Coding Roadmap

Overview

What is the coding roadmap?

The coding roadmap is a series of projects that are designed to help you learn the foundations of programming.

What the coding roadmap is NOT?

The coding roadmap is NOT a course or a tutorial. It is also NOT a substitute for a college degree or a bootcamp.

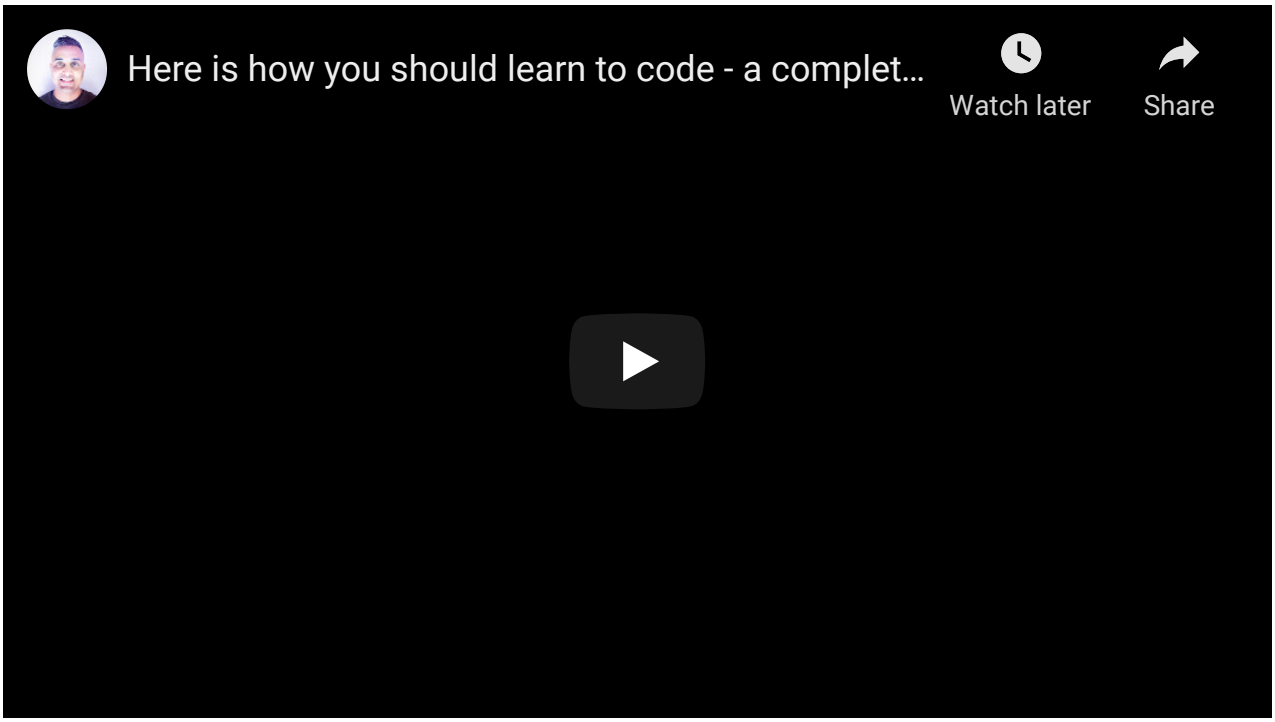
Who is the coding roadmap for?

For the most part, this roadmap is designed for beginners or students, but software engineers of all experience levels may find this useful. This is a great resource for you, if:

1. You are new to programming and want to build a solid foundation around the core concepts of programming.

2. You are a software engineer and want to brush up on the fundamentals.
3. You are already in tech and know a bit up programming, and are planning to making a career switch to software engineering.
4. You are curious about software engineering and want to see if it is a right fit for you.
5. You are preparing for software engineering interviews.

Prerequisites



Watch this to get an overview of this roadmap.

In addition to that, you should know the absolute basics of programming -- primitives, variables, operators, conditionals, loops, functions, objects, etc. Complete Module 0 for a self-assessment.

