COMS 311: Homework 3
Due: April 12^{th}, 11:59pm
Total Points: 40

Late submission policy. Any assignment submission that is late by not more than two business days from the deadline will be accepted with 20% penalty for each business day. That is, if a homework is due on Friday at 11:59 PM, then a Monday submission gets 20% penalty and a Tuesday submission gets another 20% penalty. After Tuesday no late submissions are accepted.

Submission format. Homework solutions will have to be typed. You can use word, LaTeX, or any other type-setting tool to type your solution. Your submission file should be in pdf format. Do **NOT** submit a photocopy of handwritten homework except for diagrams that can be hand-drawn and scanned. We reserve the right **NOT** to grade homework that does not follow the formatting requirements. Name your submission file: `<Your-net-id>-311-hw3.pdf`. For instance, if your netid is `asterix`, then your submission file will be named `asterix-311-hw3.pdf`. Each student must hand in their own assignment. If you discussed the homework or solutions with others, a list of collaborators must be included with each submission. Each of the collaborators has to write the solutions in their own words (copies are not allowed).

General Requirements

- When proofs are required, do your best to make them both clear and rigorous. Even when proofs are not required, you should justify your answers and explain your work.
- When asked to present a construction, you should show the correctness of the construction.

Some Useful (in)equalities

- $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}$
- $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}$
- $2^{\log_2 n} = n$, $a^{\log_b n} = n^{\log_b a}$, $n^{n/2} \leq n! \leq n^n$, $\log x^a = a \log x$
- $\log(a \times b) = \log a + \log b$, $\log(a/b) = \log a - \log b$
- $a + ar + ar^2 + ... + ar^{n-1} = \frac{a(r^n-1)}{r-1}$
- $1 + \frac{1}{2} + \frac{1}{2^2} + ... + \frac{1}{2^n} = 2(1 - \frac{1}{2^{n+1}})$
- $1 + 2 + 4 + ... + 2^n = 2^{n+1} - 1$

1. (**10 pts**) In a small town, there is a group $P = \{p_1, p_2, \ldots, p_n\}$ of people who either trust or distrust each other. An $n \times n$ matrix T captures the trust information such that for any $i, j \in \{1, \ldots, n\}$ with $i \neq j$, $T[i][j] = 1$ if p_i trusts p_j and $T[i][j] = 0$ otherwise. Additionally, we set $T[i][i] = 0$ for all $i \in \{1, \ldots, n\}$. A trustworthy person is someone everyone in the town trusts but who trusts no one.

   Write an efficient algorithm that takes T as input and returns the index of a trustworthy person who trusts no one but everyone trusts him/her, or returns -1 if there is no trustworthy person. Your algorithm should be faster than $O(n^2)$. Analyze the runtime of your algorithm using Big-O notation.

   Hint: There can be at most one trustworthy person.

```
int i, j = 0
while (i < n AND j < n):
    if(i == j OR T[i][j] == 1):
        i ++
    else:
        j++
for(int k = 0; k < n; k++):
    if(T[i][k] != 0 OR (T[k][i] AND i != k)):
        return -1
return i
```

   This algorithm runs in O(n) time. The while loop will run a max of i + j times, which both cannot exceed n. The operations inside will take constant time, meaning this branch takes 2n operations, which is O(n). The for loop also runs in O(n) time. It takes exactly n operations to double check that the person we suspect is trustworthy is actually trustworthy.

   This sums up into 3n execution time, which is O(n).

2. (**10 pts**) You are on a treasure hunt adventure in a magical land filled with islands and bridges. Let L denote the set of islands and $B \subseteq L \times L$ denote the set of bi-directional bridges. For any $l, l' \in L$, one can move from island l to l' if and only if there is a bridge between them, i.e., $(l, l') \in B$. Note that bi-directional bridges mean that the connections between islands allow movement in both directions, i.e., for any $l, l' \in L$, $(l, l') \in B$ if and only if $(l', l) \in B$.

   The treasure is hidden on an island $l_{treasure} \in L$ that is unknown to you. You are allowed to pick any island $l_{init} \in L$ as a starting point. To maximize the chance of finding the treasure, not knowing its location, you should pick l_{init} in such a way that you can reach as many islands as possible.

   Write an efficient algorithm takes L and B as input and returns an optimal starting island l_{init} and the set of islands that can be reached starting from your chosen l_{init}. Analyze the runtime of your algorithm using Big-O notation.

```
findBestIsland(L, B):
    reachable = {}
    for each vertex v in L:
        v.explored = false

    for each v in L:
        if(!v.explored):
            currReach = BFShelper(v)
            if(currReach.length > reachable.length):
                reachable = currReach
    return reachable[0], reachable

BFShelper(Node v) returns Set:
    arr = []
    Queue Q
    Q.add(v)
    arr.add(v)
    v.explored = true

    while (!Q.isEmpty):
        u = Q.pop
        for all neighbors w of u:
            if(!arr.contains(w)):
                Q.add(w)
                arr.add(w)
                w.explored = true
    return arr
```

As we know, BFS is O($|V|$ + O($|E|$)). Since our BFS only adds constant operations, it also runs in O($|V|$ + O($|E|$)) time.

