

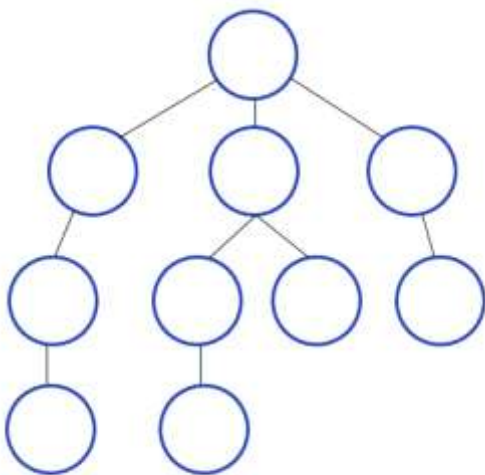
Depth-First Search

Similar to *breadth-first search*, here we will learn about another graph traversal algorithm called *depth-first search*.

Whereas the breadth-first search searches incremental edge lengths away from the source node, *depth-first search* first goes down a path of edges as far as it can.

Once it reaches one end of a path, the search will backtrack to the last node with an un-visited edge path and continue searching.

The animation below shows how the algorithm works. The algorithm starts with the top node and visits the nodes in the numbered order.



(visit URL [Data Structures: Depth-First Search | freeCodeCamp.org](https://www.freecodecamp.org/data-structures-and-algorithms/depth-first-search/))








Notice how, unlike breadth-first search, every time a node is visited, it doesn't visit all of its neighbors. Instead, it first visits one of its neighbors and continues down that path until there are no more nodes to be visited on that path.



To implement this algorithm, you'll want to use a stack. A stack is an array where the last element added is the first to be removed. This is also known as a *Last-In-First-Out* data structure. A stack is helpful in depth-first search algorithms because, as we add neighbors to the stack, we want to visit the most recently added neighbors first and remove them from the stack.

A simple output of this algorithm is a list of nodes which are reachable from a given node. Therefore, you'll also want to keep track of the nodes you visit.

Write a function `dfs()` that takes an undirected, adjacency matrix `graph` and a node label `root` as parameters. The node label will just be the numeric value of the node between `0` and `n - 1`, where `n` is the total number of nodes in the graph.

Your function should output an array of all nodes reachable from `root`.

	The input graph <code>[[0, 1, 0, 0], [1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 1, 0]]</code> with a start node of <code>1</code> should return an array with <code>0, 1, 2, and 3</code> .
	The input graph <code>[[0, 1, 0, 0], [1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 1, 0]]</code> with a start node of <code>3</code> should return an array with <code>3, 2, 1, and 0</code> .
	The input graph <code>[[0, 1, 0, 0], [1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 1, 0]]</code> with a start node of <code>1</code> should return an array with four elements.
	The input graph <code>[[0, 1, 0, 0], [1, 0, 1, 0], [0, 1, 0, 1], [0, 0, 0, 0]]</code> with a start node of <code>3</code> should return an array with <code>3</code> .
	The input graph <code>[[0, 1, 0, 0], [1, 0, 1, 0], [0, 1, 0, 0], [0, 0, 0, 0]]</code> with a start node of <code>3</code> should return an array with one element.
	The input graph <code>[[0, 1, 0, 0], [1, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]]</code> with a start node of <code>3</code> should return an array with <code>2 and 3</code> .
	The input graph <code>[[0, 1, 0, 0], [1, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]]</code> with a start node of <code>3</code> should return an array with two elements.

	The input graph <code>[[0, 1, 0, 0], [1, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]]</code> with a start node of <code>0</code> should return an array with <code>0</code> and <code>1</code> .
	The input graph <code>[[0, 1, 0, 0], [1, 0, 0, 0], [0, 0, 0, 1], [0, 0, 1, 0]]</code> with a start node of <code>0</code> should return an array with two elements.