Section 2: Solutions

1. Big-$\mathcal{O}$-No

Put these functions in increasing order. That is, if f comes before g in the list, it must be the case that $f(n)$ is $\mathcal{O}(g(n))$. Additionally, if there are any pairs such that $f(n)$ is $\Theta(g(n))$, mark those pairs.

- $2^{\log(n)}$
- $2^{n \log n}$
- $\log(\log(n))$
- $2^{\sqrt{n}}$
- $3^{\sqrt{n}}$
- $\log(n)$
- $\log(n^2)$
- $\sqrt{n}$
- $(\log(n))^2$

Hint: A useful trick in these problems is to know that since $\log(\cdot)$ is an increasing function, if $f(n)$ is $O(g(n))$, then $\log(f(n))$ is $O(\log(g(n))$. But be careful! Since $\log(\cdot)$ makes functions much smaller it can obscure differences between functions. For example, even though n^3 is less than n^4, $\log(n^3)$ and $\log(n^4)$ are big-Θ of each other.

Solution:

(a) $\log(\log(n))$

(b) $\log(n)$

(c) $\log(n^2)$ This function is $\Theta(\log(n))$. They differ only by the constant factor 2.

(d) $(\log(n))^2$

(e) $\sqrt{n}$

(f) $2^{\log(n)}$ Note that this is just n.

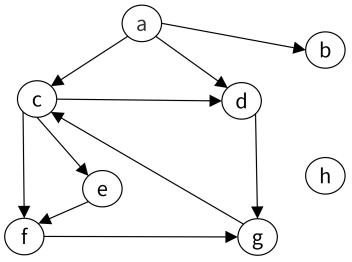
(g) $2^{\sqrt{n}}$

(h) $3^{\sqrt{n}}$

(i) $2^{n \log n}$

2. Mechanical: DFS

Run Depth-First-Search on the graph below to classify the edges. Mark the start and end times for each vertex.



Solution:

Tree Edges: (a,b), (a,c), (c,d), (d,g), (c,e), (f,g)
Back Edge: (g,c)
Forward Edges: (a,d), (c,f)
Cross Edges: (f,g)

3. Another DFS application

In an *undirected* graph, call an edge $e = (u, v)$ a **cut edge** if u and v cannot reach each other in $G \setminus e$.

Intuitively, a cut edge is one which, when "cut" (i.e., removed) causes the graph to have more components than it did before. Cut edges are also called *bridges*.

We'll adapt DFS to find all the cut edges in a graph.

3.1. Setting the Foundation

Usually the first step in designing an algorithm is making absolutely sure you know what you're trying to find! That is, to make sure you really understand the problem. And secondly, to try to get some intuition for what the thing you're looking for "usually" looks like. The best way to do both of these steps is to look at a few examples. We'll do that before we get to designing the algorithm.

(a) We're in an undirected graph now, so edge classification will be different from lecture. Look back at the edge classification for DFS for directed graphs. Convince yourself that it's not possible to have a cross edge in a DFS on an undirected graph. (1-3 sentences of intuition, not a proof).

(b) Convince yourself it is not possible to have a forward edge in a DFS on an undirected graph. (2-3 sentences of intuition, not a proof).

(c) NOW, let's try to recognize cut edges. Prove that a cut edge must be a tree edge.

(d) Draw a graph, and a corresponding DFS, which has at least one tree edge that is not a cut edge, and at least one tree edge that is a cut edge.

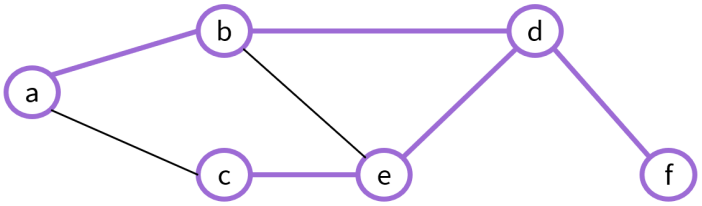
Solution:

(a) For (u, v) to be a cross-edge, we have to discover v first then get to u. But in an undirected graph, when we process v, we can view (v, u) as an edge and will discover u through it.

(b) Similarly, for (u, v) to be a forward edge, we must discover u, and (using some other edge(s)) discover v, then return to u and process the edge (u, v). In an undirected graph, when we found v, we would process all its edges, including (u, v) before popping it off the stack, so we can't get back to u to process.

