Graphs

A graph is any collection of nodes and connections between those nodes.

A binary tree is a graph with a lot of restrictions on it. The most prominent ones are that every node has at most one parent and at most two children. When we lift the restrictions, graphs can take many forms.

Graphs need not be connected.

Graphs are part of our everyday lives. Examples are social networks, roads in cities, circuit boards, internet network traffic, crypto blockchains, models in biology, and so much more.

Graph terminology

Edges can be **directed** or **unidirected**. In binary trees, the edges were directed. Binary trees are **directed graphs**. You can't access a node's parent, only its children. Once you move to a child, you can't move back.

Another important term is **connected component**. A connected component is a group of nodes that are connected by edges. In binary trees, there must be only one connected component (all nodes are reachable from the root).

In a directed graph, the number of edges that can be used to reach the node is the node's **indegree**. The number of edges that can be used to leave the node is the node's **outdegree**. Nodes that are connected by an edge are called **neighbors**. In binary trees, all nodes except the root had an indegree of 1 (due to their parent). All nodes have an outdegree of 0, 1, or 2. An outdegree of 0 means that it is a leaf. Specific to trees, we used the parent/child terms instead of "neighbors".

A graph can be cyclic or acyclic. Binary trees cannot have a cycle.

How are graphs given in algorithm problems?

In linked list problems, the `head` of the linked list is given. In binary tree problems, the `root` of the tree is given. In graph problems, only information about a graph is given. There are multiple common formats that this information can come in. We will take a look at a few.

An important thing to understand is that with linked lists and binary trees, you are literally given objects in memory that contain data and pointers. With graphs, the graph doesn't literally exist in memory!

In fact, only the "idea" of the graph exists. The input will give you some information about it, and it's up to you to figure out how to represent and traverse the graph with code.

Many times, the nodes of a graph will be labeled from `0` to `n - 1`. The problem statement may or may not explicitly state the input is a graph. Sometimes there might be a story, and you need to determine that the input is a graph. For example, "there are `n` cities labeled from `0` to `n - 1`". You can treat each city as a node and each city has a unique label.

With binary trees, traversal was easy because at any given `node` we only needed to reference `node.left` and `node.right`. This allowed us to focus only on the traversal (with DFS or BFS). With graphs, a node can have any number of neighbors. Before we start our traversal, we usually need to do some work to makes sure that for any given `node`, we can immediately access all the neighbors of said `node`.

First input format: array of edges

In this input format, the input will be a 2D array. Each element of the array will be in the form `[x, y]`, which indicates that there is an edge between `x` and `y`. The problem may have a story for these edges - using the cities example, the story would be something like "`[x, y]` means there is a highway connecting city `x` and city `y`".

The edges could be directed or undirected. This information will be in the problem description.

So, why can't we immediately start traversal? Let's say that we want to start a DFS from node `0` (sometimes the problem will state which node you should start from, sometimes you will need to figure this out yourself). When we're at node `0`, how do we find the neighbors? We would need to iterate over the entire input to find all edges that include `0`. When we move to a neighbor node, we would need to iterate over the entire input again to find all the neighbors of that node.

