TIP102 | Intermediate Technical Interview Prep

Intermediate Technical Interview Prep Spring 2025 (@ Section 3 | Tuesdays and Thursdays 6PM - 8PM PT)
Personal Member ID#: 117667

Unit 10 Cheatsheet

Overview

Here is a helpful cheatsheet outlining common syntax and concepts that will help you in your problem-solving journey! Use this as a reference as you solve the breakout problems for Unit 10. This is not an exhaustive list of all data structures, algorithmic techniques, and syntax you may encounter; it only covers the most critical concepts needed to ace Unit 10. In addition to the material below, you will be expected to know any required concepts from previous units.

Standard Concepts

Graphs

A graph is a data structure that is used to represent relationships between different entities. For example, a graph can be used to describe relationships between people or roads between cities.

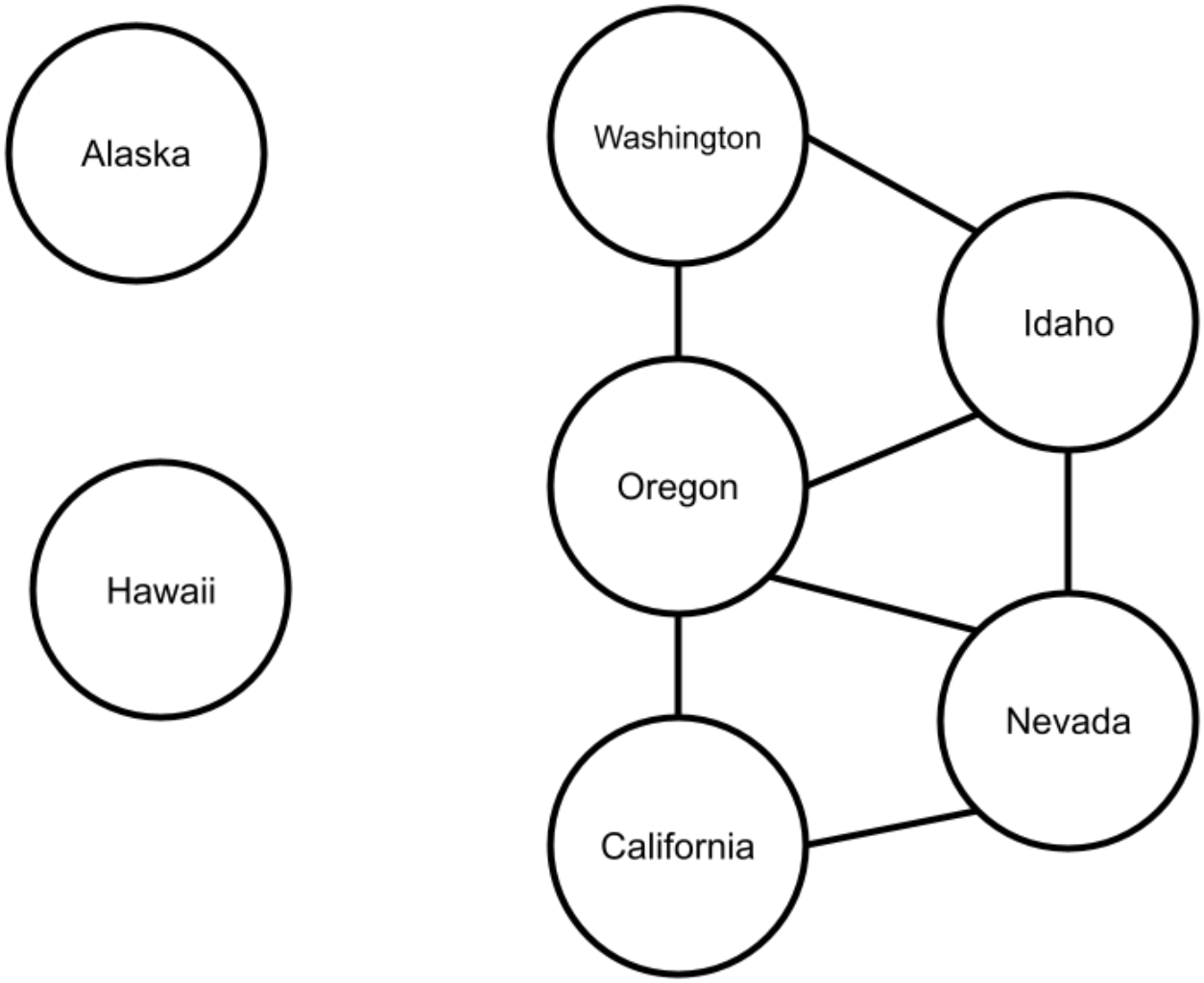
Graphs consist of two components:

- **Nodes** represent different entities
    - Nodes are also referred to as **vertices**
- **Edges** connect two nodes, representing a relationship between two nodes.

Like trees, graphs are a non-linear data structure. But whereas a node is linked to at most two other nodes in a binary tree, graph nodes can have an edge to any number of other nodes.

When diagramming graphs, labeled circles are used to represent nodes, and solid lines are used to denote edges to other nodes.

For example, in the graph below, each node is a state and an edge between two nodes indicates that the states share a border.



Nodes that share a direct edge are referred to as **neighbor** nodes. For example, California and Oregon are neighbors. California and Idaho are not neighbors because they do not share an edge, but it is possible to **reach** Idaho from California if we follow a **path** or sequence of edges that connects a series of nodes from California to Oregon and then to Idaho. Alternatively, we could follow the path from California to Nevada to Idaho, or even California to Oregon to Washington to Idaho. There are multiple paths between the two states.

