Exercise 1.1. Explain how to use BFS or DFS to detect if a cycle exists in an undirected graph, and find one in O(n + e).

We perform a DFS traversal from a given starting point. Then,

- Keep track of the parent of each node to avoid counting the edge back to the parent as a cycle.
- If we encounter an already visited node that is not the parent of the current node, we found a back edge and confirm that a cycle exists.
- To find the cycle, we can store the path or use a recursion stack to backtrack.

Exercise 1.2. Explain why every nontree edge always point "upwards", i.e., from a root to one of its ancestors. (Hint: The graph is undirected. In particular, an edge can be traversed in either direction. What does this imply for DFS?)

In an undirected graph, DFS classifies edges as either tree edges or back edges. Since when exploring an edge (u, v), if v is already visited and v != parent[u], then v must have been discovered before u in the DFS. That means v is an ancestor of u in the DFS tree.

Since the graph is undirected, the edge (u,v) is the same as (v, u), so the DFS will only encounter it once, when going from the descendant to the already visited ancestor.

Thus, every nontree edge always points from a node to one of its ancestors in the DFS tree.

Exercise 1.3. Why is it that if we don't check the condition j != p, all edges will be marked back edges?

In the DFS code, when exploring an undirected graph, every edge will be considered twice, once in each direction.

If we omit the check j != p, then when we go from a node i to its parent p, we will see that p is already visited (since it's the parent) and mark (i, p) as a back edge. This will happen for every edge, because the parent of every node (except the root) will always be visited when we look at it from the child.

Thus, all edges would be incorrectly marked as back edges. The condition j != p ensures we don't count the edge back to the parent as a back edge.

Exercise 1.4. In the DFS tree:

- **When is an edge a bridge?**

A tree edge (u,v) is a bridge when the subtree at v has no back-edge to u or above.

- **When is a node an articulation point?**

For the root: The root has > 1 child.
For internal nodes: Has a child with no back-edge to an ancestor.

Exercise 1.s. Why is a back edge never a bridge?

In an undirected graph, a bridge is an edge whose removal disconnects the graph. If a back edge is removed, the cycle it was part of still remains connected via the tree edges in that cycle. Therefore, removing a back edge does not increase the number of connected components, which means it isn't a bridge.

Exercise 1.6. Show that a node *a* is an articulation point iff either of the following is true:

- ***a* is not the root of a DFS tree and *a* has a child *b* such that there's no back edge from any descendant of *b* to any proper ancestor of *a***
- ***a* is the root of a DFS tree and *a* has at least two children.**

Let G be a connected undirected graph, and consider its DFS tree.

[Case 1: *a* is not the root]
Suppose *a* has a child *b* such that no descendant of *b* (including *b* itself) has a back edge to a proper ancestor of *a.* Then the only path from the subtree rooted at *b* to the rest of the graph (above *a*) is through the tree edge (a,b).

Removing *a* disconnects *b*'s subtree from the tree, so *a* is an articulation point.

Conversely, if *a* is an articulation point, there must be at least one child *b* whose subtree is disconnected from the rest when *a* is removed. That means no back edge from *b*'s descendants goes above *a.* So the condition holds.

[Case 2: *a* is the root]

Suppose the root has at least two children, then there is no edge between the two subtrees. Removing the root disconnects these subtrees from each other, which means that the root *a* is an articulation point.

Thus, the root is an articulation point iff it has >= 2 children.

Therefore, the two conditions exactly characterize articulation points.

Exercise 2.1. Explain why there are no forward and cross edges in any DFS forest of any undirected graph.

Suppose in the DFS tree, we have an edge (u,v).

A forward edge would require *u* to be an ancestor of *v* and *v* already visited, but in an undirected graph, if *v* is a descendant, then when exploring from *v* earlier, the edge would have been considered and marked as a tree edge.

A cross edge would connect two different branches, but in an undirected graph, such an edge would create a cycle and would have been encountered earlier as a back edge when exploring the first branch.

