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.

   The problem is about identifying a universal sink, i.e., an index $k \in \{1, \ldots, n\}$ such that row k of T contains only 0 and column k of T contains only 1, except for $T[k][k]$.

```
          k
 +---+---+---+---+---+
 |   |   | 1 |   |   |
 +---+---+---+---+---+
 |   |   | 1 |   |   |
 +---+---+---+---+---+
k| 0 | 0 | 0 | 0 | 0 |
 +---+---+---+---+---+
 |   |   | 1 |   |   |
 +---+---+---+---+---+
 |   |   | 1 |   |   |
 +---+---+---+---+---+
```

   The first key observation to solve this problem is that there is at most one universal sink. Another key observation is that for any $i, j \in \{1, \ldots, n\}$, $T[i][j]$ is either 0 or 1. If $T[i][j] = 0$ and $i \neq j$, then p_j cannot be the universal sink. On the other hand, if $T[i][j] = 1$, then p_i cannot be the universal sink. So we construct a loop that keeps incrementing either the row index i or the column index j depending on whether $T[i][j] = 0$, in which case we increment j, or $T[i][j] = 1$, in which case we increment i. We quit the loop when either $i > n$ or $j > n$.

   Suppose that the universal sink p_k exists. Consider 2 possibilities.

   (a) We reach column k before row k. So we have, $j = k$ and $i < k$. As a result, we have $T[i][j] = 1$, so we keep incrementing i until $i = k$. At this point, we have $T[i][j] = T[k][k] = 0$, so we increment j. As $T[k][j] = 0$ for all $j > k$, we keep incrementing j until $j > n$.

   (b) We reach row k before column k or simultaneously. So we have $i = k$ and $j \leq k$. As a result, we have $T[i][j] = 0$, so we keep incrementing j until $j > n$.

Note that in both cases, $i = k \leq n$ at the end of the loop. So we can conclude that if a sink exists, then it must be p_i. We then check whether p_i is actually the universal sink, i.e., row i contains only 0 and column i contains only 1 except for $T[i][i]$. Note that we need to perform this check because we do not know what our algorithm actually does if the sink does not exist.

```
i=1; j=1;
while (i<n and j<n)
  if (T[i][j] == 1) // we need to look at the next row
     i++;
  else
     j++;           // we need to look at the next column
// end of while

// If i > n, then there cannot be a universal sink
if i > n
  return -1;

// Check that i is actually the universal sink
for (j = 1 to n)
  // If row i contains 1, there cannot be a universal sink
  if (T[i][j] == 1)
    return -1;
  // If column i contains 0 (except T[i][i]),
  // there cannot be a universal sink
  if (j != i and T[j][i] == 0)
    return -1;

// At this point, we have verified that row i contains only 0
// and column i contains only 1 except for T[i][i].
// So p_i is the trustworthy person who trusts no one.
return i;
```

The runtime of each loop is $O(n)$ so the total time complexity is also $O(n)$.

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.

Consider a undirected graph $G = (V, E)$ with V and E corresponding to islands and bridges, respectively. An optimal starting island l_{init} is simply any island that is in the largest connected component of G. So we compute all the connected components and find the largest one. First, we pick an arbitrary vertex $v \in V$ and perform DFS starting from v, keeping track of visited vertices and the number of vertices visited by DFS, which is the size of the connected component containing v. We then keep applying DFS for unvisited vertices until all the vertices are visited.

Let $L = \{l_1, \dots, l_n\}$, where n is the number of islands. We first convert B to an adjacency list `adj` such that for any $i \in \{1, \dots, n\}$, `adj`$[i]$ is the array of indices of islands connected by a bridge with l_i. This construction takes $O(|L| + |B|)$ time. Our algorithm below then only takes `adj` as input and returns the index of an optimal starting island.

```
int get_optimal_island(adj)
  max_size = 0;
  optimal_island = 1;
  for (i = 1 to n)
    visited[i] = false;

  for (i = 1 to n)
    if (visited[i] == false)
      // Run DFS starting from i to get the connected component containing i.
      // Our version of DFS returns the size of this connected component.
      size = DFS(adj, i, visited);
      if (size > max_size)
        optimal_island = i;
        max_size = size;

  return optimal_island;
```

Here, we slightly modify the standard recursive version of DFS to update the `visited` variable as well as return the number of vertices reachable from a given starting vertex.

```
int DFS(adj, i, visited)
  visited[i] = true;
  size = 1;
```

```
for (j in adj[i])
  if (visited[j] == false)
    size = size + DFS(adj, j, visited);
return size;
```

The time complexity of `get_optimal_island` is $O(|V| + |E|)$, which is $O(|L| + |B|)$. So the overall complexity, including the construction of `adj`, is $O(|L| + |B|)$.

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.

