaneously.

Because of the extra processing power required to handle large routing tables & filtering, routers tend to cost substantially more than switches. However, most networks only need one router! We sometimes describe routers as being our gateway to the Internet.

Physically, routers come in all shapes and sizes. They only need two connectors (one incoming, one outgoing), but they often come as part of an integrated device with multiple connectors and functions.

A practical example of network hardware

That's enough theory! Let's think through a practical example of each of the pieces of network hardware we've discussed.

Imagine you need to get a message to a friend across the room. There are lots of people between you and your friend: how might you communicate?

One way might be to shout. If you shout your message to the room, the rest of the people in the room could repeat it to make sure it's heard! This is going to result in some ringing ears, but your friend will definitely hear the message. This is how hubs work: broadcast to everyone, no matter who's listening.

An alternative might be to whisper. You could pass a message to the person right next to you. Of course, they won't know everyone in the room, so they'll have to ask the people close to them to pass it along. It may take a bit to get to your friend the first time, but any responses will be lightning-fast since everyone now knows who to talk to. This is a switched model: flood the message once to learn how to get to the destination, then use what we've learned to forward & filter appropriately.

Finally, imagine your friend has left the room. Uh oh! We could still use the shouting or whispering approach, but none of that matters since we now need to find the correct room your friend is in. To do this, we'd need people familiar with each room. We could pass our message to those gatekeepers, who could pass the message for us from room to room until it reaches the correct one. At that point, the process will reverse in the new room: the gatekeeper will whisper or shout, the room will respond accordingly, and responses will come back via the gatekeepers. The gatekeepers are acting as our routers here: they pass our messages between rooms, but don't necessarily care how the room communicates internally.

Notice in this example that each type of communication has a purpose, and all of them work together. We're simplifying things, but it's easy to concoct an example where we need all three types, or where we only need one. This is true of hardware was well. You should choose the correct devices for the network you're building. Not thinking through your needs may result in a steep cost, both in terms of performance and in terms of money!

Integrated devices

It's easiest to think of these three classes of hardware as separate, distinct devices. However, this won't always be the case. Let's discuss some situations where these devices exist in unfamiliar packages.

When you set up Internet at a new home, you usually get a modem from your ISP. Years ago, this modem was dead simple: one inbound connection for your phone line or cable, and one outbound connection for your home PC. Today, however, our homes have multiple devices! Consumers became increasingly frustrated with getting a modem from one company but still needing a router and/or switch to connect all their computers.

In response, ISPs upped their game by integrating extra devices in their "modems". The average home gateway today includes:

- a modem to translate the physical signal from the cable/phone line,
- a router to manage your internal and external IP addresses,
- a wireless antenna for wi-fi connections,
- and a 5-connector switch for wired connections.

Woah! That's a lot of hardware in one device. This behavior has blurred the lines between types of hardware and made communication about networked devices more difficult. Your ISP might call that single device a "modem", a "router", a "gateway", or an "access point". All of these are true!

Before making decisions about a network you're investigating, make sure you understand what each device is doing. Does that router include a switch? Is the modem a simple modem or does it include a hub as well? As always, make sure you're using the right tool for the job.

What we've learned

There are lots of different types of network hardware out there, but most of them fall into three separate categories. Hopefully, the next time you see inside a messy server closet, you'll be curious about which parts you can identify!

After this reading, you should be comfortable with:

- the difference between hubs, switches, and routers,
- identifying each type of device,
- switched internal networking,
- and the basic concept of routing on interconnected networks.

TCP Connections

The Transmission Control Protocol (TCP) is the backbone of the Internet. The majority of services we use rely on it, and a deeper understanding of it will help you navigate those services with ease.

Let's deep-dive into TCP! We'll cover:

- TCP segments what information they carry,
- "control bits" that manage the TCP lifecycle,
- and the process of forming & discarding connections.

Segments

Just like the Internet Protocol (IP) uses packets, TCP uses data units called segments. Segments are formed from application data: TCP receives this data, breaks it down into transmittable units, and attaches a header to each unit. This header contains everything we need to ensure a reliable connection is established.