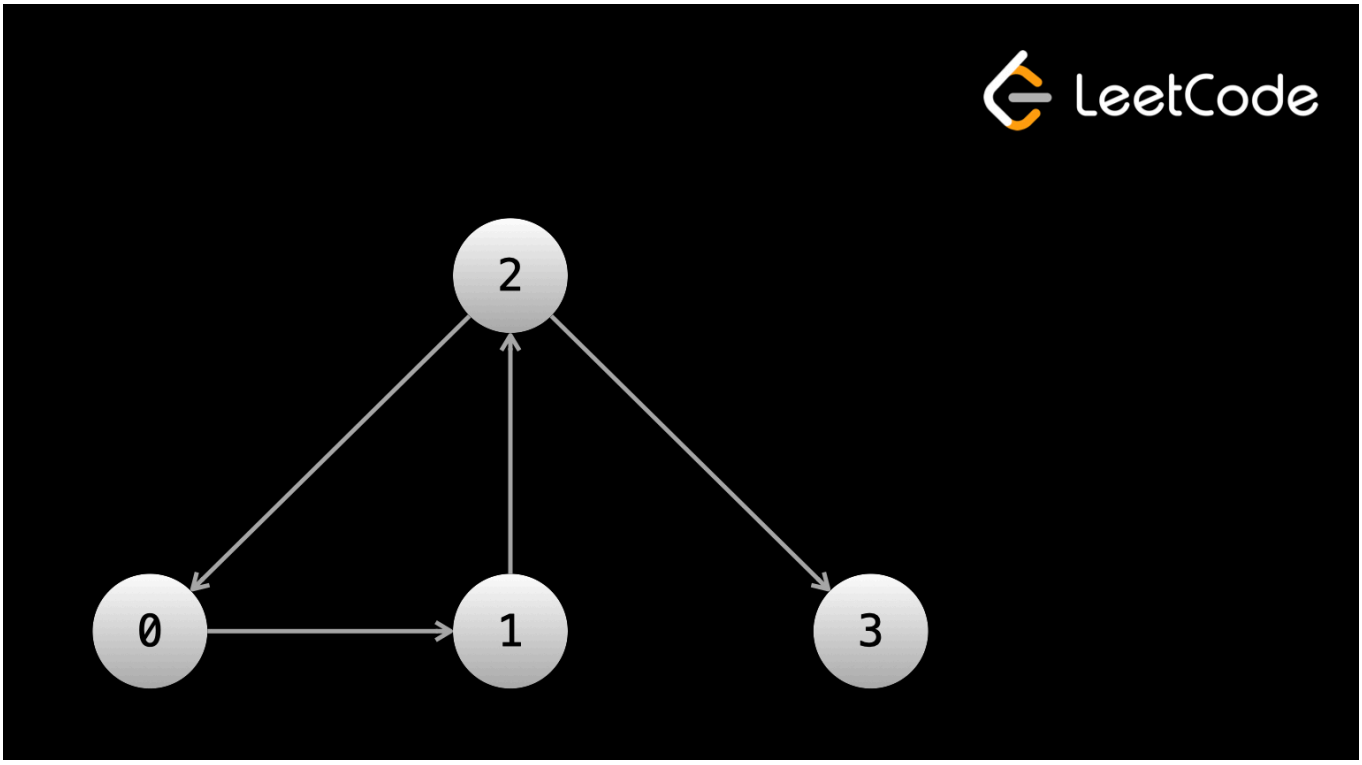
At every node, we would need to iterate over the entire input to find the neighbors. This is very slow!

Before starting the traversal, we can pre-process the input so t hat we can easily find all neighbors of any given `node`. Ideally you want a data structure were you can give `node` as an argument and be returned a list of neighbors. The easiest way to accomplish this is using a hash map.

Let's say you had a hash map `graph` that mapped integers to lists of integers. We can iterate over the input and for each `[x, y]` pair, we can put `y` in the list associated with `graph[x]`. If

the edges are undirected, we will also need to put `x` in the list associated with `graph[y]`. After building this hash map, we can do `graph[0]` and immediately have all the neighbors of node `0`.

A good analogy for this: imagine you're on Facebook and you want to see a list of all your friends. However, the Facebook engineers decided to keep their graph in the form of an array of edges! You would need to look at **every single connection in world** (which is likely in the hundreds of billions if not trillions) and find the connections that involve you. However, if the `graph` is built beforehand, you can easily just click the friends tab on your profile to see only your friends.



This example graph can be represented by an array of directed edges: `edges = [[0, 1], [1, 2], [2, 0], [2, 3]]`.

Notice that the graph in the image does not exist in memory. It exists only as an idea derived from the array `[[0, 1], [1, 2], [2, 0], [2, 3]]`.

