Section 1: Solutions

1. Write it Better: A Proof by Contradiction

What follows is a **correct**, but poorly presented, proof by contradiction. We'll show two different ways to make it cleaner.

Claim: For every simple graph G, if every node of G has degree at least 2, then G has a cycle.

An unclear proof. Suppose, for the sake of contradiction, there is a graph G such that every node of G has degree at least 2, but G has no cycle.

We will construct a simple path in G. Start at some node v_0, of G. Follow v_0 along an edge to find v_1. Now, since v_1 (like every other node) has degree at-least 2, there is another edge attached to v_1. Follow it to a vertex v_2. If v_2 is a vertex we have already visited (i.e., v_0), then we have found a cycle, a contradiction! Otherwise, from v_2, we may repeat the same argument. Continue from v_2 (also degree at-least-2) to a vertex v_3. If v_3 is a repeat (v_0 or v_1) then we have a contradiction! Otherwise continue finding $v_4, v_5, ...$ The graph is finite, so we cannot continue this process forever. Eventually we find a repeated vertex, which means we have a cycle, a contradiction! □

(a) It's common in proofs by contradiction to have cases like we've seen here "Option A: we're done with our proof! Option B: do something else." Here, though, that "do something else" has us basically where we started (a new end-vertex on our path where we've used one edge), and it's tempting to say "repeat indefinitely, eventually you hit the other case." That's mathematically correct! But not particularly elegant. And you have to write down enough steps that your reader knows what the pattern is, which could be a lot. The more elegant version is to use "proof by contradiction with extremality" Instead of slowly building an object (here, the path), just start with the most *extreme* version of the object at the beginning (usually the biggest one or the first one). Starting with the right object lets us eliminate Option A and jump right to option B.[1]

Let's see if this proof is any cleaner. We've set you up with the extreme path, finish the proof.

Proof. Suppose, for the sake of contradiction, there is a graph G such that every node of G has degree at least 2, but G has no cycle. Let $P = v_0, v_1, ..., v_k$ be a longest simple path in G. □

Solution:

Since v_0 has degree at-least 2, it must have another neighbor, w other than v_1. Since P is a longest simple path, w must be a repeat of a vertex among $v_1, ..., v_k$ (otherwise $w, v_0, v_1, ...v_k$ would be a longer simple path). Combining the edge (v_0, w) with P from v_0 to w gives a cycle. But G was a cyclic, that's a contradiction!

(b) There's another style yellow-flag in this proof. We're proving an implication and our contradiction was the negation of one of the two things we supposed at the start. That usually means that proof by contrapositive would be clearer. Try writing this proof by contrapositive. **Solution:**

Proof. We argue by contrapositive. That is, we will show for all graphs, G, that if G does not have a cycle, then it has a node of degree at-most 1.

Let G be an arbitrary acyclic graph, and consider a maximal path $v_0, v_1, ..., v_k$. Since the graph is acyclic, v_0 cannot be adjacent to $v_2, ..., v_k$. Since the path is maximal, v_0 cannot be adjacent to any vertex other than the other v_i (as that would lead to a larger path). Thus v_0 is adjacent only to v_1, so it has degree at-most 1 as required. □

[1] Don't be intimidated by how superficially different the start looks. We haven't changed the idea of the proof – this is just a proof-writing trick! We'll use this trick **a lot** when we get to greedy algorithms.

With both of these alterations, we have a proof that will be much clearer to our readers. Notice, though, the core idea is identical in all three proofs: if everything is degree-two-or-more you could keep discovering new vertices, but the graph is finite so you can't do that. All three proofs are just different ways of presenting that core idea.

2. Find The Bug: Spoof By Induction

What follows is an **incorrect** proof by induction.

Claim: Every (undirected) tree with at least three nodes has at least two nodes of degree-one.

Spoof. Let $P(n)$ be "Every tree with at least n nodes has at least two nodes of degree-one."

Base Case: $n = 3$. There is only one undirected tree with three nodes. It has two nodes of degree-one.

Inductive Hypothesis: Suppose $P(n)$ holds for $n = 3, ..., k$ for an arbitrary $k \geq 3$.

Inductive Step: let T be an arbitrary tree with k nodes. By inductive hypothesis, T has at least two nodes of degree-one. Call them u and v.

We now build T', which has $k + 1$ nodes. Take T, and create a new node w. Since we are interested in connected trees, we must connect w; we break into cases depending on what it is adjacent to.

Case 1: w is adjacent to neither u nor v
If w is adjacent to a node other than u, v then u and v still have degree-one, so the claim holds on T'.