Segment Header Fields

As with IP, TCP header fields are critical to understand. They help satisfy two needs of the protocol: **reliable data transfer** and **consistent connections**.

Here's an overview of each field in the order they appear:

- **Source / Destination Port:** The first two fields indicate which port the request originated on and which port it's directed to. This will be used, along with the IP address in the IP wrapper containing the segment, to determine which sockets to use for the TCP connection.

- **Sequence Number:** This number is used to establish the correct ordering of data. At the start of a connection, TCP sets an Initial Sequence Number (ISN) that's sufficiently large enough to avoid conflicts. Each byte of data transferred is then counted, beginning at the ISN, and used to calculate the sequence number for the segment. Sequence numbers go hand-in-hand with the ...

- **Acknowledgement Number:** This number lets the sender know which sequence is expected next. Acknowledgement numbers are cumulative, which is how TCP maintains proper ordering of segments. The receiver will send an acknowledgement number that's one higher than the last sequence number plus the length of the last data received. For example: if the last sequence number received was `10`, and the accompanying data was 4 bits long, the response will include an acknowledgement number of `15` - meaning "I'm ready for sequence number 15!".

- **Data Offset:** Defines how long the segment header is. This lets us know if there are options later on in the header or not.

- **Reserved:** A short range of three bits, held over for later use. These will always be `0`.

- **Control Flags:** These nine bits drive the TCP connection process. We'll break them down in more detail soon.

- **Window Size:** This field is used to let the sender know how much data the receiver can accept. This helps maintain the reliability of a connection: if a receiver is getting overloaded, they can lower the window size as a way of saying "slow down!". If a slow connection can move faster, a larger window size is a way of saying "bring it on!".

- **Checksum:** The checksum is an error-checking mechanism. Checksums are used to check the validity of a particular segment, not the whole series of segment (as with sequence/acknowledgement). The TCP client can use the checksum to ensure the segment has been received correctly. If it doesn't match expectations, the segment will be discarded & ignored.

- **Urgent Pointer:** TCP allows for data to be marked as urgent. This means it should be processed right away, regardless of sequence, interrupting any other transfer in process. This is useful when trying to terminate a long transfer, as we'd like to kill the connection without waiting for it to complete. This field indicates how much urgent data to expect, if there is any.

- **Options:** Like most other protocols, the TCP segment header includes a range for options at its end. There may be no options, or there may be a handful! We can verify whether there are options be comparing the length of the header so far to the data offset field. TCP header options are mostly used for flow control and may even include padding to fill out the expected length of the header with empty data.

Immediately after any options (or after the urgent pointer, if no options are given), the encapsulated data from our application begins.

TCP Connection Lifecycle

Remember that TCP is a connection-oriented protocol. This means that it establishes a long-running line of communication between two points, instead of just shouting into the Internet void like UDP. Establishing this connection involves a series of predictable steps, each with specific names. Most of these steps are driven by the control flags in the segment headers.

Control Flag Options

The segment header has 12 bits reserved for control flags. The first three of these are currently unused and will always be zero, and the next three are all used by congestion-management extensions to the protocol. The control flags that most concern us are the final six bits, each known by a short three-letter name. Let's review them in order:

- **URG:** The "**urg**ent" bit. Lets us know if this segment contains urgent data.
- **ACK:** The "**ack**nowledgement" bit. Setting this bit means a message has been received successfully.
- **PSH:** The "**p**u**sh**" bit. This bit is used to indicate that buffered data should be passed on to the connected application.
- **RST:** The "**rese**t" bit. This bit means we should reset the connection. Receivers will send an RST segment when they receive unexpected data, either to a port that's not listening or dramatically out-of-sequence.
- **SYN:** The "**syn**chronize" bit. This flag is set on the the first segment from each side of the connection, and should include an ISN for the socket to begin sequencing from.
- **FIN:** The "**fin**ished" bit. This bit lets each side know that transmission is done and the connection may be closed.

> Note that we mention all twelve bits here as "control bits" even though the first three are unassigned. While they are currently unused, protocols evolve frequently! The TCP

> specification lists "six reserved bits, six control bits", but newer specs have claimed some of those. For this reason, you should think of the remaining three reserved bits as "control flags still under development".