Books for Data Structures & Algorithms

- Computer Science Distilled (Beginner friendly)

- Grokking Algorithms (Beginner friendly)
- Data Structures and Algorithms in Python (Beginner Friendly)
- Data Structures and Algorithms Made Easy (Intermediate)
- Introduction to Algorithms (Advanced)
- Algorithms 4th Edition (Advanced)
- Algorithm Design Manual (Advanced)

Books for Good Coding Habits

- Clean Code
- The Productive Programmer
- Refactoring

Free Resources

While each module has relevant free resources within it, here are some free resources that are generally applicable to all modules.

- MIT CS 5.000 - Computational Thinking
- MIT CS 6.000 - Introduction to Computer Science & Programming
- MIT CS 6.042 - Mathematics for Computer Science
- MIT CS 6.060 - Introduction to Algorithms
- Learn Python
- Audit C50x at Harvard University for Free
- Intro to Python
- Python for Absolute Beginners
- Complete Data Structures Course

Paid Resources

- Educative.io (10% off via this link)
    - Suggested Course 1: Learn Python 3 from Scratch
    - Suggested Course 2: Python 3: From Beginner to Advanced
    - Suggested Course 3: Data Structures and Algorithms in Python
- LeetCode (15% off via this link) ***you do not need Premium plan for any of the modules*

How to use the coding roadmap?

Each project is designed to be completed in a week or two. Start with Module 0 and work your way through the modules. For each module:

- Read the project description and the project requirements to understand what you will need to deliver first.
- Review the recommended study topics and use the provided resources to learn those topics.
- Solve the self-assessment problems.
- Finish on the project.

Modules

Module 0 - The Basics

This module is mostly designed as a self-assessment to see if you understand the very basics of programming. This is required knowledge for you to proceed to the next module. If you find that you lack even the basic knowledge, use the resources in this module to learn the basics and come back to this module again when you feel ready. If you have the basic knowledge, feel free to skip

this module.

Resources

- How to use Python 3
- Introduction to Python
- Control Flow in Python
- Hello, World!
- Variables and Types
- Lists
- Basic Operators
- String Formatting
- Basic String Operations
- Conditions
- Loops
- Functions
- Classes and Objects
- Modules and Packages

Self Assessment

- Remove duplicated from a sorted array
- Pascal's Triangle
- Contains Duplicate

Module 1 - String manipulation and formatting

This is something you do frequently in the real world. Things like input validation, sanitization, conversion or internationalization are common practice.

Project - Case Converter

In programming, there are five common cases: camelCase, snakecase, kebab-case, PascalCase and UPPER_CASE_SNAKE_CASE. For this project, you are to Write a function that takes in a sentence and a desired case as the inputs, and then converts the sentence into the desired case.

Examples:

```
convert("Hello, World.", "camel")
Output: helloWorld

convert("Hello, World.", "snake")
Output: hello-world

convert("Hello, World.", "kebab")
Output: hello-world

convert("Hello, World.", "pascal")
Output: HelloWorld

convert("Hello, World.", "uppercasesnake")
Output: HELLO_WORLD
```

Resources

- String Formatting
- Basic String Operations
- ASCII
- UTF-8

- Python Strings Tutorial

Self Assessment

- Valid Palindrome
- Reverse string
- Shuffle String
- Reverse only letters
- Rearrange spaces between words

Module 2 - Linked Lists

Linked lists don't get enough love, but they are quite handy data structures, especially you are unsure about the size or the capacity ahead of time. Are you the kind that has 100 tabs open in your browser? Well, then you've been using Linked Lists daily. When you press Ctrl + Tab to cycle between those tabs, you are basically making your way through a circular linked list.

Project - Browser History

You have a **browser** of one tab where you start on the **homepage** and you can visit another **url**, get back in the history number of **steps** or move forward in the history number of **steps**.

Implement the **BrowserHistory** class:

- **BrowserHistory(string homepage)** Initializes the object with the **homepage** of the browser.
- **void visit(string url)** Visits **url** from the current page. It clears up all the forward history.

- **string back(int steps)** Move **steps** back in history. If you can only return **x** steps in the history and **steps > x**, you will return only **x** steps. Return the current **url** after moving back in history **at most** steps.
- **string forward(int steps)** Move **steps** forward in history. If you can only forward **x** steps in the history and **steps > x**, you will forward only **x** steps. Return the current **url** after forwarding in history **at most** steps.

