We are training to be algorithms chefs, not algorithms cooks
SECTION 6.4

Matchings
DEFINITIONS

- A **bipartite graph** is an undirected graph in which the vertices are split into two sets, and all edges go between these sets.
- A **matching** in a bipartite graph is a selection of edges, such that no vertex is connected to more than one of the edges.
- The **size** of a matching is the number of edges it includes.
- A **maximum matching** is one with the largest possible size.

PROBLEM STATEMENT

Given a bipartite graph, find a maximum matching.
0. Given a bipartite graph ...

1. Build a helper graph:
   - add source \( s \) and sink \( t \)
   - add edges from \( s \) and to \( t \)

2. Solve max-flow on the helper graph, to find a maximum flow \( f^* \)

3. Interpret the flow \( f^* \) as a matching

What’s the bug in my thinking?
0. Given a bipartite graph ... 

1. Build a helper graph:
   • add source $s$ and sink $t$
   • add edges from $s$ and to $t$

2. Solve max-flow on the helper graph, to find a maximum flow $f^*$

3. Interpret the flow $f^*$ as a matching

wtf?! This isn't the sort of flow I expected!
As well as specifying the two translations, we also need to prove that this procedure does indeed solve the original problem!
We used the translation strategy for finding the longest common substring using dynamic programming.
There’s a common pattern when applying the translation strategy to optimization problems.

The typical way we prove correctness is ...

CLAIM1. The optimal helper solution does translate into a possible solution to the original problem

CLAIM2. This translation is optimal for the original problem
CLAIM1. The optimal helper solution does translate into a possible solution to the original problem

Ford-Fulkerson will produce an integer flow, since all capacities are integer. Indeed, the flow on each edge must be either 0 or 1:

Thus, the capacity constraints tell us that, when we translate $f^*$ into an edge selection, it meets the definition of “matching”.

CLAIM2. This translation is optimal for the original problem

Suppose not, i.e. suppose the max flow $f^*$ translates to a matching $m^*$, but there exists a larger-size matching $m'$. Note that when we translate matching $\leftrightarrow$ flow in the obvious way,

\[
\text{value(flow)} = \text{size(matching)}
\]

Since $\text{size}(m') > \text{size}(m^*)$, there is a flow $f'$ whose value is strictly greater than the value of $f^*$. But this contradicts optimality of $f^*$. 

CLAIM1. The optimal helper solution *does* translate into a possible solution to the original problem

CLAIM2. This translation is optimal for the original problem

For every problem where you propose using a “Translation” strategy, you have to
- **invent the two translations** (original problem $\rightarrow$ helper problem, helper solution $\rightarrow$ original solution)
- **prove that your translations satisfy these two claims**
Ex5q6. A signal failure can prevent travel in both directions between a pair of adjacent stations. **How many signal failures it would take to prevent travel from Kings Cross to Embankment?**
SECTION 6.7

Topological sort
DEFINITION
Given a directed graph, a **total ordering** is an ordering of the vertices such that if there is an edge $v \rightarrow u$ in the graph, then $v < u$ in the ordering.

PROBLEM STATEMENT
Find a total ordering, if one exists.

This graph has a cycle, so there is no total order possible.
We are training to be algorithms chefs, not algorithms cooks.

These are interesting ideas, worth pursuing. We'll pursue one of them: depth-first search.
def dfs_recurse(g, s):
    for v in g.vertices:
        v.visited = False
    visit(s)

def visit(v):
    v.visited = True
    for w in v.neighbours:
        if not w.visited:
            visit(w)

attempt 1: depth-first search

This might not even visit all vertices, so it might not produce a total order.
def dfs_recurse_all(g):
    for v in g.vertices:
        v.visited = False
    for v in g.vertices:
        if not v.visited:
            visit(v)

def visit(v):
    v.visited = True
    for w in v.neighbours:
        if not w.visited:
            visit(w)

attempt 2: comprehensive depth-first search
attempt 2: comprehensive depth-first search

Some edges point backwards – not a total order.
def dfs_recurse_all(g):
    for v in g.vertices:
        v.visited = False
    for v in g.vertices:
        if not v.visited:
            visit(v)

def visit(v):
    v.visited = True
    for w in v.neighbours:
        if not w.visited:
            visit(w)

attempt 2: comprehensive depth-first search
```python
def toposort(g):
    for v in g.vertices:
        v.visited = False  # v.colour = 'white'
    totalorder = []
    for v in g.vertices:
        if not v.visited:
            visit(v, totalorder)
    return totalorder

def visit(v, totalorder):
    v.visited = True  # v.colour = 'grey'
    for w in v.neighbours:
        if not w.visited:
            visit(w, totalorder)
    totalorder.append(v)  # v.colour = 'black'
```
Correctness theorem.
Given a DAG $g$, this algorithm produces a total order such that for every edge $v_1 \rightarrow v_2$, $v_1$ appears to the right of $v_2$ in total order.