Case 2: w is adjacent to one of u, v but not the other
If w is adjacent to u or v, then the other of u, v and w will both be degree-one

Case 3: w is adjacent to both u, v
This case is impossible! If w were adjacent to both u and v, then the path in T between u and v (which exists because T was connected) along with (u, w) and (v, w) form a cycle, which is not allowed in a tree.

In all (allowed) cases, T' has the required degree-one vertices. Since we constructed T' to have $k + 1$ vertices, we have shown $P(k + 1)$. □

(a) Clearly describe the bug and why the proof is incorrect. **Solution:**

> We never introduce an arbitrary tree with $k+1$ nodes in the inductive step! Why should our reader believe we have handled all possibilities between our three cases? Notice that we don't have a recursive definition of "tree" – we aren't doing structural induction here! We need to start with an arbitrary tree with $k + 1$ nodes. That's how you show a statement of the form "for all trees with $k + 1$ nodes..."
>
> It turns out that we really have addressed all cases here – every tree really can be built with these rules – but without a recursive definition, we'd need to write a detailed proof to explain that every tree really can be built that way first. It's not worth it. When proving a for-all statement by induction **always** start with the big thing and find the smaller thing inside.

(b) What is the correct "skeleton" of the inductive step (i.e., the right things to assume and the right target)? **Solution:**

> We must start with "Let T' be an arbitrary tree with $k + 1$ nodes."
>
> Our conclusion will be that T' has at least two nodes of degree-one, so $P(k + 1)$ holds.

(c) Is the claim true? If so, write a correct proof. If not, provide a counter-example. You may use the fact "every tree has at least one node of degree-one" (this fact is just the Claim from Problem 1 in contrapositive form). **Solution:**

Proof. The proof is identical except for the IS:

Inductive Step: Let T' be an arbitrary tree with $k+1$ nodes. Let u be a vertex of T' of degree-one (this first vertex exists by the fact), and call its neighbor v. Let T'' be the tree created by deleting u from T'.

Observe that, since u was degree-one, the only simple paths that used (u, v) had u as an endpoint (as once we use (u, v) to arrive at/leave u we cannot reuse it to leave/arrive). Thus T'' is still a connected tree, and we can apply the IH to T'' to conclude there are at least two vertices w_1, w_2 of T'' that are degree-one.

We now find the two degree-one nodes in the original tree T'. We know that u has degree-one (and is not the same as w_1 or w_2 since u was deleted to create T''). Since u has degree-one, it can only attach to one of w_1, w_2, thus at least one (the other one) of w_1, w_2 is an additional node of degree-one, as required. □

3. Find the bug: More Failed Induction

In this problem you will fix an incorrect induction proof.

Let's do a little bit of problem setup. Suppose you have a stable matching instance with n horses and n riders. Of the n horses, 5 of the horses are **popular**. That is, every rider's list has those 5 horses as their first 5 choices (in some order, not necessarily the same for each rider). Similarly, you have 5 **popular** riders, such that every horse has those 5 riders as their top choices.

Let $P(n)$ be "In every stable matching instance with n horses, n riders, of which 5 horses and 5 riders are popular: in every stable matching, popular horses are matched only to popular riders."

Spoof. We will show $P(n)$ holds for all $n \geq 5$ by induction on n.

Base Case $(n = 5)$
With 5 horses and riders, every horse and rider is popular. Since every stable matching pairs every agent, every agent is matched to a popular agent.

Inductive Hypothesis: Suppose $P(n)$ holds for $n = 5, ..., k$ for an arbitrary integer $k \geq 5$.

Inductive Step: Let $h_1, ..., h_k$, $r_1, ..., r_k$ be k horses and riders, with $h_1, ..., h_5, r_1, ..., r_5$ being the popular agents. We add agents h_{k+1} and r_{k+1}. By popularity, h_{k+1} has $r_1, ..., r_5$ (in some order) as its 5 favorite agents and r_{k+1} has $h_1, ..., h_5$ (in some order) as their 5 favorite agents. Further, let r_{k+1} and h_{k+1} be each other's 6^{th} choices (i.e. top choice outside the popular riders.

Now, consider any stable matching in the old (size-k) instance, and create a stable matching for the new instance by pairing h_{k+1} with r_{k+1}.