Examples:

```
Input:
["BrowserHistory","visit","visit","visit","back","back","forward","visit
[["leetcode.com"],["google.com"],["facebook.com"],["youtube.com"],[1],[1
Output:
[null,null,null,null,"facebook.com","google.com","facebook.com",null,"li

Explanation:
BrowserHistory browserHistory = new BrowserHistory("leetcode.com");
browserHistory.visit("google.com");       // You are in "leetcode.com".
browserHistory.visit("facebook.com");     // You are in "google.com". Vi
browserHistory.visit("youtube.com");      // You are in "facebook.com".
browserHistory.back(1);                   // You are in "youtube.com", m
browserHistory.back(1);                   // You are in "facebook.com",
browserHistory.forward(1);                // You are in "google.com", mo
browserHistory.visit("linkedin.com");     // You are in "facebook.com".
browserHistory.forward(2);                // You are in "linkedin.com",
browserHistory.back(2);                   // You are in "linkedin.com",
browserHistory.back(7);                   // You are in "google.com", yo
```

Resources

- Singly Linked Lists at Coursera

- CS 61B Linked Lists I
- CS 61B Linked Lists II
- Linked Lists vs Arrays
- Doubly Linked Lists
- More Linked Lists
- Implementation of Linked Lists
- Linked Lists in Python

Self Assessment

- Middle of the Linked List
- Delete Node in a Linked List
- Reverse Linked List
- Intersection of Two Linked Lists
- Linked List Cycle
- Remove Linked List Elements

Module 3 - Hash tables

Hash table is a very popular data structure in practice. It is a simple concept, but quite powerful when you use it effectively. For example, when you type www.engineeringwithutsav.com on your browser, your browser basically looks at your ISPs routing table, also called the DNS server, for the exact IP address of my web server that hosts the contents of my website. That routing table is basically a giant table with two columns: the domain name, which is the key, and the IP address, which is the value. And that's what a hash table is. It stores a bunch of unique keys that have certain values. You give it a key, and it will give you the value very quickly. In this case, you give it the domain name, and

it gives you the IP address so that your browser can fetch the webpage to render. So yeah, you are basically relying on giant distributed hash tables every time you type a domain name on your browser.

Project - Caesar Cipher

A Caesar cipher is a simple method of encoding messages. Caesar ciphers use a substitution method where letters in the alphabet are shifted by some fixed number of spaces to yield an encoding alphabet. A Caesar cipher with a shift of 1 would encode an A as a B, an M as an N, and a Z as an A, and so on. The method is named after Roman leader Julius Caesar, who used it in his private correspondence.