Performance analysis.
It has running time $O(V + E)$, just like depth-first search.

DAG = directed acyclic graph.

We’ve already seen that if there are cycles then it’s impossible for there to be a total order.

The theorem tells us that the converse is also true: if there aren’t any cycles then $\exists$ a total order.
Correctness theorem. Given a DAG $g$, this algorithm returns a totalorder such that for every edge $v_1 \rightarrow v_2$, totalorder has $[\ldots v_2 \ldots v_1 \ldots]$.

Proof. First, the algorithm must terminate (because of how it uses the 'visited' flag.) (We have to prove termination first. If it doesn't terminate, it can't return anything!)

Next, we prove the claim using the “breakpoint” strategy. We'll talk about “vertex colours”, as set in the comments of the code. These colours are a way to express “what has happened in the past” in terms of “colours of the vertices right now”. It's just to save us some circumlocution.

```python
def toposort(g):
    for v in g.vertices:
        v.visited = False
        # v.colour = 'white'
    totalorder = []
    for v in g.vertices:
        if not v.visited:
            visit(v, totalorder)
    return totalorder

def visit(v, totalorder):
    v.visited = True
    # v.colour = 'grey'
    for w in v.neighbours:
        if not w.visited:
            visit(w, totalorder)
    totalorder.append(v)
    # v.colour = 'black'
```
Pick an arbitrary edge \( v_1 \rightarrow v_2 \), and consider the instant that \( v_1 \) got coloured grey. (It must happen at some point in execution). What colour is \( v_2 \)?

- If \( v_2 \) is black: then \( v_2 \) is already in total order, so \( v_2 < v_1 \) \( \checkmark \).
- If \( v_2 \) is white: then \( v_2 \) has not yet been visited. It's a descendant of \( v_1 \), so \( \text{visit}(v_2) \) will be invoked and terminate before \( \text{visit}(v_1) \) terminates, so \( v_2 < v_1 \) in total order \( \checkmark \).
- If \( v_2 \) is grey, then \( \text{visit}(v_2) \) has started but not yet terminated. Therefore \( v_1 \) must be a descendant of \( v_2 \) (and the flame graph tells us a path \( v_2 \rightarrow v_1 \)).

But \( v_1 \rightarrow v_2 \) by assumption, hence there's a cycle, which contradicts our DAG assumption.
Preorders

Definition 139 A preorder \((P, \sqsubseteq)\) consists of a set \(P\) and a relation \(\sqsubseteq\) on \(P\) (i.e. \(\sqsubseteq \in \mathcal{P}(P \times P)\)) satisfying the following two axioms.

- Reflexivity.
  \(\forall x \in P. \ x \sqsubseteq x\)

- Transitivity.
  \(\forall x, y, z \in P. \ (x \sqsubseteq y \land y \sqsubseteq z) \implies x \sqsubseteq z\)

Definition 140 A partial order, or poset\(^a\), is a preorder \((P, \sqsubseteq)\) that further satisfies

- Antisymmetry.
  \(\forall x, y \in P. \ (x \sqsubseteq y \land y \sqsubseteq x) \implies x = y\)

\(^a\)(standing for partially ordered set)

Theorem 141 For \(R \subseteq A \times A\), let

\[ \mathcal{F}_R = \{ Q \subseteq A \times A \mid R \subseteq Q \land Q \text{ is a preorder} \} . \]

Then, (i) \(R^{\text{pos}} \in \mathcal{F}_R\) and (ii) \(R^{\text{pos}} \subseteq \cap \mathcal{F}_R\). Hence, \(R^{\text{pos}} = \cap \mathcal{F}_R\).

Let \(x \sqsubseteq y\) mean “\(y\) depends on \(x\)”. This is a partial order (and the theorem explains why partial orders correspond to directed acyclic graphs).

Might this lead to an efficient algorithm? If we have \(V\) vertices \(\text{\# items to be sorted}\), and \(E\) edges \(\text{\# relations}\),

- sorting algorithms are \(O(V^2)\) or \(O(V \log V)\)
- DFS-based topsort is \(O(V + E)\)
- \(E \leq V^2\)

So, on highly connected graphs, sorting algorithms might do better.

IDEA. Think through all our sorting algorithms, and see if they can be adapted to work with partial orders.