Thus, in an undirected DFS, only tree edges and back edges exist.

Exercise 2.2. Explain why there can't be any two nodes i and j such that start_time[i] < start_time[j] < finish_time[i] < finish_time[j] in the DFS for both directed and undirected graphs. (In other words, if the visiting intervals of i and j intersect, then one must contain the other.) This fact is important when we deduced that start_time[i] < start_time[j] < finish_time[j] < finish_time[i] in the "forward edge" case above.

In DFS, the **start** and **finish** times have a **nested interval property:**

- When DFS visits a node, it assigns a start time, explore all its descendants recursively, and then assigns a finish time.
- If node j is visited **after** node i starts, but before i finishes, then j must be a descendant of i in the DFS tree.
- As a descendant, j must finish before i finishes because DFS finishes a node only after finishing all its descendants.

Thus, if start[i] < start[j], and start[j] < finish[i], then j is a descendant of i, so finish[j] < finish[i].
The supposed inequality, finish[i] < finish[j] would contradict this nesting property. Hence, such overlapping intervals where one starts inside another but finishes after it are impossible in DFS.

Intervals are either disjoint or nested, never partially overlapping in the crossed way shown.

Exercise 2.3. Show that there is a cycle if and only if there is a back edge in the DFS forest.

- **Hint: Consider the DFS forest, and suppose there are no back edges. Can you somehow find a cycle using only the remaining edge types?**
- **Hint 2: Consider the sequence of finish_times of the nodes you visit while only using forward edges, cross edges, and tree edges.**

[=>] If there is a back edge in the DFS forest, then there is a cycle.

A back edge (u, v) connects u to an ancestor v in the DFS tree. The tree path from v down to u plus the back edge (u,v) forms a cycle.

[<=] If there is a cycle, then there is a back edge in the DFS forest.
Suppose there is a cycle C in the graph. Consider the first node x in C visited by DFS. Let y be the neighbor of x along C visited later in the cycle.

Since x is the first visited in the cycle, y is a descendant of x in the DFS tree, with all other node in the cycle visited after x so they are in x's subtree.

Now, in the cycle, there is an edge from y back to x, closing the cycle. When DFS explores from y, it will see x as already visited and x is an ancestor of y. That edge (y, x) is exactly a back edge. Thus, every cycle contains at least one back edge in the DFS forest.

Therefore, a directed graph has a cycle if and only if its DFS forest contains a back edge.

Exercise 2.4. How do you find a cycle once the algorithm detects that there exists one?

Once a back edge (u,v) is detected, where v is an ancestor of u in the DFS tree, a cycle can be found as follows:

1. Start from u and follow parent pointers up the DFS tree until reaching v.
2. Collect all nodes along this path.
3. Then the cycle is v->(path from v to u)->u-(back edge)->v

This runs in O(cycle length) time and can be done during DFS by storing the recursion stack or parent array.

Exercise 2.s. Prove that Floyd's cycle-finding algorithm correctly computes s_{cycle}, l_{cycle}, and l_{tail}.

Let the functional graph be defined by f: W -> W, starting from *a*.

The sequence is:

$$a,\ b = f(a),\ c = f(b),\ \ldots$$

It eventually becomes periodic: there exist smallest $\mu \geq 0$ (tail length) and $\lambda \geq 1$ (cycle length) such that

$$x_\mu = x_{\mu+\lambda}$$

And for all i >= μ, $x_{i+\lambda} = x_i$.

Floyd's algorithm uses two pointers:

tortoise moves 1 step at a time: $T_k = x_k$

hare moves 2 steps at a time: $H_k = x_{2k}$

Because the sequence enters a cycle of length λ, there exists some k such that $x_k = x_{2k}$.

Once they meet, reset hare to the start *a,* keep tortoise at the meeting point, and move both one step at a time. They will meet again at x_μ, so $l_{tail} = \mu$ and $s_{cycle} = x_\mu$.