The findBestIsland function takes $2|V|$ operations. Both for loops will run once per vertex in the graph. The first for loop only contains constant time operations, but the second contains the call to BFShelper. This means this runs in O($|V|$($|V|$+$|E|$)), or O($|V^2| + |V * E|$).

This will usually run in less time, however. BFShelper is only ran if v is not explored, meaning the runtime will get worse the more connected components there are, with worst runtime being $|V|$ connected components and no edges. This is a very unlikely case, resulting in an algorithm that will run faster than O($|V^2| + |V * E|$) most of the time.

3. (**10 pts**) You are designing the curriculum for a set of courses, each with prerequisites. Additionally, some courses must be taken simultaneously, meaning that students cannot split them across different semesters. Your goal is to determine if it's possible to schedule the courses in a way that satisfies the prerequisites and accommodates the simultaneous course requirements.

   Let C denote the set of courses. Let $Pre : C \to 2^C$ represent the prerequisites, i.e., for each course $c \in C$, $Pre(c) \subseteq C$ is the set of courses that must be taken before c. Finally, $S \subseteq 2^C$ is the set where each element of S is a set of courses that must be taken simultaneously. Write an efficient algorithm that takes C, Pre, and S as input and returns `true` if it is possible to schedule the courses in a way that satisfies the prerequisites and accommodates the simultaneous course requirements and returns `false` otherwise. Analyze the runtime of your algorithm using Big-O notation.

   Example: Consider the set of courses $C = \{c_1, c_2, c_3, c_4, c_5, c_6\}$ and

   - $S = \{\{c_1, c_2\}, \{c_4, c_5\}\}$,
   - $Pre(c_1) = Pre(c_2) = \emptyset$,
   - $Pre(c_3) = \{c_2\}$,
   - $Pre(c_4) = \{c_3\}$,
   - $Pre(c_5) = \{c_2\}$, and
   - $Pre(c_6) = \{c_3, c_5\}$.

   The set S indicates that courses c_1 and c_2 need to be taken simultaneously (i.e., in the same semester). Additionally, courses c_4 and c_5 need to be taken simultaneously. Pre indicates that course c_2 is a prerequisite for c_3; thus, c_2 must be taken before c_3 can be taken. Additionally, Course c_3 is a prerequisite for c_4 and course c_2 is a prerequisite for course c_5. Finally, courses c_3 and c_5 are prerequisites for c_6.

   It is possible to schedule the courses to satisfy the prerequisites and accommodate the simultaneous course requirements as follows. In the first semester, take c_1 and c_2 since both of these courses have no prerequisite and they have to be taken simultaneously. In the second semester, take course c_3. Note that c_5 cannot be taken in this semester because even if it only has c_2 as a prerequisite and c_2 is already taken in the first semester, c_5 needs to be taken simultaneously with c_4, which has c_3 as a prerequisite. In the third semester, take c_4 and c_5, which have to be taken simultaneously. Finally, in the fourth semester, take c_6.

