• **DISCLAIMER:** These notes are not necessarily an accurate representation of what I said during the class. They are mostly what I intend to say, and have not been carefully edited.

• Main topics:
  - Test 2 Review
  - Depth first search
  - Back edges and cut edges.

• Comments from Last Time:
  - More about time complexity of Prim / Dijkstra.
  - Revisit shortest path algorithms.

Last time we introduced the depth first search algorithm for traversing a graph. Its main idea is to greedily explore unvisited parts of the graph. Written recursively it leads to the following algorithm:

```
DFS(v)
1. Iterate through all neighbors of u, v
   (a) If v is unvisited.
      i. Mark v as unvisited.
      ii. DFS(v)
```

This is another way to find connected components. We can show by induction on the path length from u to v that any v reachable from u gets visited during a call of DFS(u).

It gives far more structural information about the graph though: note that as with any other graph traversal algorithms such as BFS, Bellman-Ford, Dijkstra’s, and Prim’s, each node that’s not the starting point gets a parent indicating the vertex that makes it visited. This forms a tree, and the most important fact about DFS on an undirected graph is:

**Lemma 0.1.** All non-tree edges are between a vertex u and one of its ancestor v.
This can be proven considering the first time the DFS reaches one of \( u \) or \( v \). Without loss of generality say it’s \( u \). Then when the for loop iterating through neighbors of \( u \) reaches \( v \), it will be marked as visited. So \( v \) must be contained in the subtree of the DFS tree rooted at \( u \).

Such non-tree edges are referred to as back edges: there is one more type of edge that can occur when one does DFS on directed graphs, which is what the textbook focuses on. This property in turn leads to many highly intricate computational primitives involving the DFS tree. One example is finding cut edges:

**Definition 0.2.** A cut edge \( e = uv \) is an edge whose removal disconnects \( u \) from \( v \).

Clearly such edges can be found in \( O(m^2) \) time by trying to remove all edges in the graph. We can get to \( O(m) \) based on the following two observations:

1. All cut edges must belong to the DFS tree.
2. A tree edge \( uv \) with \( u \) as \( v \)’s parent is a cut edge if and only if there are no edges in \( v \)’s subtree that goes to \( u \) or higher.

So all we need to do is compute for each subtree, the lowest in the DFS tree that a back edge can reach. This value can either be the depth of the other end point, or the discovery time (which was mentioned last time???)

The computation of these values is recursive, and almost like dynamic-programming (which is the focus of the next 3 weeks). For each node, we compute the minimum depth of a back edge in its subtree. Then once we recursively computed these values for all children of a node \( u \), we take the min across them, as well as edges incident to \( u \) to find the value on \( u \). This takes \( O(m) \) time because we only scan through the neighbors of each \( u \) once, and can also be written into the DFS schemes.

2