Representing Graphs.

Method 1: (Adjacency Matrix)

For an \(n\)-node graph \(G = (V, E)\) (\(n = \# \text{ of nodes}\)), \(m = \# \text{ of edges}\).
Assume \(V = \{1, 2, \ldots, n\}\).

The adjacency matrix of \(G\) is an \(n\times n\) matrix \(A\) where
\[
A[u, v] = \begin{cases} 1 & \text{iff } (u, v) \in E, \text{ or } (u, v) \notin E \end{cases}
\]

\[
A = \begin{bmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}.
\]

This takes \(\Theta(n^2)\) space.

Method 2: (Adjacency List)

For each \(u \in V\), we have an array Adj\([u]\) that contains all nodes \(v\) where \((u, v) \in E\).

\[
\begin{align*}
\text{Adj}[1] &= 2 \\
\text{Adj}[2] &= 1 \quad 3 \\
\text{Adj}[3] &= 2
\end{align*}
\]

Claim: The adjacency list representation takes \(O(n + m)\) space.

Proof: First we need pointers to Adj\([u]\). \(\Rightarrow O(n)\)
Each edge appears in exactly two adj. lists \(\Rightarrow O(m)\)

Implementing BFS/DFS:

**Stack**: First-In-Last-Out

**Queue**: First-In-First-Out

Push: add an element to the data structure (as).
Pop: remove an element.

Stack:

```
2
1
```

Push(3) \rightarrow 3 2

Pop() \rightarrow 2

Queue:

```
1 2
```

Push(3) \rightarrow 1 2 3

Pop() \rightarrow 2 3
Implementing BFS:

BFS(s):

Set discovered[s] = 1 and discovered[v] = 0, \( \forall v \neq s \), queue Q = empty.

Q.push(s) ← Q contains all nodes that are discovered but not explored.

while (Q is not empty)

\( u = Q.pop() \);

For each edge \((u,v)\) incident to \( u \)

If discovered[v] = 0

\( \) discovered[v] = 1

Add edge \((u,v)\) to the BFS tree \( T_B \).

Q.push(v)

End if

End for

End while

How do we iterate over all neighbors of \( u \)?

- If \( G \) is represented by an adjacency matrix \( A \):
  - we loop over \( v = 1, \ldots, n \) and check if \( A[u][v] = 1 \).
- If \( G \) is represented by adjacency lists \( Adj[u] \):
  - we iterate over \( Adj[u] \).

Runtime of BFS:

(Assume \( Q.push(x) \) and \( Q.pop() \) runs in \( O(1) \) time.)

- In either adj. matrix or adj. lists representation,
  - BFS runs in time \( O(n^2) \).
  - Because WHILE loop runs at most \( n \) times, and each for loop runs at most \( n \) times.

- In the adj. lists representation:
  - BFS runs in time \( O(m + n) = O(n^2) \).
  - Because the FOR loop of node \( u \) runs \( |Adj(u)| \) times.
  - Total number of time we run the FOR loop is \( O(\sum \{|Adj(u)|\}) = O(2m) = O(m) \).

Additional \( O(n) \) to set up and manage the length \( n \) array "discovered"