   Note that if $\{c_2, c_4\} \in S$, then it won't be possible to schedule the courses. This is because according to the prerequisite Pre, c_2 needs to be taken before c_3 and c_3 needs to be taken before c_4. Thus, c_2 and c_4 cannot be taken simultaneously.

```
// this is the main function this will return
// the boolean value indicating if a schedule is possible
checkIfPossible(C, Pre, S):

    // set up graph
    G, dict = MakeGraph(C,Pre)
    G, dict, hasCycle = mergeCoReqs(G, S, dict)
    if hasCycle:
        return false

    // make all nodes unexplored
    for all node in G:
        node.explored = false

    // iterate through nodes, checking for cycles and
    // adding them to the proper sets
    for all node in G:
        if node.explored == false:
            create Set visiting
            create set visited
            hasCycle = cycleDetection(dict, node, visited, visiting)
            if hasCycle:
                return false

    return true

// graph construction helper function
makeGraph(Set C, Pre):

        // create graph, add nodes to dicitonary
        Create Graph G
        Create dictionary dict
        // add all classes to graph
        For class in C:
                Add new node n to G
                Add node n to node dict with a key of the class name
                    so {classname: node object}
                N.prereqs = prereqs(class)

        // add all prereq edges
        For node n in G:
                For prereq in n.prereqs:
```

```
                        prereqnode = dict.get(prereq)\\gets the node object
                        Create edge from prereqnode to n
                        Prereqnode.outedges.append(edge)
                        n.recievingedges.append(edge)

        return G, dict


// simple cycle detection algorithm
cycleDetection(Dictionary dict, Node n, Set visited, Set visiting):

    // base cases
    if n in visited:
        return false

    if n in visiting:
        return true

    visiting.add(n)

    // check for back or cross edges
    for edge in node.outgoing:
        if cycleDetection(dict, edge.end, visited, visiting):
            return true

        visiting.remove(n)
        visited.add(n)
        n.explored = True

    return false


// merge the corequisite classes into a single node
mergeCoReqs(Graph G, Set S, Dictionary dict):

        hasCycle = false

        For set in S:
                newNode
                newOutgoing = new set
                newReceiving = new set
```

```
            For class in set:
                   Classnode = dict.get(class)
                    // before merge is complete, make sure there
                    // is not a cycle within merging nodes
                   for prereq in prereqs:
                          if set.contains(prereq):
                                 hasCycle = true

                   newOutgoing += Classnode.outedges
                   newReceiving += Classnode.recievingedges
                   dict[class] = newNode
                   Remove classnode from graph

            For edge in newOutgoing:
                   Edge = new edge from new node to edge.end

            For edge in newRecieving:
                   Edge = new edge from edge.start to newNode

            Add newNode to G
            Newnode.outgoing = newOutgoing
            Newnode.recieving = newRecieving

      return G, dict, hasCycle
```

The runtime of the function checkIfPossible is O($|V| + |E|$). Excluding the function calls, this function is O($|V|$), as it runs a $|V|$ length loop twice. It also calls the functions makeGraph, mergeCoReqs and cycleDetection.

The makeGraph function will run in O($|C| + |prereqs|$) time. It contains two loops that will run C and prereqs times respectively. However, in our graph, vertexes are classes and edges are pre reqs, so this actually is running in O($|V| + |E|$) time.

The mergeCoReqs function has a nested loop, the outside taking $|S|$ iterations and the inner taking $|set|$ time. However, $|S|$ * $|set|$ should be equal to $|V|$, because we can't have new classes beyond those in the course work. Therefore this nested loop pair will run a maximum of $|V|$ times, making mergeCoReqs O($|V|$).

The cycleDetection function utilizes a depth-first search (DFS) algorithm to detect cycles within the graph. In the worst case, it might traverse each edge and node once. Since it's a simple DFS, its time complexity is O($|V| + |E|$), where $|V|$ is the number of vertices (nodes) in the graph, and $|E|$ is the number of edges.

This means checkIfPossible runs in O($|V|$) and calls the functions makeGraph, mergeCoReqs and cycleDetection, which run in O($|V| + |E|$), O($|V|$) and O($|V| + |E|$) time resepectively. This means our final runtime is 4 * $|V|$ + 2 * $|E|$, which is O($|V| + |E|$).

4. (**10 pts**) Let us revisit Problem 3. Write an efficient algorithm to determine the minimum number of semesters required to complete the curriculum while satisfying prerequisites and simultaneous course requirements. Analyze the runtime of your algorithm using Big-O notation.

   Hint: Consider a graph $G = (V, E)$. Let $L : V \to \mathbb{N}$ denote a function such that for any vertex $v \in V$, $L(v)$ is the length of the longest path in G that ends at v. A key observation to solve this problem is that for any given vertex v, we have

$$L(v) = \begin{cases} 0 & \text{if there is no incoming edge to } v \\ \max\limits_{v' \in V \text{ s.t. } (v',v) \in E} L(v') + 1 & \text{otherwise} \end{cases}$$

```
Main(C, Pre, S):

    // if a schedule is impossible, return 0
    // use problem 3 function to check this
    if (checkIfPossible(C, Pre, S):
        return 0

    // setup, using problem 3's helper to make the graph
    Graph G = MakeGraph(C, Pre)
    min = 0

    for each vertex v in G:
        if (Pre(v) == null):
            temp = findDFSLongest(G, n)
        if (temp > min):
            min = temp

    return min

findDFSLongest(G, n):
    count = 0
    Stack S
    S.add(n)
    while(!S.isEmpty):
        temp = S.pop
        for each neighbor u of temp:
            S.push(u)
        count++
    return count
```

This will run in O($|V|$ * $|V|+|E|$) worst case, but this is highly unlikely. The loop will once per class with no prereqs. Most schedules will contain one to two classes like this per component, so it should run closer to O($|V| + |E|$), which is the runtime of DFS, most of the time. The larger the connected components, the faster this algorithm will run.