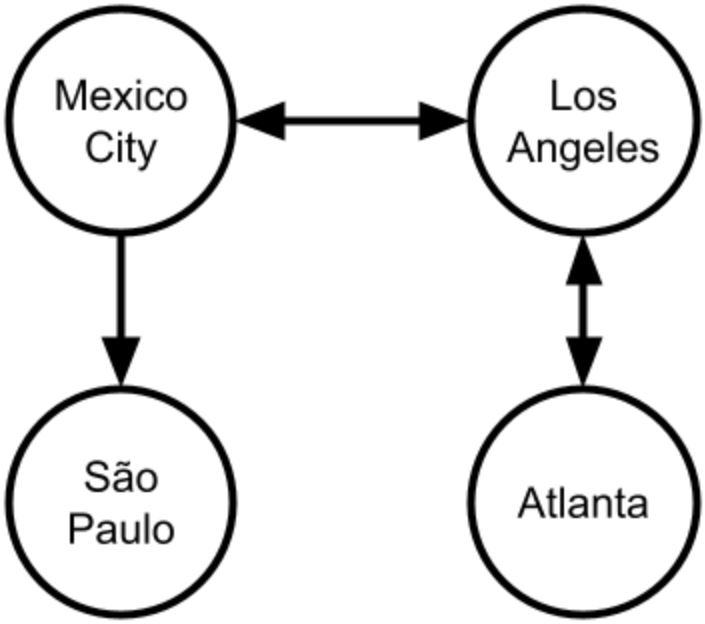
Notice that because Hawaii and Alaska are not connected to the lower 48 states, they have zero edges. We say that it is not possible to reach Hawaii from Oregon because there is no path of edges between the two nodes. While these nodes are disconnected from the rest of the graph, they are still part of the graph and form distinct components. We call a group of nodes where any two nodes are connected to each other by a path a connected **component**. The above graph has three connected components:

- Hawaii
- Alaska
- Washington, Oregon, California, Idaho, and Nevada

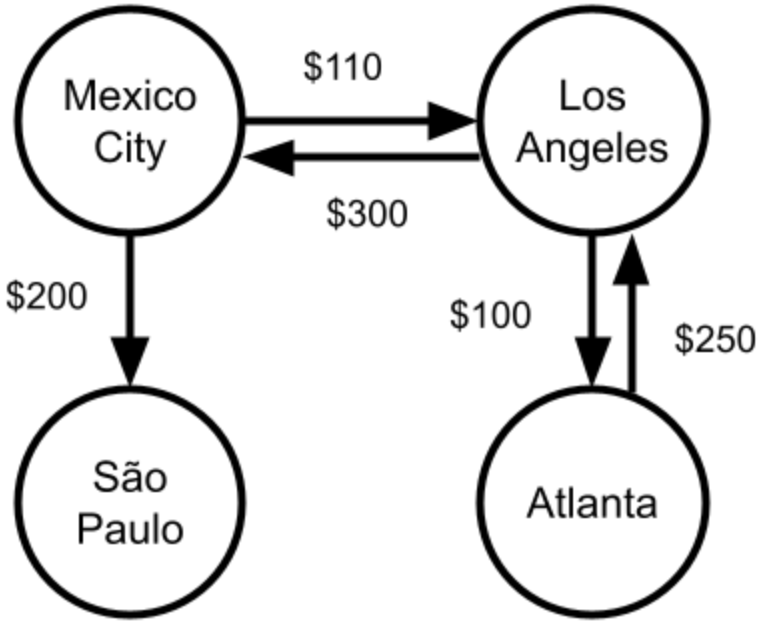
In the above graph, edges are **undirected**, meaning that there exists a two-way relationship between the two nodes. If Oregon shares a border with California, then California also shares a border with Oregon.

Edges can also be **directed**. Directed edges are used to model one-way relationships, and in diagrams are represented with an arrow.

For example, in the graph below, each node represents a city and each edge represents an available flight from one city to another. We can see that there exists a flight from Mexico City to Los Angeles and a corresponding flight in the opposite direction from Los Angeles to Mexico City. However, while there is a flight from Mexico City to São Paulo, there is no return flight in the other direction.



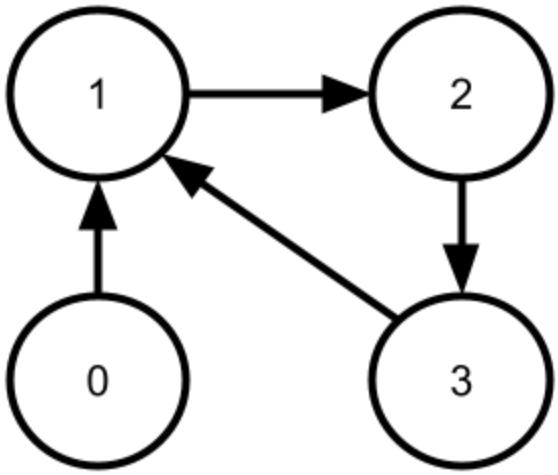
Edges can also have **weights** or **costs**. We can add weights to the previous flights graph, where the weight of each edge represents the cost of the flight. For example the flight from Mexico City to Los Angeles has cost $110, and the flight from Los Angeles to Mexico City has cost $300.



`irected Acyclic Graphs (DAGs)

Graphs can have **cycles**. A cycle is a path that starts at a particular node in the graph, visits other nodes, and then returns to the starting node.

In the example below, there is a cycle from node 1 to node 2 to node 3 and back to node 1.



A directed acyclic graph (DAG) is a special type of graph that maintains the following properties:

- Edges are directed.
- The graph has no cycles.