Getting to know each other: the three-way handshake

TCP connections begin with a process called a three-way-handshake, also sometimes referred to as SYN-SYN-ACK. This name comes from the three interactions before the connection is officially "open":

- The client notifies the server that data is incoming with a `SYN` segment.
- The server acknowledges that data and sends its own segment, including both an `ACK` and its own `SYN`.
- The client `ACK`nowledges with another segment to the server. Now both sides are ready to go!

Notice that we can send both a `SYN` and `ACK` on a single segment. This is called piggybacking and saves us a ton of requests! The three-way handshake would become a four-way handshake if we had to send the `SYN` and `ACK` separately.

TCP connections are often visualized using ladder diagrams, also sometimes called timing diagrams. Let's take a look at one for the three-way handshake:

This diagram should be read top-to-bottom. Each arrow represents a single segment being passed between hosts. You can see how the client initiates the request, but the server mirrors the process. This ensures both sides are ready to work: if the client's `SYN` request fell on deaf ears, we would expect an `RST` segment back.

You can also see how the sequence number and acknowledgement number are incremented for each segment. Since these initial segments contain no data, each only increments by one. During data transfer, the numbers will increment based on the length of data received so far.

TCP maintains timers for most behaviors to ensure that connections don't hang empty forever. This is one reason time-based diagrams are so helpful: by adjusting the angle of the arrows between the client & server, we can indicate a faster or slower connection. This can be helpful for visualizing connections on a granular level. For example, here's a simplified diagram of the same three-way handshake with a slow server response:

Data transmission & error handling

Once the connection has been established, we're off to the races! The client will send data segments over and the server will respond with `ACK` segments.

Remember that the sequence number in each data segment indicates where our data starts, and the corresponding acknowledgement number should be the next position after our data ends. For example, a data segment with a sequence number of `4` and data of length `3` will be `ACK`'ed with an acknowledgement number of `7`: our data included `4`, `5`, and `6` in the sequence, and the server is letting us know that it's ready for data beginning at sequence number `7`.

Here's a fun way of visualizing this concept via text messages between the client & server:

Note that there won't be any more `SYN` segments unless the connection terminates unexpectedly: we only send segments with the `SYN` flag enabled when initializing a connection.

The acknowledgement number is important for keeping the TCP connection reliable. It will only increment when a segment is successfully received, so an `ACK` response to the client with a lower acknowledgement number than the client's current sequence number means a segment was missed and must be retransmitted.

A diagram is worth a thousand words:

This is a **major** part of TCP, and one of the reasons it's both reliable and slow: data may need to be retransmitted frequently, but the end result is always a full & correct payload on the server.

Saying goodbye: closing the connection

Once we've sent all our data across the wire, it's time to say au revoir. By default, TCP closes open connections similarly to the way they're opened: lots of handshakery! By default, this time it's a four-way handshake.

Let's take a look at a diagram of a TCP connection closing:

This diagram is similar to the three-way handshake we looked at for connection establishment, so it may raise a question for you: why the extra handshake? The reason, as usual, is reliability!

Remember how TCP maintains timers between segments.? This is because no matter how much reliability we've worked into transport protocols, they're still built on top of unreliable

protocols and infrastructure. The same is true of a closing connection: we don't want to act too quickly or we may miss a piece of extra important data.

When closing a connection, both sides wait a beat before actually closing. This allows any delayed segments to slip in at the last minute! This also means it is impractical for the server to send a piggybacked `FIN` & `ACK` in the same way it sends a `SYN` & `ACK` to open the connection. If the server waited before sending an `ACK`, the client may think something went wrong and begin retransmitting. To prevent this, the server responses are separated:

- the server returns an `ACK` to the client's first `FIN` right away,
- it waits a moment to ensure there are no remaining data segments inbound,
- and then it sends a corresponding `FIN` segment to let the client know it's shutting down.

The client will `ACK`nowledge immediately, but may also wait a bit before truly closing, just in case. With all the handshaking and waiting around, you might call TCP a "considerate" protocol as well!