Here's some example code for building `graph` from an array of edges:

```
from collections import defaultdict

def build_graph(edges):
    graph = defaultdict(list)
    for x, y in edges:
        graph[x].append(y)
```

```
        # graph[y].append(x)
        # uncomment the above line if the graph is undirected

    return graph
```

Second input format: adjacency list

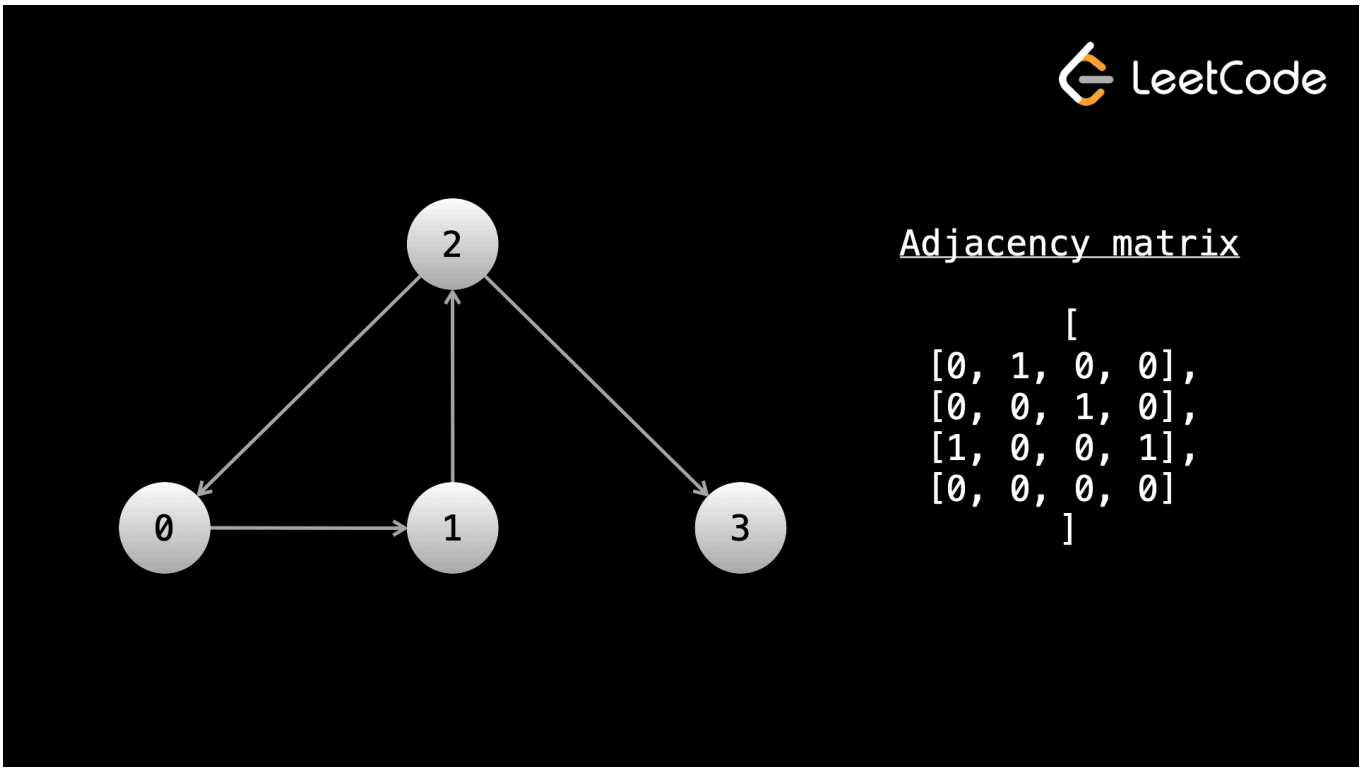
In an adjacency list, the nodes will be numbered from `0` to `n - 1`. The input will be a 2D integer array, let's call it `graph`. `graph[i]` will be a list of all the outgoing edges from the ith node.