Because of its properties, DAGs can be *topologically sorted* which will be explored further in the advanced version of Unit 11.

Graph Representations

There are many different ways we can represent a graph.

Node Class

Similar to the way in which we represented nodes for Linked Lists and Binary Trees, we can create a `GraphNode` class that has two properties:

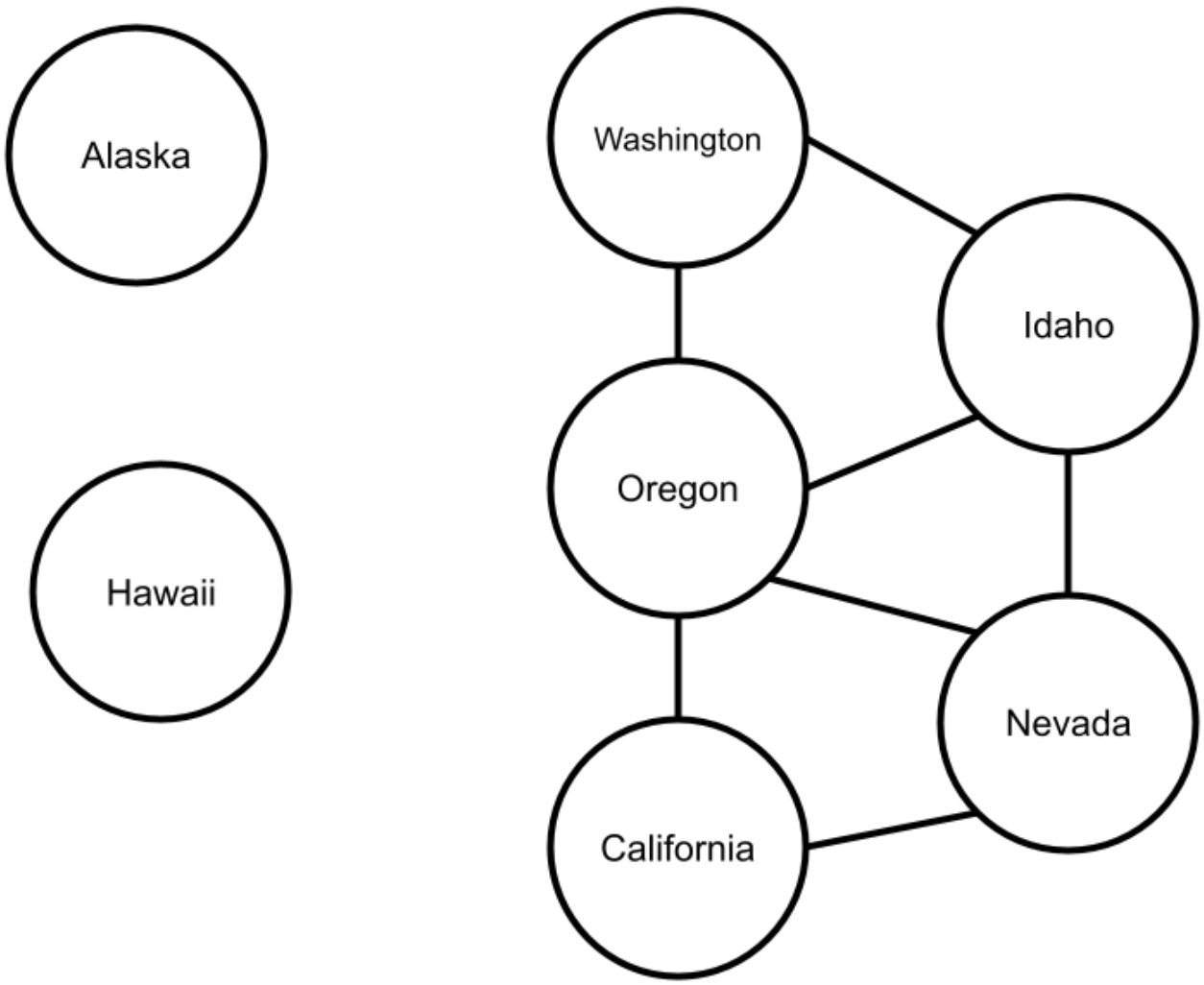
- `val` to hold the node data
- a list `edges`, where each item in `edges` is another `GraphNode` instance that the current node shares an edge with.

```
class GraphNode:

  def __init__(self, value, edges = None)
      self.val = value
      if not edges:
          self.edges = []
      else:
          self.edges = edges

  def add_connection(self, new_node):
      self.edges.append(new_node)
```

However, unlike with linked lists and binary trees, this form of graph representation is fairly uncommon.

In a linked list, all nodes are reachable from the `head` node. In a binary tree, all nodes are reachable from the `root` node. Graphs do not have an equivalent to the `head` / `root` node. We may begin a traversal from any point in the graph.

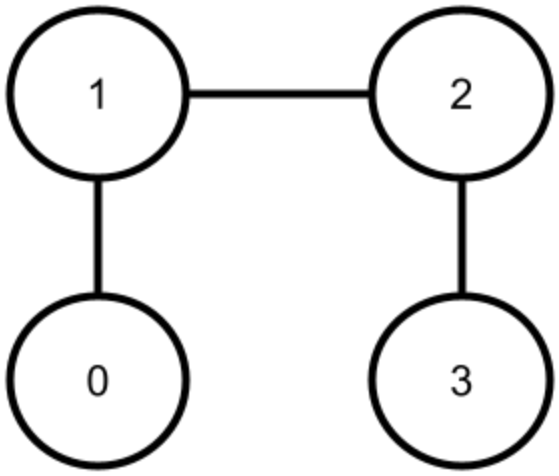
Using a singular `GraphNode` to represent a graph doesn't work well in graphs with multiple components. Using the `edges` property to access other nodes in the graph, we will be able to access any node for which there exists a path from the start node to the destination node. But we would never find disconnected components like the nodes for Hawaii and Alaska in the example below.