We now show that this matching is stable for the new instance. Since it was stable for the small instance, the only possible blocking pairs must involve h_{k+1} or r_{k+1}. By IH, every popular agent is matched to another popular agent. Regardless of where h_{k+1} and r_{k+1} was added to the popular agent's list, they fall after the popular agents, so h_{k+1} and r_{k+1} cannot form a blocking pair with the popular agents. And since they have each other as their next choices, they cannot form a blocking pair with anyone else. Thus we have that there are no blocking pairs, and the matching is stable. The popular agents remain matched to each other, as required. □

(a) There are at least two errors in this proof. Describe them! **Solution:**

The first mistake is in the setup of the inductive step. We need to show a claim for every instance of size $k+1$. Instead we build a particular instance of size $k+1$. In this example, the mistake is quite fundamental – there is no reason in the problem that r_{k+1} and h_{k+1} should have each other as their 6^{th} choices. That is, if we introduced a recursive definition of stable matching instances and changed this to structural induction, we would have more steps to do beyond the one here (In contrast to the tree problem where all the cases are actually handled).

The second bug is again a mistake with handling a for-all. This time, the quantifier on the "every stable matching" part of the statement. We don't check every stable matching! We check every matching we built

by starting with a stable matching on the small instance – how do we know there aren't stable matchings where r_{k+1} is matched to h_4 (for example)? We need to start with an arbitrary stable matching and argue whether the popular agents are matched or not.

(b) Write a correct proof of this claim. Do NOT use induction. Use a proof by contradiction instead. **Solution:**

Suppose, for the sake of contradiction, that there is a stable matching instance with 5 popular riders and horses, and there is a stable matching M for this instance so that some popular horse h is not matched to a popular rider. Since there are the same number of popular horses and riders, there is a popular rider, r, that is also not matched to a non-popular agent. We claim that r and h form a blocking pair. Indeed, since each is popular, they are each in the top 5 of each other's lists, but each is matched to a non-popular agent, which must be 6^{th} or lower on both lists. Thus r and h would rather be with each other than with their matches, so they form a blocking pair. But M was supposed to be a stable matching. A contradiction! So every popular horse must be matched to a popular rider.

4. Proof Practice: Proving Code Correct

Recall Dijkstra's Algorithm from 332.

```
Dijkstra(Graph G, Vertex source)
     initialize distances to ∞, source.dist to 0
     mark all vertices unprocessed
     initialize MPQ as a Min Priority Queue
     add source at priority 0
     while(MPQ is not empty){
          u = MPQ.removeMin()
          foreach(edge (u,v) leaving u){
               if(u.dist+weight(u,v) < v.dist){
                    if(v.dist == ∞) //if v not in MPQ
                         MPQ.insert(v, u.dist+weight(u,v))
                    else
                         MPQ.decreaseKey(v, u.dist+weight(u,v))
                         v.dist = u.dist+weight(u,v)
                         v.predecessor = u
               }
          }
          mark u as processed
   }
```

Consider the following claim:

For all n, the n^{th} time a vertex, v, is removed from `MPQ`, `v.dist` contains the true shortest path distance from `source` to v.

(a) Prove the claim by induction. **Solution:**

We prove the claim by induction on n.

Base Case: $n = 0$, before any iterations, `source` is the closest vertex (at distance 0 – it is the closest, because all edge weights are non-negative). The distance from `source` to itself is 0, as stored.

IH: Suppose for the first k iterations, at the top of the while loop, the vertex removed from `MPQ` had the correct distance.

IS: Now consider the closest vertex at the $(k+1)^{\text{st}}$ iteration, `u`. The shortest path from `source` to `u` visits a

vertex w right before u.

w is already processed (it is closer to source than v, because there are no negative edge weights) – when it was, by IH the distance from source was correct. We then updated u.dist with the distance from source to w plus the edge weight from w to u. Observe that this is the length of the path from source to u. We will not update u.dist again, because if we ever see another edge ending at u, the source of that edge is a vertex being processed, and thus Dijkstra's knows the correct distance to that vertex (and would be updating with the distance along the path through the other vertex). Thus when u is closest at the start of the $k+1)^{\text{st}}$ iteration, we have the true shortest path distance.

thus by the principle of induction, every vertex removed from MPQ had the true distance when it was removed.

(b) Prove the claim by contradiction. **Hint:** It's *extreme*ly helpful to think about the closest vertex where something is wrong, rather than just any old iteration.
During your proof, you may use the following fact without proof: Every non-infinite value of v.dist is the length of a path from v.dist (not necessarily the shortest path). **Solution:**

Suppose for the sake of contradiction, that there is a vertex which, when removed from MPQ has the wrong distance, and let u be the closest such vertex (in true distance terms). Let v be the vertex before u on the shortest path from source to u. Since v.dist only decreases, and is always the length of a source-to-v path, we note that v.dist must have been more than its true value. Thus u (which, since edge weights are positive, has a smaller true distance than the distance from u to v) had a smaller value of dist when processed than v.dist did. So, v must have been processed first (since we are going in decreasing order of dist). Since u was the first vertex with an incorrect distance, u.dist had the correct value when processed. But since v had the correct distance when processed, v would have updated v.dist to u.dist $+ w(u, v)$. But that's exactly the distance from source to u. So u had the correct distance value in it. Since there are no negative length edges and every non-infinite value of dist is a true path, we will never change the value again, so it must have been correct when processed. A contradiction!