The graph in the image above can be represented by the adjacency list `graph = [[1], [2], [0, 3], []]`.

We don't need to do any pre-processing! This makes an adjacency list the most convenient format. If we want all the neighbors of node `6`, we just check `graph[6]`.

Third input format: adjacency matrix

The next format is an adjacency matrix. Once again, the nodes will be numbered from `0` to `n - 1`. You will be given a 2D matrix of size `n x n`, let's call it `graph`. If `graph[i][j] == 1`, this means there is an outgoing edge from node `i` to node `j`. For example,



When given this format, you have two options. During the traversal, at any given `node` you can iterate over `graph[node]`, and if `graph[node][i] == 1`, then you know that node `i` is a neighbor. Alternatively, you can pre-process the graph as we did with an array of edges. Build a hash map and then iterate over the entire graph and then iterate over the entire `graph`. If `graph[i][j] == 1`, then put `j` in the list associated with `graph[i]`. So you are kind of throwing out all the `0`s. This way, when performing the traversal, you will not need to iterate `n` times at every node to find the neighbors. This is especially useful when nodes have only a few neighbors and `n` is large.

Both of these approaches will have a time complexity of $O(n^2)$.

Last input we'll talk about is more subtle but very common. The input will be a 2D matrix and the problem will describe a story. Each square represents something. and the squares will be connected in some way. For example, "Each square of the matrix is a village. Villages trade with their neighboring villages, which are the villages directly above, to the left, to the right, or below them."

In this case, each square `(row, col)` of the matrix is a node, and the neighbors are `(row - 1, col), (row, col - 1), (row + 1, col), (row, col + 1)` (if in bounds).

Unlike other input formats, the nodes in these graphs are not numbered `0` until `n`. Instead, each element in the matrix represents a node. The edges are determined by the problem description, not the input. In the example above, the problem description states that villages

trade with those directly adjacent to them. Thus the edges are those within 1 square. You will need to carefully think about the problem to understand these kinds of graphs.

Code differences between graphs and trees

There are a few big differences between solving graph problems and solving binary tree problems. While a binary tree has a `root` node to start traversal from, a graph does not always have an obvious "start" point.

For a binary tree, we are given objects for the nodes, and each node has a pointer to its children. In a graph, we might need to convery the input into a hash map first. When traversing a tree, we refer to `node.left` and `node.right` at each `node`. When traversing a graph, we will need to use a for loop to iterate over the neighbors of the current node, since a node could have any number of neighbors.

Implementation of DFS for graphs is similar to implementation for trees. Doing it recusively follows the same format: check for the base case, recusively call on all neighbors, do some logic to calculate the answer, and return the answer. You can also do it iteratively using a stack.

In any undirected graph or a directed graph with cycles, implementing DFS the same way we did with binary trees will result in an infinte cycle (remember linked list cycles? Imagine having our code walk in a circle forever!). Like with trees, in most graph questions, we only need to (and want to) visit each node once. To prevent cycles and unnecessarily visiting a node more than once, we can use a set `seen`. Before we visit a node, we first check if the node is in `seen`. If it isn't, we add it to `seen` before visiting it. This allows us to only visit each node once in $O(1)$ time because adding and checking for existence in a set takes constant time.

This wasn't necessary with trees because we started at the root and the edges only moved "down" - once we left a node, there was no way to get back to it. With graphs, you could have a relationship like `A <-> B`, and move between `A` and `B` forever.

As for the node we should start traversing from, this will depend on the problem and what you are trying to solve.

In a language like Python, using a set for `seen` is very easy and relatively fast. In other languages, it may be much better (in terms of runtime) to use an array for `seen` if the range of states is known (which it usually is, because most graph problems have the nodes numbered from `0` to `n - 1`).