List of Edges

One basic way to represent a graph is with a list of edges. A list of edges is a 2D list, where each inner list is of the form `[start_node, dest_node]` indicating the existence of an edge between the `start_node` and `dest_node`.

Example Usage:




```
list_of_edges = [
    [0, 1],
    [1, 2],
    [2, 3]
]
```

Each inner list `[start_node, dest_node]` can represent either an undirected or directed edge. This is usually specified in the problem statement. For example, if each edge is undirected, the problem statement might say that `list_of_edges[i] = [start_node, dest_node]` represents an edge **between** `start_node` and `dest_node`. If each node is directed, the problem statement might say that `list_of_edges[i] = [start_node, dest_node]` represents a node **from** `start_node` **to** `dest_node`.

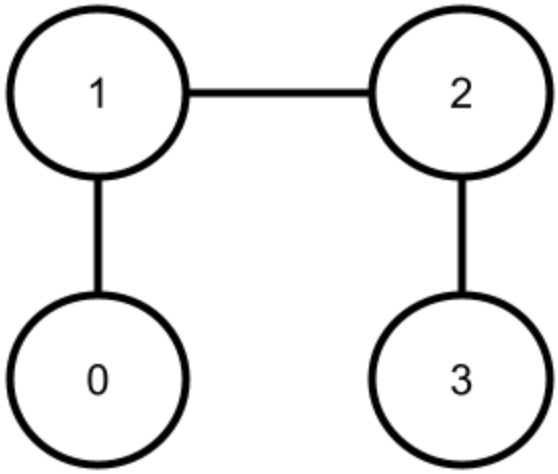
When using a list of edges, it is common to also provide the number of nodes in the graph `n`, in case there are any disconnected nodes in the graph with no edges.

To represent a weighted graph as a list of edges, we simply append the weight a third value to each edge list where `list_of_edges[i] = [start_node, dest_node, weight]`.

Adjacency List

An adjacency list also uses a 2D list to represent a graph. However this time, the *indices* of the outer list are used to represent each node in the list. The value at each index represents an unordered list of node `i`'s neighbors.

That is, `adjacency_list[i]` represents all the neighbors of node `i` in the graph. Example Usage:



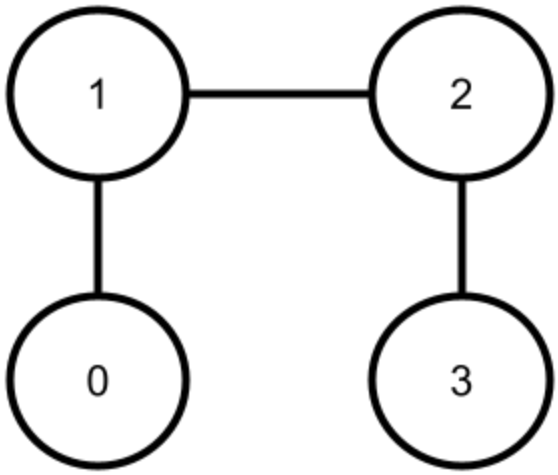

```
adjacency_list = [
    [1],       # Node 0's edges/neighbors
    [0, 2],    # Node 1's edges/neighbors
    [1, 3],    # Node 2's edges/neighbors
    [2]        # Node 3's edges/neighbors

]
```

Notice that because the example above is an undirected graph, where `adjacency_list[i]` contains a node `j`, `adjacency_list[j]` also contains node `i`. For example `adjacency_list[0]` has node `1`, and `adjacency_list[1]` also has node `0` denoting that there is an undirected edge between node `0` and node `1`.

If a node `i` has no edges to another node, then `adjacency_list[i]` is an empty list.

Adjacency lists are also often represented using dictionaries. When using a dictionary each key represents a node in the graph, and each corresponding value is a list of that key node's neighbors.

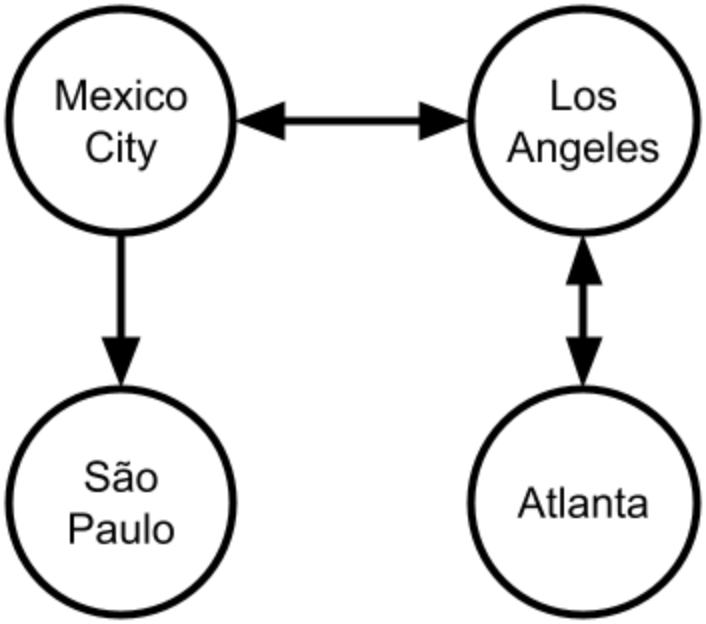
Example Usage:



```
adjacency_dictionary = {
    0: [1],
    1: [0, 2],
    2: [1, 3],
    3: [2]
}
```

Because we label nodes by key instead of index, we can easily use a dictionary to represent nodes that hold non-numerical data.