The TCP Socket State Lifecycle

Remember that a TCP connection is between two sockets, or joint IP address/port pairs. As the connection progresses, these sockets change state. For example, during the process of data transfer, both sockets are considered to be in the `ESTABLISHED` state, while before a connection is established a server's open socket would be in the `LISTEN` state.

It's important to remember that this flow isn't identical for the client & server, too: after the client sends its `SYN`, it enters a `SYN SENT` state while the server enters a `SYN RCVD` (or "`SYN` Received") state of its own.

Here's a diagram of a simple data transfer from start to finish. Notice that we've added the socket states for each side of the connection outside the diagram. These may be noted by network tools and can be helpful to reference if you find yourself debugging a TCP connection problem:

The original TCP specification includes an alternative text-based chart for the lifecycle of these socket states, and you can read more about this process on the Wikipedia page for TCP.

What we've learned

Wow! It's incredible to visualize everything happening across the wire when you browse the Web. TCP is such a foundational piece of your daily interactions with the Internet and will only become more so as you begin contributing to the Web yourself!

After reading this lesson, you should have a clear understanding of:

- TCP segment headers and how they relate to the segment's contents,
- TCP control flags,
- the TCP connection process, including the many handshakes,
- and basic error handling across TCP connections.

Following The Trail With `traceroute`

Remember that the Internet is a "network of networks"1. This can make it tough to debug problems between networks: how do we identify the culprit? Enter traceroute! Let's explore this utility and learn how to find problems with inter-network connections.

We'll cover:

- what `traceroute` is and when to use it,
- how to read `traceroute` output,
- and how to use `traceroute` to solve problems.

Where are we going?

When we create connections between networks, they're rarely direct. We've already discussed how the IP protocol works to connect devices across multiple intermediaries. While the theory and addressing makes sense, it raises a new challenge: how do we solve problems when we can't get access to the other networks involved?

Thankfully, there's a utility that lets us peek at devices along the way. It's called `traceroute`. The `traceroute` utility runs on the command line and uses UDP packets to monitor each device that data passes through as it moves from the source location to the target location. Using this utility, we can determine where a network failure or slowdown might be occurring.

> `traceroute` vs. `tracert`
>
> If you do some research of your own, you may find `traceroute` referred to as `tracert`. These utilities work slightly differently. `tracert` is included with the Windows operating system, and uses ICMP (an alternative protocol also used by the `ping` utility) to trace data. `traceroute` is on Unix-based operating systems, including macOS, and uses UDP. We'll prefer `traceroute` here, but the output of both utilities is almost identical! If you find yourself investigating network problems on a Windows computer, the skills covered here will translate seamlessly.

You'll sometimes hear using `traceroute` referred to as "running a trace". Running a trace of your own is easy - enter the following on your command line:

```
> traceroute appacademy.io
```

Boom! Your first trace! So what does all that gibberish mean? Let's break it down.

Reading a trace

Here's a screenshot of `traceroute appacademy.io` from my terminal:

Metadata

First, let's look at the overall breakdown. At the top of the trace, you'll see some general info. There's a warning about "multiple addresses", meaning that `appacademy.io` resolves to more than one IP address. This is common for popular websites that use multiple servers to reduce traffic and keep speeds high. Next, you'll see the true beginning of the trace, including the domain we're tracing to, the IP address it resolved to, and a maximum number of hops ("64") and packet size "(52 byte").

We mentioned hops when discussing networking protocols, but here's a quick review: a "hop" is one connection to another server. Think of the houses on your street or apartments on your hall. A single hop would walking to your next door neighbor's home. Three hops would be walking three doors down. When we run `traceroute`, it tracks the location of each hop, but it limits the number of hops to make sure we're not searching for an unavailable address forever! Our `traceroute` won't go more than 64 hops, though this default may differ across systems.

Below the general info, we see a numbered list of addresses. Each line represents one hop, and includes some important info about that destination. Let's check it out!

The Hop

Here's our first hop:

We know it's the first because of the `1` on the left side. Each hop is preceded by a number indicating how many hops it took to get there.

Next, we see two IP addresses. These identify the network location our trace has reached. In this case, the location has no resolvable DNS name so we just see the IP address. If you look ahead a bit, you can see that addresses with a resolvable name will show that name first instead.