Write a program to encrypt plaintext into ciphertext using the Caesar cipher. Here is the program skeleton:

```
def build_coder(shift):
 """
 Returns a dict that can apply a Caesar cipher to a letter.
 The cipher is defined by the shift value. Ignores non-letter characters
 like punctuation and numbers.
 shift: -27 < int < 27
 returns: dict
 Example: >>> build_coder(3)
 {' ': 'c', 'A': 'D', 'C': 'F', 'B': 'E', 'E': 'H', 'D': 'G', 'G': 'J',
 'F': 'I', 'I': 'L', 'H': 'K', 'K': 'N', 'J': 'M', 'M': 'P', 'L': 'O',
 'O': 'R', 'N': 'Q', 'Q': 'T', 'P': 'S', 'S': 'V', 'R': 'U', 'U': 'X',
 'T': 'W', 'W': 'Z', 'V': 'Y', 'Y': 'A', 'X': ' ', 'Z': 'B', 'a': 'd',
 'c': 'f', 'b': 'e', 'e': 'h', 'd': 'g', 'g': 'j', 'f': 'i', 'i': 'l',
 'h': 'k', 'k': 'n', 'j': 'm', 'm': 'p', 'l': 'o', 'o': 'r', 'n': 'q',
 'q': 't', 'p': 's', 's': 'v', 'r': 'u', 'u': 'x', 't': 'w', 'w': 'z',
 'v': 'y', 'y': 'a', 'x': ' ', 'z': 'b'}
 (The order of the key-value pairs may be different.)
 """
```

```
 ### YOUR CODE GOES HERE


def build_encoder(shift):
 """
 Returns a dict that can be used to encode a plain text. For example, yo
 could encrypt the plain text by calling the following commands
 >>>encoder = build_encoder(shift)
 >>>encrypted_text = apply_coder(plain_text, encoder)
 The cipher is defined by the shift value. Ignores non-letter characters
 like punctuation and numbers.
 shift: 0 <= int < 27
 returns: dict
 Example: >>> build_encoder(3)
 {' ': 'c', 'A': 'D', 'C': 'F', 'B': 'E', 'E': 'H', 'D': 'G', 'G': 'J',
 'F': 'I', 'I': 'L', 'H': 'K', 'K': 'N', 'J': 'M', 'M': 'P', 'L': 'O',
 'O': 'R', 'N': 'Q', 'Q': 'T', 'P': 'S', 'S': 'V', 'R': 'U', 'U': 'X',
 'T': 'W', 'W': 'Z', 'V': 'Y', 'Y': 'A', 'X': ' ', 'Z': 'B', 'a': 'd',
 'c': 'f', 'b': 'e', 'e': 'h', 'd': 'g', 'g': 'j', 'f': 'i', 'i': 'l',
 'h': 'k', 'k': 'n', 'j': 'm', 'm': 'p', 'l': 'o', 'o': 'r', 'n': 'q',
 'q': 't', 'p': 's', 's': 'v', 'r': 'u', 'u': 'x', 't': 'w', 'w': 'z',
 'v': 'y', 'y': 'a', 'x': ' ', 'z': 'b'}
 (The order of the key-value pairs may be different.)
 HINT : Use build_coder.
 """
 ### YOUR CODE GOES HERE


def build_decoder(shift):
 """
 Returns a dict that can be used to decode an encrypted text. For exampl
you
 could decrypt an encrypted text by calling the following commands
 >>>encoder = build_encoder(shift)
 >>>encrypted_text = apply_coder(plain_text, encoder)
 >>>decrypted_text = apply_coder(plain_text, decoder)
 The cipher is defined by the shift value. Ignores non-letter characters
```

```
 like punctuation and numbers.
 shift: 0 <= int < 27
 returns: dict
 Example: >>> build_decoder(3)
 {' ': 'x', 'A': 'Y', 'C': ' ', 'B': 'Z', 'E': 'B', 'D': 'A', 'G': 'D',
 'F': 'C', 'I': 'F', 'H': 'E', 'K': 'H', 'J': 'G', 'M': 'J', 'L': 'I',
 'O': 'L', 'N': 'K', 'Q': 'N', 'P': 'M', 'S': 'P', 'R': 'O', 'U': 'R',
 'T': 'Q', 'W': 'T', 'V': 'S', 'Y': 'V', 'X': 'U', 'Z': 'W', 'a': 'y',
 'c': ' ', 'b': 'z', 'e': 'b', 'd': 'a', 'g': 'd', 'f': 'c', 'i': 'f',
 'h': 'e', 'k': 'h', 'j': 'g', 'm': 'j', 'l': 'i', 'o': 'l', 'n': 'k',
 'q': 'n', 'p': 'm', 's': 'p', 'r': 'o', 'u': 'r', 't': 'q', 'w': 't',
 'v': 's', 'y': 'v', 'x': 'u', 'z': 'w'}
 (The order of the key-value pairs may be different.)
 HINT : Use build_coder.
 """
 ### YOUR CODE GOES HERE


def apply_coder(text, coder):
 """
 Applies the coder to the text. Returns the encoded text.
 text: string
 coder: dict with mappings of characters to shifted characters
 returns: text after mapping coder chars to original text
 Example:
 >>> apply_coder("Hello, world!", build_encoder(3))
 'Khoor,czruog!'
 >>> apply_coder("Khoor,czruog!", build_decoder(3))
 'Hello, world!'
 """
  ### YOUR CODE GOES HERE


def apply_shift(text, shift):
 """
 Given a text, returns a new text Caesar shifted by the given shift
 offset. The empty space counts as the 27th letter of the alphabet,
```

```
so spaces should be replaced by a lowercase letter as appropriate.
Otherwise, lower case letters should remain lower case, upper case
letters should remain upper case, and all other punctuation should
stay as it is.
text: string to apply the shift to
shift: amount to shift the text
returns: text after being shifted by specified amount.
Example:
>>> apply_shift('This is a test.', 8)
'Apq hq hiham a.'
"""
### YOUR CODE GOES HERE
```