Example Usage:

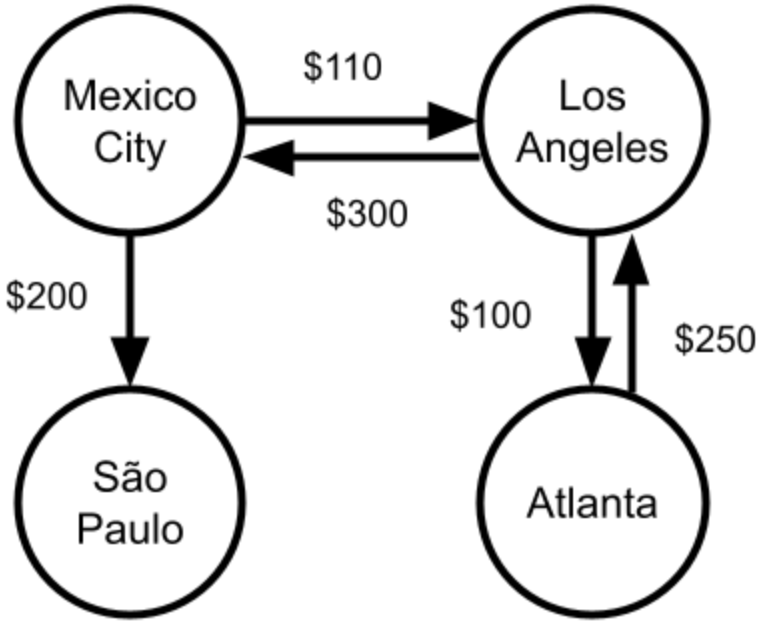



```
adjacency_dictionary = {
    "Mexico City": ["São Paulo", "Los Angeles"],
    "Los Angeles": ["Mexico City", "Atlanta"],
    "Atlanta": ["Los Angeles"],
    "São Paulo": []
}
```

For clarity, CodePath will refer to a dictionary graph representation as an **adjacency dictionary** in the problem sets, but you may see it lumped in with the more general term "adjacency list" elsewhere.

To represent a weighted graph in either an adjacency list or dictionary, we use a tuple to represent each edge. The first value in each tuple represents the neighboring node and the second value represents the weight.

Example Usage:

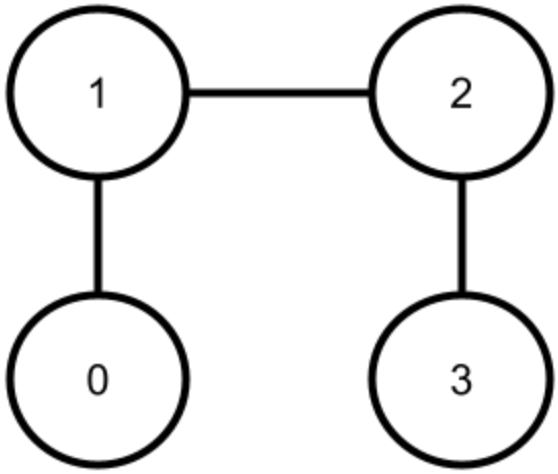



```
adjacency_dictionary = {
    "Mexico City": [("São Paulo", 200), ("Los Angeles", 110)],
    "Los Angeles": [("Mexico City", 300), ("Atlanta", 100)],
    "Atlanta": [("Los Angeles", 250)],
    "São Paulo": []
}
```

Adjacency Matrix

In an adjacency matrix we create an `n x n` 2D list or matrix to represent the graph where `n` is the number of nodes in the graph. `adjacency_matrix[i][j] = 1` indicates that there is a directed edge from node `i` to node `j` in the graph. A value of `0` indicates that there is no edge from `i` to `j`. Unless a node explicitly has an edge to itself, the default value for `adjacency_matrix[i][i]` is `0`.

You may occasionally see boolean values of `True` / `False` used in place of `0` / `1`.



```
adjacency_matrix = [
                    [0, 1, 0, 0], # Node 0
                    [1, 0, 1, 0], # Node 1
                    [0, 1, 0, 1], # Node 2
                    [0, 0, 1, 0]  # Node 3
]
```

In an undirected graph, `adjacency_matrix[i][j] = adjacency_matrix[j][i]`.

To represent a weighted graph we simply replace the value of `1` with the edge weight. Usually a value of `0` still indicates the absence of an edge in a weighted graph. If you wanted to use `0` as a possible weight for an edge, you could use some other value to represent the absence of an edge like `None`.

Graph Traversal

There are two primary algorithms used to traverse graphs. The pseudocode and logic behind the base algorithms using adjacency lists is outlined below. These two algorithms are the foundation for solving many graph problems, but often need to be modified to work with different graph representations and to solve each individual problem.

Graph traversals can start from any node in the graph. Both Breadth-First Search (BFS) and Depth-First Search (DFS) will find all nodes that are reachable from the start node. The difference between the two algorithms lies in the order they traverse the nodes.

Breadth-First Search (BFS)

BFS explores nodes **level by level**, meaning that it first visits all nodes at a distance of one edge from the starting node, then all nodes at a distance of two edges, and so on. Because of this behavior, BFS guarantees finding the shortest path in **unweighted** graphs, or graphs where all edges have the same weight. This is because BFS reaches a node in the fewest possible edges before visiting nodes that are further away.