Finally, we see three numbers. These are time intervals, (indicated by the "ms", short for "milliseconds") that let us know how long it took us to reach this location from our system. When we run a trace, `traceroute` attempts each hop three times. This is because UDP (and the Internet beneath it!) is inherently unreliable. Testing the hop three times ensures we get truly representative data and not a false reading due to a dropped packet or network congestion. Each of the three numbers we see is the response time from one of those attempts. They'll always vary slightly, but we can get a good idea of the average response time. In this case, the numbers are **very** small: two tenths of a millisecond! Wow!.

> This first hop is my home internet router! We can tell this not only because it's the first device reached beyond my own computer, but also because the IP address is within one of IPv4's reserved ranges, meant for private networks inside homes and businesses. It's unlikely you'll see reserved addresses outside of your current network.

Hops proceed from our own device to the gateway for the device we're trying to reach. We can analyze each hop using the same breakdown.

The first hop in our trace is straightforward, but that's not always the case! Let's take a look at a few not-so-standard situations you may see come up.

Special cases

Notice that our second hop doesn't have any of the info we expect from `traceroute`. Instead, it has three asterisks:

In `traceroute`-speak, an asterisk (`*`) represents a hop with no response. This doesn't necessarily mean the hop failed, just that our system didn't get a response back! This could be due to server configuration or a slow connection. By default, a hop will return a `*` if there's no response for five seconds. There are three asterisks in this case to indicate that all three attempts went unanswered.

Let's also take a look at our eighth hop:

This one entry has multiple addresses! What's going on? This is an example of load balancing in action. In this case, the router at our seventh hop is directing traffic to multiple locations. This balances the load and ensures that one single router isn't handling too much.

In this case, one of our hop attempts went to the `atln.ga` (Atlanta, Georgia) router, while the other two went to the `snjs.ca` (San Jose, California) router. We still see timestamps for all three hop attempts.

Because of load balancing, your connection to `appacademy.io` isn't guaranteed to be the same each time. Try running `traceroute appacademy.io` again - do you see the exact same addresses as before?

When should I run a trace?

Tracing is most useful for diagnosing network connectivity issues. Imagine your internet at home suddenly goes down! You can `traceroute` to a familiar domain to see if the connection fails before it gets to your own router (in which case, it's likely a problem on your device), or it fails at a network outside yours.

You can also diagnose slow connections this way! If a hop has a very long response time (> 50 ms), then it's possible that a previous device in line is experiencing downtime or network congestion.

`traceroute` is simple to use without them, but does include some command line arguments that can enhance its abilities. Check out `man traceroute` to learn more about what it can do!

What we've learned

When in doubt, `trace-` the `-route`! Tracing network traffic using the `traceroute` utility is a great way to identify what's happening outside your own network.

After this lesson, you should be comfortable:

- using `traceroute` to diagnose connection problems,
- reading the output of a `traceroute` command,
- and knowing when `traceroute` is the correct tool for the job.

Use Wireshark To Capture Network Traffic

Wireshark, a network analysis tool formerly known as Ethereal, captures packets in real time and display them in human-readable format. Wireshark includes filters, color coding, and other features that let you dig deep into network traffic and inspect individual packets.

This tutorial will get you up to speed with the basics of capturing packets, filtering them, and inspecting them. You can use Wireshark to inspect a suspicious program's network traffic, analyze the traffic flow on your network, or troubleshoot network problems.

Installing Wireshark

You can download Wireshark for Windows or macOS from its official website. If you're using Linux or another UNIX-like system, you'll probably find Wireshark in its package repositories. For example, if you're using Ubuntu, you'll find Wireshark in the Ubuntu Software Center.

Just a quick warning: Many organizations don't allow Wireshark and similar tools on their networks. Don't use this tool at work unless you have permission.

Capturing packets

After downloading and installing Wireshark, you can launch it and double-click the name of a network interface under Capture to start capturing packets on that interface. For example, if you want to capture traffic on your wireless network, click your wireless interface. You can configure advanced features by clicking Capture > Options, but this isn't necessary for now.