Resources

- Intro to Hash tables
- Intro to Hash tables (alternate)
- Hashing and chaining
- Simple Hash table in Python

Self Assessment

- Two Sum
- Unique email addresses
- Intersection of two arrays
- Missing number
- Subdomain visit count

Module 4 - Stacks

Consider your text editor. When you type something, then hit UNDO, you revert your latest action, right? Well, that is utilizing a stack. Your actions are basically added to a stack, which is a LIFO (Last in, First out) data structure, so your last action will always be on the top. And when you hit undo, the stack is simply "popped", undoing your latest action. If you hit undo again, the action you performed previous to that one is reverted since that would have been the second-to-last action you performed. So on and so forth.

Project - Basic Calculator

Given a string **s** which represents an expression, *evaluate this expression and return its value*.

The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of **[-231, 231 - 1]**.

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as **eval()**.

Examples:

Example 1:

Input: s = "3+2*2"
Output: 7

Example 2:

Input: s = " 3/2 "
Output: 1

Example 3:

```
Input: s = " 3+5 / 2 "
Output: 5
```

Resources

- What is a stack?
- Stack data structure
- More stacks
- Stacks in Python

Self Assessment

- Valid parenthesis
- Min stack
- Backspace string compare
- Build an array with stack operations
- Number of students unable to eat lunch

Module 5 - Queues

Queues are pretty self-explanatory and quite common as well. How many of you tried buying a Playstation 5 when it released in the summer of 2020, and got hit with the dreaded "You have been added to a queue... do not refresh the browser" message? What is happening there? Well, because so many people rushed to buy the product at the same time, the web server reached its capacity and could not handle any more traffic. But instead of just erroring out, the load balancer redirected you to a temporary queue, which

is a FIFO (First in, First out) data structure. As traffic improves in the web server, the queue is "dequeued", letting the first person that got put in the queue through to the website, then the second, then third ... just like a queue in the shopping mall.

Project - Circular Queue

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implementation the **MyCircularQueue** class:

- **MyCircularQueue(k)** Initializes the object with the size of the queue to be **k**.
- **int Front()** Gets the front item from the queue. If the queue is empty, return **-1**.
- **int Rear()** Gets the last item from the queue. If the queue is empty, return **-1**.
- **boolean enQueue(int value)** Inserts an element into the circular queue. Return **true** if the operation is successful.
- **boolean deQueue()** Deletes an element from the circular queue. Return **true** if the operation is successful.
- **boolean isEmpty()** Checks whether the circular queue is empty or not.
- **boolean isFull()** Checks whether the circular queue is full or not.

You must solve the problem without using the built-in queue data structure in your programming language.

Examples:

```
Input
["MyCircularQueue", "enQueue", "enQueue", "enQueue", "enQueue", "Rear",
[[3], [1], [2], [3], [4], [], [], [], [4], []]
Output
[null, true, true, true, false, 3, true, true, true, 4]

Explanation
MyCircularQueue myCircularQueue = new MyCircularQueue(3);
myCircularQueue.enQueue(1); // return True
myCircularQueue.enQueue(2); // return True
myCircularQueue.enQueue(3); // return True
myCircularQueue.enQueue(4); // return False
myCircularQueue.Rear();     // return 3
myCircularQueue.isFull();   // return True
myCircularQueue.deQueue();  // return True
myCircularQueue.enQueue(4); // return True
myCircularQueue.Rear();     // return 4
```

Resources

- Queue Python Documentation
- Queues in Python
- Python Queues

Self Assessment

- Implement queue using stacks

- First unique character in a string
- Time needed to buy tickets

Module 6 - Recursion

Recursion is usually the first thing that makes new programmers nervous. And that is expected, because recursion is not a natural thing to happen in real life. It is a very abstract and mathematical concept, and that makes it a bit tricky to grasp at first. So, don't fret it if you end up spending more time on this module. However, in the programming world though, you will see recursive data structures quite frequently. For example, clicking on a folder, then its sub-folder, and then the sub-folder of that folder... this is a recursive action. A folder can have a file or a folder. And for all those sub-folders, the same definition applies -- each one of them can have files or more folders. You get the idea.