However, BFS guarantees the shortest path **only when all edges have equal weight or no weight**. In graphs with weighted edges, algorithms like Dijkstra's are needed to find the shortest path.

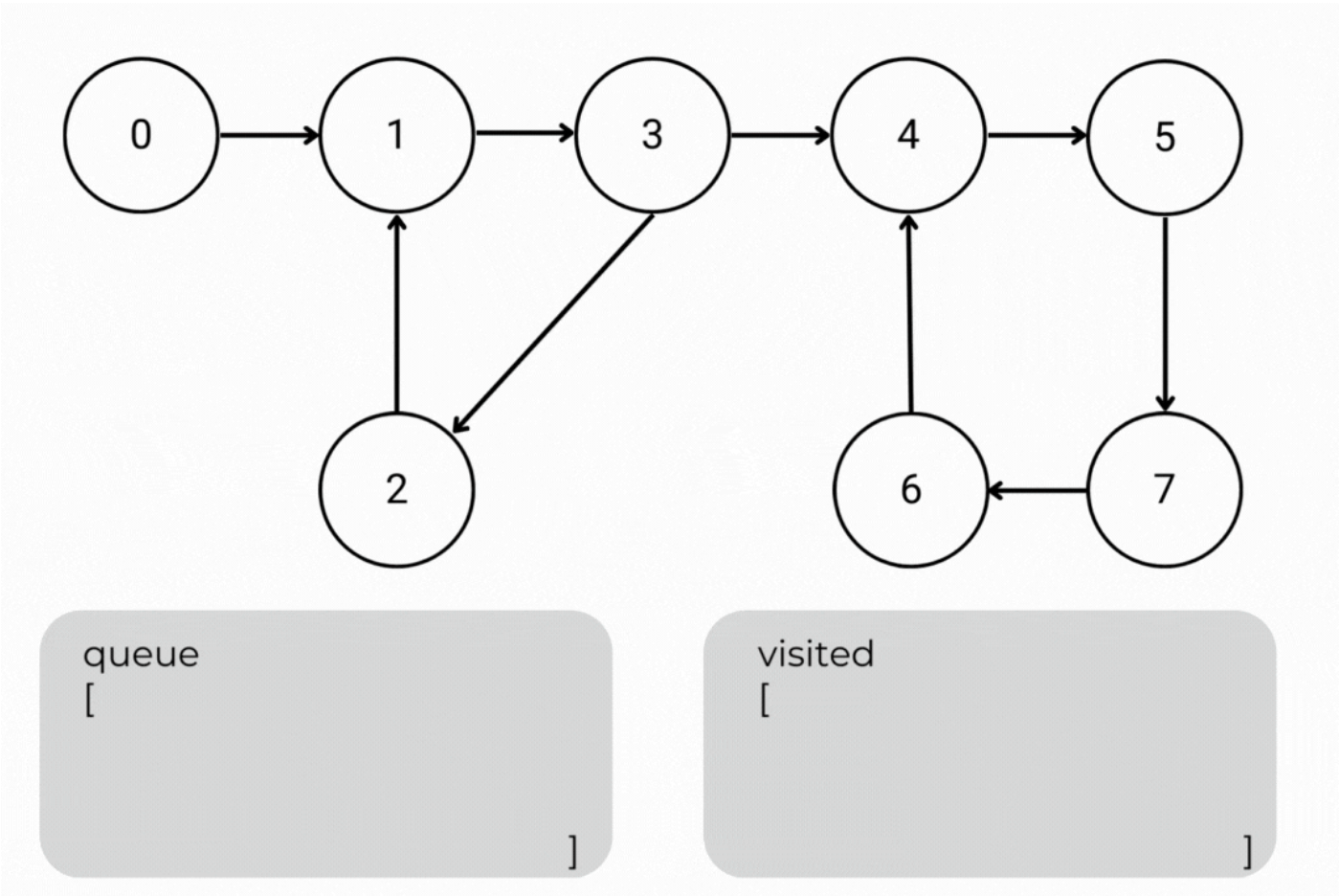
Given a starting node, BFS visits the starting node's nearest nodes - neighbors - first, then its neighbors' neighbors, and so on and so forth until it has visited all of the nodes it can reach via a path of edges. Notice that this is the same pattern as BFS for trees: given the starting node (the root node), BFS will explore the root's children, grandchildren, and so on, until it reaches the leaves of the tree.

Because graphs may contain cycles that lead us back to nodes we have already visited, we need to keep track of which nodes we have already visited to ensure that we only traverse each node once. Otherwise, we will get stuck in a never-ending loop!

BFS uses a queue to determine which node should be visited next. We start by adding our starting node to the queue. Then we begin a loop that continues iterating so long as the queue is not empty (so long as there are still nodes left to visit).

Inside our loop, we dequeue a node - initially this is our starting node. We then loop through the node's neighbors and add any unvisited neighbors to the queue.

Then we start our outer loop again! Because queues are first-in-first-out, the next nodes to be visited will be the starting node's neighbors!