(c) Let $e = (u, v)$ be a cut edge. Without loss of generality, let u be the first of u and v that DFS processes. When u is processed, it stays on the stack until all edges, including (u, v) are processed. (u, v) will be a tree edge unless DFS discovers v before (u, v) is processed. To discover v, we would need to follow a walk from u to v. If such a walk existed, then (u, v) would not be a cut edge, as u would still be connected to v. Thus we cannot discover v, and when (u, v) is processed, it will be a tree edge.

(d) There are many different examples you can come up with, but here is an example undirected graph with the corresponding DFS.



The edge (d,f) is a cut edge, while the other tree edges (a,b), (b,d), (d,e), and (e,c) are not.

3.2. Building The Bridges

Now, we're ready to start designing an algorithm.

(a) Prove that an edge is a cut edge if and only if it is not part of a simple cycle. **Solution:**

Forward direction: By contrapositive (part of a simple cycle $\rightarrow$ cut edge)
Suppose $e = (v_1, v_2)$ is part of a simple cycle $v_1, v_2, ..., v_k$. Every walk that uses e can have e replaced with $v_1, v_k, v_{k-1}, ...v_2$ and still be a valid walk (i.e., a trip "the long way" around the cycle), so the connected components of G cannot change when e is deleted.

Backward direction:
by contrapositive (not cut edge $\rightarrow$ not part of simple cycle) Suppose, for the sake of contradiction, that $e = (u, v)$ is not part of a simple cycle and (u, v) is not a cut edge. Then u and v must be able to reach each other in $G - e$. Let $v, w_1, ..., w_k, u$ be a (simple) path from v to u in $G - e$. Then adding e would give a simple cycle involving e in G, a contradiction.

(b) Define $\texttt{low}(u)$ to be the smallest `pre` number that can be reached by: starting at u, following any number of tree edges, and at most one back edge. Show that if (u, v) is a tree edge, discovered going from u to v, and (u, v) is part of a cycle, then $\texttt{low}(v) \leq \texttt{u.pre}$. (Notice we're comparing v's low to u's pre!) **Solution:**

(u, v) was a tree edge, so v was a new discovery, and there will be a DFS call starting from v made immediately. By the discovery property, v will discover every undiscovered vertex reachable from v before coming off the stack. Because (u, v) is part of a cycle, that must include returning to some already visited vertex in the cycle (possibly u itself, or possibly a predecessor of v). The `pre` number for that vertex is less than the `pre` for v since it was discovered first. Thus we have $\texttt{low}(v) \leq \texttt{u.pre}$

(c) Show that if (u, v) is a tree edge, discovered going from u to v, and (u, v) is not part of a cycle then $\texttt{u.pre} < \texttt{low}(v)$. **Solution:**

Since (u, v) is not part of a cycle, there is no way to return from u. Thus no back edge can reach any ancestor of u (if one did, combining with tree edges would give a cycle). We therefore have that $\texttt{low}(v)$ being at most $v.\texttt{pre}$. Since u discovered v, $u.\texttt{pre} < v.\texttt{pre} \leq \texttt{low}(v)$.

(d) Now, modify DFS to detect every cut edge of an undirected graph. **Solution:**

We just need to add some bookkeeping to keep track of low numbers.

```
DFS_Cut_Edge(u)
    Mark u as "seen"
    u.start = counter++
```

```
u.low = u.start
For each edge (u,v) //leaving u
    If v is not “seen”
        DFS(v)
        If(v.start < u.low)
            u.low = v.start
        Else
            mark (u,v) as a cut edge
        End If
    Else
        if(v.low < u.low)
            u.low = v.low
        End If
    End If
End For
u.end = counter++
```

4. Graph Modeling

In this problem we're going to solve a classic riddle.

(a) First, you should solve the classic riddle yourself to get a feel for the problem.

You are on the beach with a jug that holds exactly 5 gallons, a jug that holds exactly 3 gallons, and a large bucket. Your goal is to put **exactly** 4 gallons of water into the bucket. Unfortunately, the jugs are not graduated (e.g., you can't just fill the larger jug $4/5$ full). What you can do are the following operations.

- Completely fill any of your jugs.
- Pour from one of your containers into another until the first container is empty or the second is full.
- Pour out all the remaining water in a container.

How do you get 4 gallons of water into the bucket?

Solution:

Fill the 5 gallon jug, pour into the 3 gallon jug until it is full (the jugs now contain 2 and 3 gallons respectively). Pour from the larger jug into the large bucket (it now has 2 gallons). Empty the 3 gallon jug, and repeat all these steps to get the desired 4 gallons.

