Topological Sort

Dependency Graph

Vertices are tasks/classes...

Edge $u \rightarrow v$ if $u$ must be finished before starting $v$.

- Undies
- Pants
- Socks
- Shoes

...
Topological ordering

List of vertices s.t. u comes before v if there is an edge u → v.

No ordering is ∀ any cycles.

Directed graph with no cycles, called a directed acyclic graph (DAG)
Can always topological sort on a DAG.

Alg 1:

In-degree (number of incoming edges) of first vertex must be 0.

So start with any in-degree 0 vertex.
If we remove it, we just want a top ordering of what's left.

Could scan for next vertex in $O(n)$ time per vertex.

-or-

Add vertices to a queue whenever their in-degree goes to 0.
void topsort() throws CycleFoundException {
    Queue<Vertex> q = new Queue<Vertex>();
    int counter = 0;

    for each Vertex v {
        if (v.indegree == 0) {
            q.enqueue(v);
        }
    }

    while (!q.isEmpty()) {
        Vertex v = q.dequeue();
        v.topNum = ++counter; // Assign next position in sorted order

        for each Vertex w adjacent to v {
            if (--w.indegree == 0) {
                q.enqueue(w);
            }
        }
    }

    if (counter != NUM_VERTICES) {
        throw new CycleFoundException();
    }
}

if this is true, we have a subgraph of in-degree ≥ 1 vertices. It contains a cycle.
Runs in $O(IV(1V)+1E1)$ time.

Alg 2

Suppose $G$ is a DAG...

$\Rightarrow$ no back edges in a DFS.

Consider any edge $u \rightarrow v$.

$u \rightarrow v$ is not a back edge.

Cases:

If we call dfs(u) before we visit v, we'll call and complete dfs(v) before dfs(u) completes.
If we call dfs(u) after dfs(v) finishes, we finish dfs(u) after dfs(v) finishes.

We won't call dfs(u) while dfs(v) is running, because u\rightarrow v is not a back edge. 

\Rightarrow dfs(v) completes before dfs(u) completes 

So, order vertices in reverse order of when their dfs calls complete.
u comes before v in this order for all edges u ≅ v

⇒ it's a topological order

This one is called a reverse postorder.

(order by start of dfs calls is a preorder)
```java
boolean topsortDFS() throws CycleFoundException {
    int counter = NUM_VERTICES;
    for each Vertex v {
        if (!v.visited) {
            counter = topsortDFS(v, counter);
        }
    }
}

void topsortDFS(Vertex v, int counter) throws CycleFoundException {
    v.visiting = true;
    for each Vertex w adjacent to v {
        if (w.visiting) {
            throw new CycleFoundException();
        } else if (!w.visited) {
            counter = topsortDFS(w, counter);
        }
    }
    v.visiting = false;
    v.visited = true;
    v.topNum = counter--;
    return counter;
}
```

$O((V+E) \times \text{time})$
Shortest paths

Given a path $P$ on an edge-weighted graph.

Its **unweighted length**

is \# edges on $P$.

Its **weighted length** is

sum edge weights along $P$. 

Single source shortest paths.

Given weighted directed graph \( G = (V, E) \) and a designated vertex \( s \in V \) called the source.

What is smallest weighted path length to every other vertex?

Edges might have negative weight!
Today: Non-negative weights.

General strategy:
Assume large distances but decrease them as you find better paths.

When you do so, record previous vertex on that path.
First, assume weights are 1.

broadth-first search:

```java
void unweighted(Vertex s) {
    Queue<Vertex> q = new Queue<Vertex>();
    for each Vertex v {
        v.dist = INFINITY;
    }
    s.dist = 0;
    q.enqueue(s);
    while (!q.isEmpty()) {
        Vertex v = q.dequeue();
        for each Vertex w adjacent to v {
            if (w.dist == INFINITY) {// we haven't found w yet
                w.dist = v.dist + 1;
                w.path = v; // v was the last vertex on the path to w
                // want to search edges of w when we finish with closer vertices
                q.enqueue(w);
            }
        }
    }
}
```

$O(1 + EI)$ time
Dijkstra's algorithm

arbitrary non-negative weights

Queue 6rd.

Instead, explicitly grab closest unexplored vertex

→ code calls this unknown
void dijkstra(Vertex s) {
    for each Vertex v {
        v.dist = INFINITY;
        v.known = false;
    }
    s.dist = 0;

    while (there is an unknown vertex) {
        Vertex v = unknown vertex of smallest distance;
        v.known = true;

        for each Vertex w adjacent to v {
            if (!w.known) {
                // DistType is probably int or double
                DistType cvw = cost of edge from v to w;

                if (v.dist + cvw < w.dist) {
                    // We found a shorter path to w
                    w.dist = v.dist + cvw;
                    w.path = v;
                }
            }
        }
    }
}

Easy! Scan for vertex of smallest distance each iteration.

$O(\left| V \right|^2 + |E|) = O(\left| V \right|^2)$ \times
Min heap to store + pick unknown vertices.
key: distance

$O(|V|)$ inserts
$O(|V|)$ deleteMins
$O(|E|)$ decrease keys

$O(\log |V|)$ per op

so

$O(|V|\log |V| + |E|\log |V|)$

$= O(|E| \log |V|)$ time ✓
Fib. heap:

$O(1)$ amortized for insert + decrease key

$O(\log n)$ \quad \text{for deleteMin}

$\Rightarrow O(\max(1V, 1\log IV) + 1E))$

↑

for deleteMins

Don't actually do this. Use a binary heap.