The pseudocode for BFS is as follows:

```
- Initialize an empty list of visited nodes
- Initialize an empty queue
- Add the node we would like to start our traversal from to the queue
- Add the node we would like to start our traversal from to visited
- While the queue is not empty:
    - Remove an element from the queue and store it in a variable, `current`
    - Loop through each of the current node's neighbors:
        - If the neighbor has not yet been visited:
            - Add the neighbor to the queue
            - Add the neighbor to the list of visited nodes
- Return the list of visited nodes
```

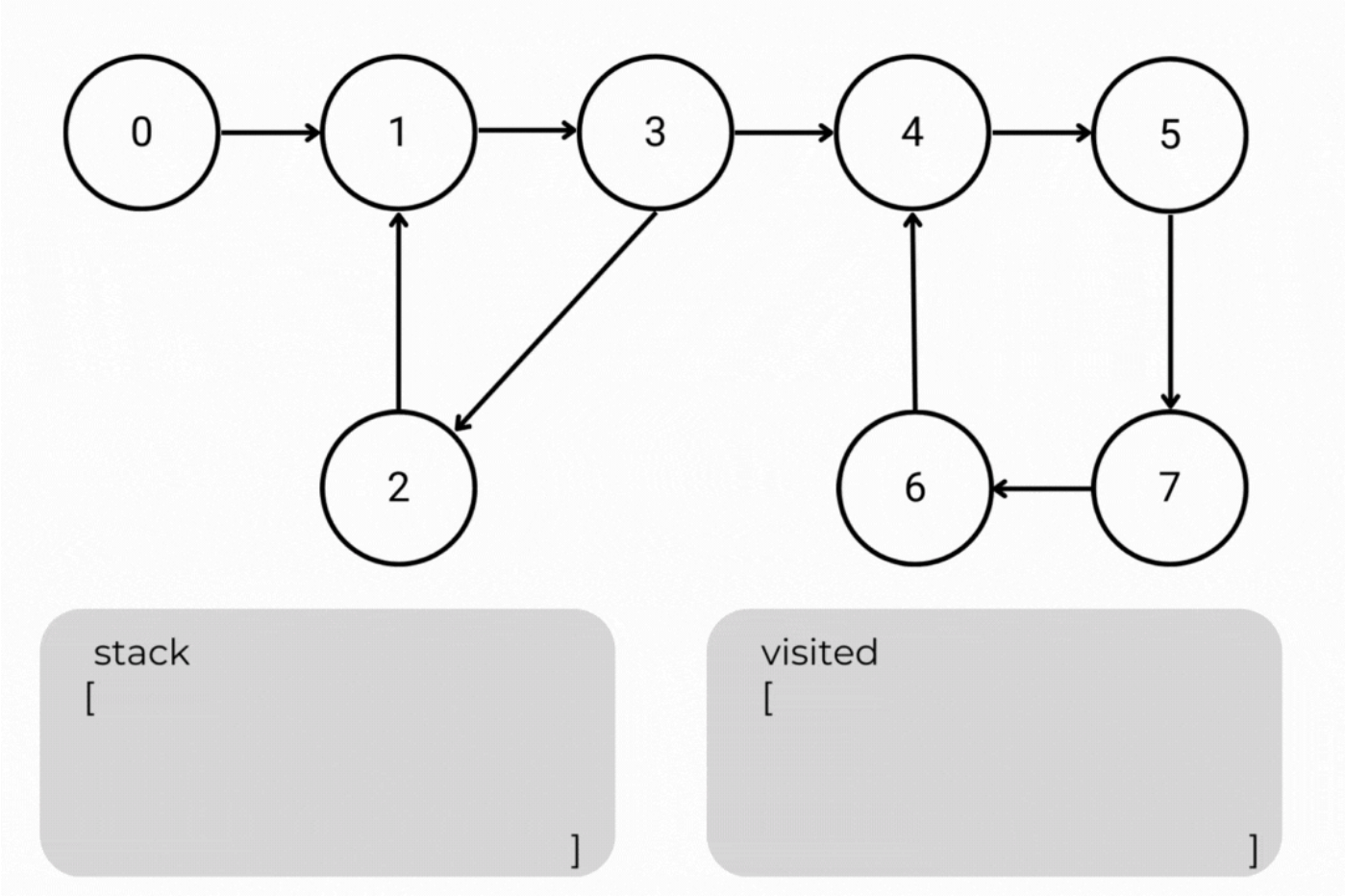
Note again that BFS with the above pseudocode, BFS will only find *reachable* nodes from the start node. To traverse all nodes in the graph, including nodes unconnected to the start node, we must run BFS multiple times. We can do this by checking which nodes have not yet been visited after our initial execution of BFS, and choose an unvisited node as our new starting node. We can continue this pattern until we find all nodes have been visited.

Depth-First Search (DFS)

DFS acts as if it were exploring a maze. At each fork in the road (each node in the graph) it will choose one path to travel down (usually following an edge towards an unvisited neighbor). It will continue to follow the path, moving further away from the start node, until it cannot find any further untraversed edges to travel along from its current position.

Once it reaches a dead-end, it will return to the most recently visited node where it was possible to travel down an alternative path (choose a different edge) and follow that path as far as it can go. It will continue this pattern until it has traversed all paths that can be reached from the starting node.

As with BFS, we need to keep track of which nodes we have already visited to avoid getting stuck in a cycle and ensure that we only traverse each node once.



DFS uses a stack to determine which node to visit next. We start by adding our starting node to the stack. Then we begin a loop that continues iterating so long as the stack is not empty (so long as there are still nodes left to visit).

Inside our loop, we pop the next node off the stack - initially this is our starting node. We then loop through the node's neighbors and add any unvisited neighbors to the stack.

Then we start our outer loop again! Because stacks are last-in-first-out, the next node to be visited will be the most recently added node to our stack. This will lead us to visit nodes further away from the starting node before visiting all of the starting node's neighbors.