Alternatively, Fill the 3 gallon jug, pour into the 5 gallon jug. Fill the 3 gallon jug again, pour until the 5 gallon jug is full (they now contain 5 gallons and 1 gallon respectively) . Pour the 1 gallon from the 3 gallon jug into the bucket. Refill the 3 gallon jug and pour into the bucket to bring the total to 4 gallons.

There may be other solutions.

(b) Now, let's write an algorithm to solve any instance of this puzzle. You are given a list of 10 jugs with (positive integer) capacities $c_1, ..., c_{10}$, ranging from 1 to C. Your goal is to determine whether it is possible to get exactly t gallons into the bucket.

Solution:

Intuition
The "state" of the puzzle can be represented as the number of gallons in each of the jugs and the bucket. We encode the rules of the puzzle such that each possible step is an edge.

Algorithm
Let S be the set of all 11-tuples, where for the first 10 entries, the entry is an integer between 0 and c_i, and the final entry is an integer between 0 and t. There are $(C+1)^1 0 \cdot (t+1)$ such states.

We make a graph with a vertex for every element of S. And add an edge from u to v if and only if the states meet one of these conditions.

From a given state $(j_1, j_2, ..., b)$, you can move to another state $(j'_1, j'_2, .., j'_{10}, b')$ if and only if one of the following hold:

- There is only one index, k, where the tuples differ, and $j'_k = 0$. (we emptied a jug or the bucket)
- There is only one index, k, ($k \leq 10$) where the tuples differ, and $j_k = c_k$. (we filled a jug)
- There are two indices, k, ℓ where the tuples differ:
  - $j'_k = 0$ or $j'_\ell = c_\ell$
  - $j_k + j_{ell} = j'_k + j'_\ell$.

Finally, we add a target vertex z, to the graph. Add an edge from every tuple where $b = t$ to z.

We then use [B/D]FS, starting from the all 0's tuple, and searching to see if z is reachable. If it is, we can return true (and predecessor edges will show the steps to take). If t is not reachable, then return false.

Correctness
Suppose our algorithm returns true. Then [B/D]FS found a walk from all 0's to z. By construction of the graph, each edge corresponds to a valid rule we can apply in the original puzzle. Since the only edges going into t are from states where $b = t$, the walk must reach such a state, thus the puzzle can be solved by doing the steps on the edges of that walk.

Conversely, suppose the puzzle is solvable. Then there is a series of steps that can be taken to put t gallons into the bucket. Each legal step has a corresponding edge in the graph to the next state by construction, so there is a path to a valid stopping state (i.e., a tuple where $b = t$), we added an edge from all such vertices to z, so there is a path from all 0's to z. [B/D]FS will discover this path, so we will return true.

Running Time
Let $n = (t+1)\prod(c_i + 1)$. Note that this is the number of vertices in our graph. Each vertex has a constant number of edges leaving it (as you are performing one of three operations (dump, pour, fill) among a constant number of jugs. So the graph can be has $\mathcal{O}(n)$ edges and can be constructed in $O(n)$ time. Running [B/D]FS in a graph with $\mathcal{O}(n)$ vertices and edges takes $\mathcal{O}(n)$ time, so the overall running time is $\mathcal{O}(n)$.

5. Judging Books by Their Covers

You have a large collection of books, and just got a new bookshelf. For aesthetic reasons, you're going to arrange your books by the color of their covers (not by author or subject). You wish to put only books of a single color on any given shelf. You have a list of pairs of books which you know to be the same color. This list might be only partial (it's possible that u, v, and w are all the same color, but your list might only have "u and v are the same color. w and v are the same color.", for example). You should assume that the "same color" relation is transitive.

Given your list, your job is to give an upper-bound on the number of shelves you need so that no shelf has more than one color of book. Describe an algorithm to give the best bound you can on the number of shelves needed.

You do not need a full proof of correctness, but you should describe the running time in terms of (whichever subset is appropriate): b, the number of books; p the number of pairs listed; s the number of shelves required (i.e., your final answer). **Solution:**

We'll make a graph as follows: have a vertex for every book and an edge between u and v if they are listed as a "same color" pair. Since "same color" is transitive, if there is a walk from u to v then u and v are the same color. Thus any subset of a connected component can be on its own shelf (and we can't put books from separate

connected components on the same shelf).

Our algorithm can run [B/D]FS to find connected components, and then iterate through the vertices to count the number. The graph will have b vertices and p edges, so the running time is $\mathcal{O}(b+p)$.