The square of a directed graph $G = (V, E)$ is the graph $G^2 = (V, E^2)$ such that $(u, w) \in E^2$ iff for some $v \in V$, both $(u, v) \in E$ and $(v, w) \in E$; ie. there is a path of exactly two edges.

Give efficient algorithms for both adjacency lists and matrices.

Given an adjacency matrix, we can check in constant time whether a given edge exists. To discover whether there is an edge $(u, w) \in G^2$, for each possible intermediate vertex $v$ we can check whether $(u, v)$ and $(v, w)$ exist in $O(1)$. Since there are at most $n$ intermediate vertices to check, and $n^2$ pairs of vertices to ask about, this takes $O(n^3)$ time.

With adjacency lists, we have a list of all the edges in the
For a given edge \((u, v)\), we can run through all the edges from \(v\) in \(O(n)\) time, and fill the results into an adjacency matrix of \(G^2\), which is initially empty. It takes \(O(mn)\) to construct the edges, and \(O(n^2)\) to initialize and read the adjacency matrix, a total of \(O((n + m)n)\). Since \(n \leq m\) unless the graph is disconnected, this is usually simplified to \(O(mn)\), and is faster than the previous algorithm on sparse graphs.

Why is it called the square of a graph? Because the square of the adjacency matrix is the adjacency matrix of the square! This provides a theoretically faster algorithm.
Traversals Orders

The order we explore the vertices depends upon what kind of data structure is used:

- **Queue** – by storing the vertices in a first-in, first out (FIFO) queue, we explore the oldest unexplored vertices first. Thus our explorations radiate out slowly from the starting vertex, defining a so-called *breadth-first search*.

- **Stack** - by storing the vertices in a last-in, first-out (LIFO) stack, we explore the vertices by lurching along a path, constantly visiting a new neighbor if one is available, and backing up only if we are surrounded by previously discovered vertices. Thus our explorations
quickly wander away from our starting point, defining a so-called *depth-first search*.

The three possible colors of each node reflect if it is unvisited (white), visited but unexplored (grey) or completely explored (black).
Breadth-First Search

BFS(G, s)
for each vertex \( u \in V[G] - \{s\} \) do
  color\([u]\) = white
  \( d[u] = \infty \), ie. the distance from \( s \)
  \( p[u] = NIL \), ie. the parent in the BFS tree
color\([u]\) = grey
\( d[s] = 0 \)
\( p[s] = NIL \)
\( Q = \{s\} \)
while \( Q \neq \emptyset \) do
  \( u = head[Q] \)
  for each \( v \in Adj[u] \) do
if \( \text{color}[v] = \text{white} \) then
\[
\begin{align*}
\text{color}[v] &= \text{gray} \\
\text{d}[v] &= \text{d}[u] + 1 \\
\text{p}[v] &= u \\
\text{enqueue}[Q,v]
\end{align*}
\]
dequeue[Q]
\[
\begin{align*}
\text{color}[u] &= \text{black}
\end{align*}
\]
Depth-First Search

DFS has a neat recursive implementation which eliminates the need to explicitly use a stack. Discovery and final times are sometimes a convenience to maintain.

DFS(G)
for each vertex $u \in V[G]$ do
  $\text{color}[u] = \text{white}$
  $\text{parent}[u] = \text{nil}$
  $\text{time} = 0$

for each vertex $u \in V[G]$ do
  if $\text{color}[u] = \text{white}$ then DFS-VISIT[u]
Initialize each vertex in the main routine, then do a search from each connected component. BFS must also start from a vertex in each component to completely visit the graph.

**DFS-VISIT**[u]

- `color[u] = grey (*u had been white/undiscovered*)`
- `discover[u] = time`
- `time = time + 1`

for each `v ∈ Adj[u]` do

  if `color[v] = white` then
    `parent[v] = u`
    `DFS-VISIT(v)`

- `color[u] = black (*now finished with u*)`
- `finish[u] = time`
- `time = time + 1`
BFS Trees

If BFS is performed on a connected, undirected graph, a tree is defined by the edges involved with the discovery of new nodes:

This tree defines a shortest path from the root to every other
node in the tree.
The proof is by induction on the length of the shortest path from the root:

- \textit{Length} = 1 First step of BFS explores all neighbors of the root. In an unweighted graph one edge must be the shortest path to any node.

- \textit{Length} = s Assume the BFS tree has the shortest paths up to length \( s - 1 \). Any node at a distance of \( s \) will first be discovered by expanding a distance \( s - 1 \) node.
The *key idea about DFS*

A depth-first search of a graph organizes the edges of the graph in a precise way.

In a DFS of an undirected graph, we assign a direction to each edge, from the vertex which discover it:

![Diagram](image)

In a DFS of a directed graph, every edge is either a tree edge or a black edge.
In a DFS of a directed graph, no cross edge goes to a higher numbered or rightward vertex. Thus, no edge from 4 to 5 is possible:
Edge Classification for DFS

What about the other edges in the graph? Where can they go on a search?
Every edge is either:

1. A Tree Edge
2. A Back Edge to an ancestor
3. A Forward Edge to a descendant
4. A Cross Edge to a different node

On any particular DFS or BFS of a directed or undirected
graph, each edge gets classified as one of the above.
DFS Trees

The reason DFS is so important is that it defines a very nice ordering to the edges of the graph.

*In a DFS of an undirected graph, every edge is either a tree edge or a back edge.*

Why? Suppose we have a forward edge. We would have encountered \((4, 1)\) when expanding 4, so this is a back edge.
Suppose we have a cross-edge

When expanding 2, we would discover 5, so the tree would look like:
Paths in search trees

Where is the shortest path in a DFS?

It could use multiple back and tree edges, where BFS only uses tree edges. DFS gives a better approximation of the longest path than BFS.
The BFS tree can have height 1, independent of the length of the longest path.

The DFS must always have height $\geq \log P$, where $P$ is the length of the longest path.