Project - Deep Copy

Write a program that can copy all files and folders (including sub-folders) from source location to destination.

Example usage:

```
python copy.py [SOURCE FOLDER] [DESTINATION FOLDER]
will copy everything from source folder to the destination folder
```

Resources

- What on Earth is Recursion?
- Some basic math required to grasp recursion
- Programming Abstractions - Recursion
- Recursion Blueprint
- Recursion Guide
- How recursion works (text)
- How recursion works (video)

Self Assessment

- Reverse string
- Reverse a linked list
- Fibonacci Number
- Merge two sorted lists
- Decode string
- Combination Sum
- Restore IP Addresses
- Permutations II
- Subsets
- Combinations

Module 7 - Binary Search

If you need to search anything that is ordered, you will most likely use binary search. Have you ever searched a physical dictionary for a definition? How did you do it? You surely did not look page after page and word after word, until you found the word you were looking for. You must has started at the middle of the dictionary to check if the word you are looking for falls on the left half or the right half. If it falls on the left half, you search the left half of the dictionary. And if it falls on the

right half, you search the right half of the dictionary. You keep discarding one of the halves until you have landed at the page that has the word you are looking for. Well, this is what binary search is. It is a very efficient search algorithm that discards half of search space at every iteration greatly reducing the time it takes to search for something.

Project - Dictionary Search

Given this word list, write a program that allows you to search for a word. If word is found on this list, it returns the word, if not, it returns the word that is alphabetically closest to it. The search will have be written using binary search.

Examples:

```
Given the word list: [apple, banana, bingo, cat, curl]
Input: bingo
Output: bingo

Input: banjo
Output: bingo

Input: cut
Output: curl

Input: cat
Output: cat
```

Resources

- Binary Search
- Binary Search (HackerRank)

- Binary Search (Brilliant)

Self Assessment

- Binary Search
- Kth Missing Positive Integer
- Guess number higher or lower
- First bad version
- Sqrt(x)

Module 8 - Trees

Trees are a very common data structure in the real world. They are very useful for storing data in a hierarchical manner. For example, you might have a tree that represents a company structure. Each node in the tree has a name, and each node can have a list of employees. And each employee can have a list of subordinates. But not just that, trees are the foundations of storage systems like SQL server and a critical part of searching. For example, when you search for something on YouTube, it auto-completes your word or sentence for you, right? Under the covers, it is basically walking through a special kind of tree called a Trie to produce those suggestions.

Project - Trie

A **trie** (pronounced as "try") or **prefix tree** is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

- **Trie()** Initializes the trie object.
- **void insert(String word)** Inserts the string **word** into the trie.
- **boolean search(String word)** Returns **true** if the string **word** is in the trie (i.e., was inserted before), and **false** otherwise.
- **boolean startsWith(String prefix)** Returns **true** if there is a previously inserted string **word** that has the prefix **prefix**, and **false** otherwise.

Examples:

```
Input
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
Output
[null, null, true, false, true, null, true]

Explanation
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple");   // return True
trie.search("app");     // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app");     // return True
```

Resources

- Tree data structure @ CMU
- What is a Tree Data structure?
- Trees in Python
- Trees at Coursera

Self Assessment

- Binary Tree Inorder Traversal
- Binary Tree Preorder Traversal
- Binary Tree Postorder Traversal
- Binary Tree Level Order Traversal
- Sum of Root To Leaf Binary Numbers
- Maximum Depth of Binary Tree
- Binary Tree Paths
- Same Tree
- Balanced Binary Tree
- Path Sum
- Subtree of Another Tree
- Maximum Depth of N-ary Tree

Module 9 - Graphs

Graphs are everywhere -- social networks, recommendation systems, maps, machine learning, you name it, graphs are lurking behind the scenes. Graphs sound intense because they are often spoken about in very mathematical lingo. For example, "There is an undirected weighted graph made up of X edges and Y vertices, which may or may not have disjoint components. How can we find the shortest path from vertex A to vertex B?" As a beginner software engineer, this may be overwhelming. But don't worry, in reality, this is basically like saying, "We have the map of the United states, that has a bunch of states with multiple roads connecting them and we can drive on those roads on either direction. In addition to that, some of states, like Alaska or Hawaii, may not be accessible by roads at all. Given that information, what do you think is the

shortest way to drive from New York to California?" Not that bad now, eh? This is basically how maps work. Think of graphs as "less structured trees". In fact, a tree is a special type of a graph -- a connected, acyclic and typically undirected graph.