Thus the distance is always correct when removed from the priority queue.

5. Gale-Shapley

Consider the following stable matching instance:

$r_1 : h_3, h_1, h_2, h_4$
$r_2 : h_2, h_1, h_4, h_3$
$r_3 : h_2, h_3, h_1, h_4$
$r_4 : h_3, h_4, h_1, h_2$

$h_1 : r_4, r_1, r_3, r_2$
$h_2 : r_1, r_3, r_2, r_4$
$h_3 : r_1, r_3, r_4, r_2$
$h_4 : r_3, r_1, r_2, r_4$

(a) Run the Gale-Shapley Algorithm with riders proposing on the instance above. When choosing which free rider to propose next, always choose the one with the smallest index (e.g., if r_1 and r_2 are both free, always choose r_1).

Solution:

The steps of the Gale-Shapley Algorithm with the rider with lowest index proposing first:

r_1 chooses h_3	(r_1, h_3)
r_2 chooses h_2	$(r_1, h_3), (r_2, h_2)$
r_3 chooses h_2	$(r_1, h_3), (r_3, h_2)$
r_2 chooses h_1	$(r_1, h_3), (r_2, h_1), (r_3, h_2)$
r_4 chooses h_3	$(r_1, h_3), (r_2, h_1), (r_3, h_2)$
r_4 chooses h_4	$(r_1, h_3), (r_2, h_1), (r_3, h_2), (r_4, h_4)$

(b) Run the Gale-Shapley Algorithm with riders proposing on the same instance. But now, when choosing which free rider to propose next, always choose the one with the largest index. Do you get the same result?

Solution:

The steps of the Gale-Shapley Algorithm with the rider with highest index proposing first:

r_4 chooses h_3	(r_4, h_3)
r_3 chooses h_2	$(r_3, h_2), (r_4, h_3)$
r_2 chooses h_2	$(r_3, h_2), (r_4, h_3)$
r_2 chooses h_1	$(r_2, h_1), (r_3, h_2), (r_4, h_3)$
r_1 chooses h_3	$(r_1, h_3), (r_2, h_1), (r_3, h_2)$
r_4 chooses h_4	$(r_1, h_3), (r_2, h_1), (r_3, h_2), (r_4, h_4)$

We ended up with the same result!

(c) Now run the algorithm with horses proposing, breaking ties by taking the free horse with the smallest index. Do you get the same result?

Solution:

The steps of the Gale-Shapley Algorithm with horses proposing:

h_1 chooses r_4	(r_4, h_1)
h_2 chooses r_1	$(r_1, h_2), (r_4, h_1)$
h_3 chooses r_1	$(r_1, h_3), (r_4, h_1)$
h_2 chooses r_3	$(r_1, h_3), (r_3, h_2), (r_4, h_1)$
h_4 chooses r_3	$(r_1, h_3), (r_3, h_2), (r_4, h_1)$
h_4 chooses r_1	$(r_1, h_3), (r_3, h_2), (r_4, h_1)$
h_4 chooses r_2	$(r_1, h_3), (r_2, h_4), (r_3, h_2), (r_4, h_1)$

No, the result is different when we have the horses propose as opposed to the riders.

6. A Quick Proof

Is it possible to have a stable matching instance with more than 2 stable matchings? If so, give an instance and at least 3 stable matchings. If not, prove that every instance has at most 2 stable matchings.

Solution:

Consider the following instance:

$r_1 : h_1, h_2, h_3, h_4$
$r_2 : h_2, h_1, h_4, h_3$
$r_3 : h_3, h_4, h_1, h_2$
$r_4 : h_4, h_3, h_2, h_1$

$h_1 : r_2, r_1, r_4, r_3$
$h_2 : r_1, r_2, r_3, r_4$
$h_3 : r_4, r_3, r_2, r_1$
$h_4 : r_3, r_4, r_1, r_2$

This instance has four stable matchings:
$(r_1, h_1), (r_2, h_2), (r_3, h_3), (r_4, h_4)$
$(r_1, h_1), (r_2, h_2), (r_3, h_4), (r_4, h_3)$
$(r_1, h_2), (r_2, h_1), (r_3, h_3), (r_4, h_4)$
$(r_1, h_2), (r_2, h_1), (r_3, h_4), (r_4, h_3)$