Please enter your name and uID below.

Name: Sabath Rodriguez

uID: u1254500

Submission notes

- **(PS 1 specific) This is a warm-up assignment. Full points will be given for answers that have some correct elements and that clearly been given a legitimate effort.**

- *Solutions must be typeset* using one of the template files. For each problem, your answer must fit in the space provided (e.g. not spill onto the next page) *without* space-saving tricks like font/margin/line spacing changes.
- Upload a PDF version of your completed problem set to Gradescope.
- Teaching staff reserve the right to request original source/tex files during the semester, so please retain the original files.
- Please remember that for problem sets, collaboration with other students must be limited to a high-level discussion of solution strategies. If you do collaborate with other students in this way, you must identify the students and describe the nature of the collaboration. You are not allowed to create a group solution, and all work that you hand in must be written in your own words. Do not base your solution on any other written solution, regardless of the source.

1. (Alternative Topological Sorting)

   a. Since our first-vertex or the leftmost vertex in our topologically sorted array has to be a source, and the reason why it has to be a source is because it has the largest post-order time, which means that it has no incoming edges, then we can infer that using some version of DFS, we will come back to this node and finish after visiting and finishing every other possible vertex in our graph.
   b. First, we know that we should only look at sources and not sinks, because we know that if it's a sink then it will therefore have a source that will finish after that sink and therefore have a larger post-order time.
   c. The possible in-degrees from our 2nd element in our topologically sorted array is 1. To clarify, the MAX is 1. This is because we know that we only have 1 possible source and we can only have one edge connecting u->v, or to that element in our array.
   d. If our right most vertex in our topologically sorted array removed all out-edges or our-degrees, then our 2nd most right element could have a max in-degree of 0. If we don't have an edge outgoing from our 1st element which should have the highest possible out-edges, then there is no way that our 2nd left most element can have any possible in-edges. To clarify, since our 2nd most left element can at max have 1 incoming edge from out left-most vertex, then if we remove all out-edges from our left-most element, then our 2nd element cannot have any in-edges. These answers hint at an algorithm for constructing a topological sort starting from the first vertex (instead of starting from the last vertex). Use the hints from the questions above, and briefly describe an approach for topological sorting that fills in the sorted list from left to right (starting with the first vertex in the sort, ending with the last vertex in the sort).

My algorithm:

We would want to iterate through $v \in V$ and find the v with the smallest in-degree, then remove it from out graph, add it to our array starting from the left, then repeat until there are no more vertices in our graph. This will fill our array from right to left in a topological order. (pseudocode in next page if needed for clarification)

2. (Analysis)
   I would first like to show my algorithm using pseudocode and then describe and argue for it's complexity:

```
i = 0
A = []
While |V| > 0
        Sort G by in-degree, which will take a minimum of n*ln(n) time, in ascending
        order
        t = remove from V at index 0
        arr[i] = t
        i += 1
return A
```

We know that we will iterate through our array of vertices, of size V, so we will iterate through our array |V| times and we will perform *n*lg(n)* work at every iteration, therefore my algorithm should be upper bounded by $O(|V|^2 \ln(|V|))$.

3. (Cut Vertices)

If we only have 1 path from v->...->t, such that $v \notin \{s, t\}$, or if we have a forward edge or a tree edge from v to t and it's the only connection we have from v->...->t then it's a cut vertex.

```
FindCut(G)
        cutVList = []
        for v in V
                if v ∉ {s, t},
                        temp = v.to-edges
                        v.to-edges = []
                        TopOrderG)
                        v.to-edges = temp
                        if if G.V.getT.status ≠ finished
                                cutVList.add(v)
        return cutVList

TopOrder(G)
        for v in V
                v.status = new
        for v in V
                if v.status = new
                        TopOrderDFS(v)

TopOrderDFS(v)
        v.status = active
        for v->w in v.to-edges
                if w.status = new
                        TopOrderDFS(w)
                else if w.status = active
                        return
        v.status = finished
```

This is a brute force approach, where we remove all valid vertices from our graph G and then run some DFS to see if t is still reachable, I decided to go with the Topological order algorithm provided in the textbook. We remove all to-edges from a vertex, then run topOrder on G and check if t is *finished,* this just means that t was reachable from *s,* if ut wasn't reachable then we know that v is a cut-vertex and we will then add it to our cutVList, and finally we will return our cutVList once we have tried all valid vertices from G.