We keep tortoise at x_μ, move hare one step at a time, counting steps until hare returns to tortoise. The number of steps is λ.

Therefore, Floyd's algorithm correctly finds the cycle entry, tail length, and cycle length in linear time and constant space.

Exercise 2.6. Prove that Floyd's cycle-finding algorithm takes $O(l_{tail} + l_{cycle})$ time.

Tortoise moves 1 step, hare moves 2 steps per iteration. After k steps: tortoise at x_k, hare at x_{2k}. The worst case is when l_{tail} iz large relative to l_{cycle}. The hare may need $O(l_{tail} + l_{cycle})$ step to meet, because the tortoise needz l_{tail} ztepz to enter the cycle. Once inzide, the distance between them (mod l_{cycle}) reduces by 1 each tep, zo at mozt l_{tail} ztepz. Zo this part take <= $l_{tail} + l_{cycle}$.

Reset hare to start, both move 1 step at a time. They meet exactly after l_{tail} steps. Keep tortoise at meeting point, move hare 1 step at a time around cycle. They will meet again after exactly l_{cycle} step.

Total time: $O(l_{tail} + l_{cycle})$. QED. fuck you

Exercise 2.7. Why does naive BFS/DFS not work for directed graphs?

- They only find one-way reachability, not mutual reachability. The set of nodes visited from a start node is not necessarily an SCC, it may include nodes from other SCCs that are reachable but not mutually connected.
- Different starting orders give different results.

Exercise 2.8. Explain why the SCCs of G are exactly the same as the SCCs of G^R.

Let G^R be the graph obtained by reversing all edges of G.

- Two nodes *u* and *v* are in the same SCC of G iff there is a path from u to v and a path from v to u in G.
- Reversing all edges swaps the directions of all paths. so, a path u -> v in G becomes a path v -> u in G^R, and vice versa.
- Therefore, u can reach v in G and v can reach u in G iff v can reach u in G^R and u can reach v in G^R.

Therefore, the strongly connected components of a graph and its reverse are e8actly the same.

Exercise 2.13. In the pseudocode for Tarjan's algorithm:

- **Is line 11 being run only when i → j a back edge? Is it being run on forward edges as well? What about cross edges?**

```
else if instack[j]:
   low[i] = min(low[i], disc[j])
```

No, it runs on any edge where j is already visited (disc[j] != -1) and j is on the stack (instack[j] true). This includes:

- Back edges (j is an ancestor of i)
- Cross edges to a node in another branch, if that node is still on stack.

- **If yes on either, does that mean Tarjan's algorithm is incorrect? Why or why not?**

No, Tarjan's algorithm remains correct.

- Low[i] tracks the lowest discovery time reachable from i's subtree via any path, including cross edges, as long as the target is still on stack (same partially built SCC).
- If a cross edge goes to a node not on stack, that node is already in a completed SCC, so it's irrelevant to its SCC.
- The condition instack[j] ensures we only consider nodes that could still be part of the same SCC as i.

so updating low[i] via cross-edges to on-stack nodes is necessary for correctly propagating reachability information within the current SCC.

Therefore, this is correct because it properly accounts for all ways to reach an ancestor within the same SCC.

Exercise 2.14. What happens if we move the line instack[j] = False from line 21 to the very end of the DFS function?

In Tarjan's algorithm, instack[v] indicates whether node v is currently in the recursion stack (part of the currently active DFS path and not yet assigned to an SCC.

When an SCC is found (low[i] == disc[i]), all nodes in that SCC are popped from the stack, and instack is set to False for each of them (line 21 in get_SCC).

If we move instack[j] = False to the very end of DFS(i) after returning all children and just before finishing, then it would **break correctness**, the algorithm would no longer correctly identify SCC and could produce merged components. The stack and instack must stay synchronized for Tarjan's algorithm to work.