The pseudocode for DFS is as follows:

```
- Initialize an empty stack
- Initialize an empty list to store visited nodes
- Add the node we would like to start our traversal from to the stack
- while the stack is not empty:
    - Pop the topmost node off the stack and store it in a variable, `current`
    - If the node is not already in the list of visited nodes:
        - Add `current` to the list of visited nodes
    - Loop through the current node's neighbors:
        - If the neighbor has not yet been visited
            - Push the neighbor onto the stack
- Return list of visited nodes
```

Notice that this is very similar to the BFS algorithm above, except that we have replaced the queue with a stack. Because we are using a stack data structure, DFS is more commonly implemented recursively to take advantage of the recursive call stack.

The pseudocode for recursive DFS is as follows:

```
- Main function
    def dfs(self, start_node):
        - Create empty list to store nodes that have been visited
        - Pass list of visited nodes and `start_node` into `dfs_helper`
        - Return list of visited nodes

- Helper function
    def dfs_helper(self, start_node, visited)
        - If `start_node` is not in list of visited nodes
            - Append `start_node` to list of visited nodes
            - Loop through `start_node`'s neighbors
                - Call dfs_helper passing in list of visited nodes and the neighbor
                  as the new start node
```

BFS and DFS Big O

Both BFS and DFS have `O(V + E)` time complexity because each node and edge is explored once while traversing a fully connected graph.

BFS vs DFS

For many problems, both BFS and DFS can be used to find a solution. In some scenarios, we prefer one over the other.

Both

- Finding connected nodes in a graph
- Finding a path between two nodes in a graph
- Traversing all nodes in the graph

DFS Special Use Cases

- Finding the shortest/least-cost path in an *unweighted* graph
    - Because BFS visited closest nodes first, in an unweighted graph it will always find the shortest path between two nodes
- When the node you are searching for is likely to be close to the start node

DFS Special Use Cases DFS is especially effective for cycle detection in directed graphs by checking for back edges during traversal, while BFS is often used for detecting cycles in undirected graphs.

- **Backtracking problems**
    - DFS naturally lends itself to backtracking problems, because it explores one path until it hits a dead end, then 'backtracks' to try an alternative route
- **Detecting Cycles in Graphs**
    - DFS is especially effective for cycle detection in directed graphs by checking for back edges during traversal. As DFS travels down a single path, it will eventually find that the path leads to an already visited node, indicating a cycle. While BFS can also be used for cycle detection, it is more commonly used for detecting cycles in undirected graphs, whereas DFS is often more efficient and suited for directed graphs.
- **Topological Sorting of Directed Acyclic Graphs**
    - DFS is a common method for performing topological sorting on DAGs, since the depth-first nature of the algorithm allows it to process nodes in the correct order without cycles.

Advanced Concepts

Graph Representation Big O

Each graph representation has different time and space complexities for common graph operations. When choosing a graph operation, you may choose the representation based upon what you want to optimize for.

List of Edges

- **Determining if Two Nodes Share an Edge**: `O(E)` time complexity where `E` is the number of edges in the graph.
- **Retrieving A Nodes Neighbors**: `O(E)` time to find *all* of a nodes neighbors, as we have to search through each edge in the list
- **Space Complexity**: `O(E)` space (one list of length 2 for each edge) to represent the entire graph.

Adjacency Lists

- **Determining if Two Nodes Share an Edge**: `O(D)` time complexity where `D` is the **degree** of the node, meaning the number of edges the node has.

    - In the worst case, each node will have `O(N)` edges where `N` is the number of nodes in the graph. In this scenario every node is connected to every other node in the graph.

- **Retrieving A Nodes Neighbors**: `O(1)` time to find all the neighbors of a particular node, since this is as simple as indexing the list or dictionary.
- **Space Complexity**: `O(E)` space where `E` is the number of edges in the graph because each element in the outer array stores a list of edges the node has.
    - In the case of an undirected graph, each edge in the graph is stored twice making the space complexity `O(2E)` which reduces down to `O(E)`.

Adjacency Matrices

- **Determining if Two Nodes Share an Edge**: `O(1)` time complexity. We can find if nodes `i` and `j` share an edge by indexing the matrix `adj_matrix[i][j]`
- **Retrieving A Nodes Neighbors**: `O(N)` where `N` is the number of nodes in the graph. To find a node `i`'s neighbors, we need to iterate over the entire row of the matrix `adj_matrix[i]` to determine if each `adj_matrix[i][j]` has value `1`.
- **Space Complexity**: $O(N^2)$ space (one list of length 2 for each edge) to represent the entire graph.

Bonus Syntax & Concepts

The following concepts are nice to know and may improve your graphs knowledge and help you solve certain problems more easily and efficiently. However, they are not *required* to solve any of the problems in this unit and we recommend mastering BFS and DFS first. There are endless graph algorithms that you can learn and these are only a few. These concepts are **not in scope for either the Standard or Advanced Unit 10 assessments**, and you do not need to memorize them! Click on each concept to read more about how to use it.

- Backtracking is a general algorithmic technique in which you incrementally build partial solutions, and turn around and try alternative solutions once you determine the current solution won't work. This is repeated until a solution is found (or it is determined that there is no solution).
- Topological Sorting is the linear ordering of vertices in a DAG such that for every directed edge `i -> j`, the node `i` comes before node `j`. Commonly used to order tasks or processes that have dependencies.
- Dijkstra's Algorithm finds the shortest path in a *weighted* graph.
- Minimum Spanning Trees is a subset of edges in an undirected, connected, weighted graph that includes all nodes in the graph while minimizing the total edge weight.
- Prim's Algorithm finds the minimum spanning tree of a graph.
- Kruskal's Algorithm also finds the minimum spanning tree of a graph.