We will construct a graph $G = (V, E)$ and represent E by an adjacency list `adj`. For each set $s_i \in S$ of courses that must be taken simultaneously, construct a vertex v_i to represent this set of courses. So, we have $V = \{v_1, \dots, v_n\}$, where $n = |S|$. Add `j` to `adj[i]` (i.e., a directed edge from v_i to v_j) if for some course $c \in s_j$, $Pre(c) \cap s_i \neq \emptyset$, i.e., a course in s_i is a prerequisite of a course in s_j. In other words, an incoming edge means that another set of courses must be taken before this one. Based on this construction, we get $|V| = |S| \leq |C|$ is the number of sets of courses that must be taken simultaneously and $|E| \leq \sum_{c \in C} Pre(c)$ is the number of prerequisites among the sets of courses. The runtime of this construction depends on the data structure to keep the information about the set of courses corresponding to each vertex. For example, we can keep an array `vertex` of length $|C|$ such that `vertex[i]` is the index of the vertex containing course c_i. In this case, the construction of `vertex` takes time $O(|C|)$ and allow us to obtain a vertex containing any given course in $O(1)$ time when constructing `adj`. Building G then takes $O(|C| + \sum_{c \in C} Pre(c))$ time.

Now, we have a graph that represents the dependencies of courses. For the curriculum to be possible, the graph must not contain a cycle, i.e., it must be a DAG (directed acyclic graph). A key observation here is that there is a cycle in the graph if only if the graph contains a back edge, i.e., a vertex that points to one of its ancestors. Thus, it is possible to schedule the courses in a way that satisfies the prerequisites and accommodates the simultaneous course requirements if and only if no backedge exists.

With this, we will modify DFS to detect back edges.

```
bool find_cycle(adj)
  for (i = 1 to n)
    visited[i] = false;
    stack[i] = false;

  // Loop through all the vertices to make sure that
  // if there are multiple DFS trees, then we check all of them.
  for (i = 1 to n)
    if (visited[i] == false and dfs(adj, i, visited, stack) == true)
      return true;

  return false;
```

Here, we slightly modify the standard recursive version of DFS to keep track of the current recursion stack, which is essentially the path being traced. If we reach a node with an outgoing edge that points back to a node already in the recursion stack, then there is a cycle in the graph.

```
bool dfs(adj, i, visited, stack)
  if (visited[i] == true)
    return false;

  visited[i] = true;
  stack[i] = true;
  for (j in adj[i])
    if (stack[j] == true)
      return true;
    if (visited[j] == false and DFS(adj, j, visited, stack) == true)
      return true;

  // Unmark the node from the recursion stack since it is no longer
  // part of the path being traced.
  stack[i] = false;
  return false;
```

The time complexity of `find_cycle` is $O(|V| + |E|)$. Since $|V| \leq |C|$ and $|E| \leq \sum_{c \in C} Pre(c)$, we can conclude that the overall runtime, including the construction of the graph is $O(|C| + \sum_{c \in C} Pre(c))$.

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

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

Consider an arbitrary course $c \in C$. Let $v_i \in V$ denote the vertex that contains this course, i.e., $c \in s_i$. The number of semesters required to complete c as well as the necessary courses that must be taken before c is the longest path in G ending at v_i. Thus, the minimum number of semesters required to complete the curriculum is the length of the longest path in G.

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

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

To compute $L(v)$, we perform the following steps.

(a) Initialize an array `L` of length $|V|$, where `L[i]` represents the length of the longest path ending at vertex v_i, by setting

$$\texttt{L[i]} = \begin{cases} 0 & \text{if there is no incoming edge to } v_i \\ -\infty & \text{otherwise} \end{cases}$$

The time complexity for this step is $O(|V|)$, assuming that checking whether there is an incoming edge to each vertex v_i takes a constant time.

(b) Perform a topological sort of G to obtain a linear order of sets of courses that satisfies the prerequisites and accommodates the simultaneous course requirements. The time complexity for this step is $O(|V| + |E|)$.

(c) Iterate through the vertices in the topologically sorted order, and for each vertex v_i, update `L[j]` for all j such that $(v_i, v_j) \in E$ by setting

$$\texttt{L[j]} = \max\{\texttt{L[j]}, \texttt{L[i]} + 1\}$$

The time complexity for this step is $O(|V| + |E|)$.

(d) The minimum number of semesters required to complete the curriculum is the longest path in G, which is the maximum value in `L` $+ 1$. This computation takes time $O(|V|)$.

```
// Suppose the we are given the graph G = (V, E) constructed in problem 3

findMinSem(G):
    initialize an array L with size |V|
    for each vertex v_i in V:
        if v_i has no incoming edges:
            L[i] = 0
        else:
            L[i] = -infinity
    order = TopologicalSort(G)  // Get linear ordering of vertices
    for each vertex v_i in order:  // Iterate vertices in topological order
        for each (v_i, v_j) in E:
            L[j] = max{L[j], L[i] + 1}

    return max{L} + 1
```

The overall runtime is, therefore, $O(|V| + |E|)$.