As soon as you click the interface's name, you'll see the packets start to appear in real time. Wireshark captures each packet sent to or from your system.

If you have promiscuous mode enabled—it's enabled by default—you'll also see all the other packets on the network instead of only packets addressed to your network adapter. To check if promiscuous mode is enabled, click Capture > Options and verify the "Enable promiscuous mode on all interfaces" checkbox is activated at the bottom of this window.

Click the red "Stop" button near the top left corner of the window when you want to stop capturing traffic.

The packet list pane, located at the top of the window, shows all packets found in the active capture file. Each packet has its own row and corresponding number assigned to it, along with each of these data points:

- **No**: This field indicates which packets are part of the same conversation. It remains blank until you select a packet.
- **Time**: The timestamp of when the packet was captured is displayed in this column. The default format is the number of seconds or partial seconds since this specific capture file was first created.
- **Source**: This column contains the address (IP or other) where the packet originated.
- **Destination**: This column contains the address that the packet is being sent to.
- **Protocol**: The packet's protocol name, such as TCP, can be found in this column.
- **Length**: The packet length, in bytes, is displayed in this column.

- **Info**: Additional details about the packet are presented here. The contents of this column can vary greatly depending on packet contents.

When a packet is selected in the top pane, you may notice one or more symbols appear in the No. column. Open or closed brackets and a straight horizontal line indicate whether a packet or group of packets are part of the same back-and-forth conversation on the network. A broken horizontal line signifies that a packet is not part of the conversation.

Color coding

You'll probably see packets highlighted in a variety of different colors. Wireshark uses colors to help you identify the types of traffic at a glance. By default, light purple is TCP traffic, light blue is UDP traffic, and black identifies packets with errors—for example, they could have been delivered out of order.

To view exactly what the color codes mean, click View > Coloring Rules. You can also customize and modify the coloring rules from here, if you like.

Sample captures

If there's nothing interesting on your own network to inspect, Wireshark's wiki has you covered. The wiki contains a page of sample capture files that you can load and inspect. Click File > Open in Wireshark and browse for your downloaded file to open one.

Download, open, and inspect each of these capture files so you can see what it looks like for that communication to have occurred over a network.

- http.cap A simple HTTP request and response
- dns.cap Various DNS lookups
- wpa.cap 802.11 capture with WPA data encrypted using the password "Induction"

You can also save your own captures in Wireshark and open them later. Click the File > Save to save your captured packets.

Filtering packets

If you're trying to inspect something specific, such as the traffic a program sends when phoning home, it helps to close down all other applications using the network so you can narrow down the traffic. Still, you'll likely have a large amount of packets to sift through. That's where Wireshark's filters come in.

The most basic way to apply a filter is by typing it into the filter box at the top of the window and clicking Apply (or pressing Enter). For example, type "dns" and you'll see only DNS

packets. When you start typing, Wireshark will help you autocomplete your filter.

You can also click Analyze > Display Filters to choose a filter from among the default filters included in Wireshark. From here, you can add your own custom filters and save them to easily access them in the future.

Another interesting thing you can do is right-click a packet and select Follow > TCP Stream.

You'll see the full TCP conversation between the client and the server. You can also click other protocols in the Follow menu to see the full conversations for other protocols, if applicable.

Inspecting packets

Click a packet to select it and you can dig down to view its details.

You can also create filters from here — just right-click one of the details and use the Apply as Filter submenu to create a filter based on it.

Wireshark is an extremely powerful tool, and this tutorial is just scratching the surface of what you can do with it. Professionals use it to debug network protocol implementations, examine security problems and inspect network protocol internals.

Assignment

Now, start Wireshark, start capturing network traffic, and perform various tasks on your computer using different applications. You will be surprised by the number of network requests made by your computer. See if you can identify the application that makes each request and the significance of each request.

DIY Flashcards

There is a lot of terminology and knowledge in today's lessons. These aren't skills that you can necessarily apply. However, the body of knowledge is essential to understand how networking works both in ideal and practical senses.

Today, it's up to you to make your own flashcards for studying. You can make then in your pair and share the Anki files. Use them to study because this will likely be on your assessment and

in some interviews.