But on occasion, graphs are also a bit sneaky, in that they appear even at places where you'd think they don't belong. For example, consider a spelling checker. You know the one that underlines misspelled words with those squiggly red lines? Well, one of the ways you can build something like that, is by modeling dictionary words as a graph and then using an algorithm called Edit Distance or Levenstein Distance to figure out whether a word could be misspelled. Modeling non-graph-like structures as graphs can be a bit tricky and takes some practice.

Project - ThoughtWorks Trains Problem

The local commuter railroad services a number of towns in Kiwiland. Because of monetary concerns, all of the tracks are 'one-way.' That is, a route from Kaitaia to Invercargill does not imply the existence of a route from Invercargill to Kaitaia. In fact, even if both of these routes do happen to exist, they are distinct and are not necessarily the same distance!

The purpose of this problem is to help the railroad provide its customers with information about the routes. In particular, you will compute the distance along a certain route, the number of different routes between two towns, and the shortest route between two towns.

Input

The input is a directed graph where a node represents a town and an edge represents a route between two towns. The weighting of the edge represents the distance between the two towns. A given route will never appear more than once, and for a given route, the starting and ending town will not be the same town.

Consider this graph as an example input: AB5, BC4, CD8, DC8, DE6,

AD5, CE2, EB3, AE7

This is the input graph, where the towns are named using the first few letters of the alphabet from A to D. A route between two towns (A to B) with a distance of 5 is represented as AB5.

Output

You program is expected to produce the following 10 pieces of information as output about the input graph:

1. The distance of the route A-B-C.

2. The distance of the route A-D.

3. The distance of the route A-D-C.

4. The distance of the route A-E-B-C-D.

5. The distance of the route A-E-D.

***For 1 through 5, if no such route exists, output 'NO SUCH ROUTE'. Otherwise, follow the route as given; do not make any extra stops! For example, the first problem means to start at city A, then travel directly to city B (a distance of 5), then directly to city C (a distance of 4).*

6. The number of trips starting at C and ending at C with a maximum of 3 stops. In the sample data below, there are two such trips: C-D-C (2 stops). and C-E-B-C (3 stops).

7. The number of trips starting at A and ending at C with exactly 4 stops. In the sample data below, there are three such trips: A to C (via B,C,D); A to C (via D,C,D); and A to C (via D,E,B).

8. The length of the shortest route (in terms of distance to travel) from A to C.

9. The length of the shortest route (in terms of distance to travel) from B to B.

10. The number of different routes from C to C with a distance of less than 30. In the sample data, the trips are: CDC, CEBC,

CEBCDC, CDCEBC, CDEBC, CEBCEBC, CEBCEBCEBC.

Example:

```
Input: AB5, BC4, CD8, DC8, DE6, AD5, CE2, EB3, AE7
Outputs:
Output #1: 9
Output #2: 5
Output #3: 13
Output #4: 22
Output #5: NO SUCH ROUTE
Output #6: 2
Output #7: 3
Output #8: 9
Output #9: 9
Output #10: 7
```

Resources

- Introduction to Graphs @ Coursera
- Intro to Graphs
- Graph Theory
- Breadth-first Search
- Depth-first Search

Self Assessment

- Is Graph Bipartite?
- Clone Graph
- Number of Islands
- Rotting Oranges

- N-Queens
- Find if Path Exists in Graph
- Keys and Rooms
- Number of Provinces
- Cheapest Flights Within K Stops
- All Paths From Source to Target
- Number of Operations to Make Network Connected

Module - Final Project

The final project is open-ended. I encourage you to try to come up with your own ideas and build something interesting using all or most of the topics you have learn from module 0 through 9. You can even use the code from the previous projects as a starting point.

Good luck!

Best place to reach out to me is via a DM on Instagram @engineeringwithutsav