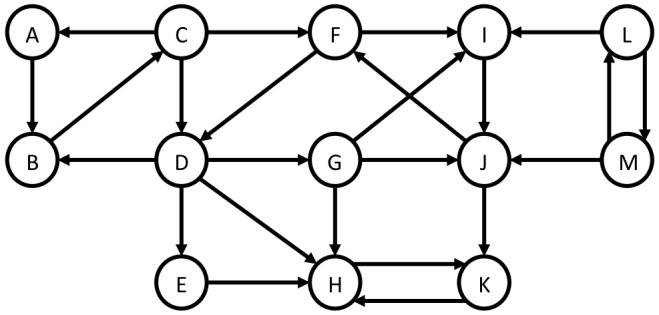
There are 3 questions for a total of 20 points.

---

1. (5 points) (a) For any $n > 3$ and any directed graph G with n vertices, if the number of edges in G is n, then the number of strongly connected components of G is strictly less than nnn.

   Answer: False

   (b) How many SCCs does this graph have?



   Answer: 4

   (c) Give all the vertices that are in the strongly connected component that contains the vertex D in the graph given.

   Answer: A, B, C, D, F, G, I, J

   (d) For any directed acyclic graph G that has a unique source vertex, this source vertex is the first vertex in every topological ordering of vertices of G.

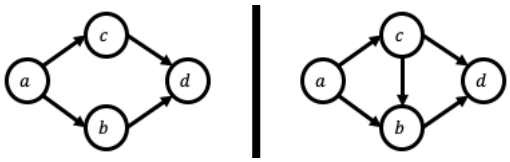
   Answer: True

   (e) Let G be an undirected graph on n nodes where n is an even number. If every node of G has degree $\geq n/2$, then G is strongly connected.

   Answer: True

2. (7 points) You are given directed acyclic graph $G = (V, E)$ and you want to determine if G has a unique topological ordering (i.e., whether there is exactly one topological ordering of the vertices).

   (For example, the DAG on the left does not have a unique topological ordering since both a, b, c, d and a, c, b, d are valid topological orderings. However, the DAG on the right has a unique topological ordering a, c, b, d.)



   Assume that the DAG is given in adjacency list representation. Design an algorithm for this problem. Your algorithm should output "yes" if G has a unique topological ordering and "no" otherwise. Give running time analysis and proof of correctness for your algorithm.

[Points distribution: Algorithm (3 points), proof of correctness (3 points), running time analysis (1 points)]

Solution: Algorithm:

```
function HasUniqueTopologicalOrder(G = (V, E))
    if |V| = 0 then
        output "yes"
    end if
    initialize array in[1 ... |V|] with 0s
    for v ∈ V do
        for (v, u) ∈ E do
            in[u] := in[u] + 1
        end for
    end for
    order := Linearize(G)
    sources := 0
    for v ∈ V do
        if in[v] = 0 then
            sources := sources + 1
        end if
    end for
    if sources > 1 then
        output "no"
        return
    end if
    for i ∈ 1..|V| do:
        sources := sources − 1
        v := order[i]
        for (v, u) ∈ E do
            in[u] := in[u] − 1
            if in[u] = 0 then
                sources := sources + 1
            end if
        end for
    end for
    if sources > 1 then
        output "no"
        return
    end if
    output "yes"
    return
end function
```

Proof of correctness:

Note that the empty graph vacuously has a unique topological ordering. Henceforth we assume G is non-empty.

Firstly we claim that after the first loop, $in[v]$ contains the indegree of vertex v.

Proof: We iterate over all possible edges in the first loop (all edges have been covered since each edge has a starting vertex, which is iterated over since it is in V, and they are covered exactly once since no edge has two starting vertices), and for each edge with the ending vertex as v, we increment

$in[v]$. This implies, given the initial value of $in[v]$ to be 0, that $in[v]$ finally contains the number of incoming edges to v, which is the indegree of v.

Then we claim that after the second loop, *sources* contains the number of sources of the graph.

Proof: We iterate over all vertices and increment *sources* by 1 iff the indegree of the vertex is 0, and thus *sources* is precisely the number of vertices with indegree 0, which are precisely all the sources of the graph by definition.

Note that if for a graph, there are at least 2 sources, say u, v, then we can come up with a topological order with the first two vertices as u, v and v, u (as both are sources) and the rest of the order being the same, which implies that there are at least 2 topological orderings of the graph.

Hence for a graph to have a unique topological ordering, it is necessary that there is exactly one source.

Now we claim that if we remove the unique source from G and the remaining graph G' also has a unique topological ordering, then G also has a unique topological ordering, and vice versa.

Proof:

(forward direction) Suppose not. Then there are at least 2 topological orderings of G, and since it has a unique source, the first element in each ordering is the unique source. If we remove the first element of both the orderings, then we claim that the remaining orderings are topological orderings for G', which will give a contradiction. To prove this claim, we do the following. Note that by the definition of topological ordering, for any edge (u, v) in G, u is to the left of v in the ordering. Consider any edge (u, v) in G'. Since (u, v) is in G, and u is not the source of G, both u and v are in the new ordering and u is to the left of v in the new ordering. Thus the new ordering (which is obtained by removing the unique source of G) is also a valid topological ordering for G' which proves the subclaim. As we had noted earlier, this implies that G' has two distinct topological orderings if we remove the source of G from the two distinct topological orderings of G, and this is a contradiction. Hence there can only be at most 1 topological ordering of G, and since G is a DAG, it has a unique topological ordering.

(backward direction) If G' has at least two topological orderings, by an argument similar to the above, we can see that if we insert the unique source in the beginning of any ordering for G' we come up with a valid topological order of G, so we get at least two topological orderings of G, which is a contradiction.

Now we claim that after the i^{th} iteration in the final loop, either we have returned or $in[]$ contains the indegrees of all vertices in the graph G_i obtained by removing vertices $1, \ldots, i$ from G.

Proof: Induction on i.

Base case: $i = 0$: this is vacuously true.

Inductive hypothesis: $P(i)$: after the ith iteration in the final loop, either we have returned or $in[]$ contains the indegrees of all vertices in the graph G_i.

Inductive step: Suppose $P(i)$ is true. We need to show $P(i + 1)$.

If we have returned already, we are done. Otherwise, note that $in[v]$ = indegree of v in the graph G_i before this iteration. When we remove the vertex $order[i]$, note that it is a source of the graph since the remaining ordering is a valid topological ordering of G_{i+1}. Hence all edges that need to be removed are those emanating from $order[i]$, and thus for each neighbour u of $order[i]$ (which is to its right in the topological ordering since it is a source), we decrement $in[u]$. After we have done this, we can see that this preserves the property that $in[u]$ has the indegree of u in G_{i+1} since either it is adjacent to $order[i]$ or it isn't, and in the case it is, we decrement it by 1, and in the case it isn't, we don't. Hence $P(i + 1)$ is true, and the induction is complete.

Now we claim that after the i^{th} iteration in the final loop, either we have returned or sources contains the number of sources in G_i.

Proof: Induction on i.

Base case: $i = 0$: follows from the previous claim regarding sources.

Inductive hypothesis: after the i^{th} iteration in the final loop, either we have returned or sources contains the number of sources in G_i.

Inductive step:

Suppose $P(i)$ is true. We need to show $P(i+1)$. Note that when we remove $order[i]$ we need to decrement sources by 1 since $order[i]$ was a source in G_i but is not a vertex in G_{i+1}. Now we increment it by 1 for each vertex whose new indegree is 0 after removing this from the graph (the validity of new indegree follows from the previous claim), i.e., each new source in the graph. Hence sources is the number of the sources in G_{i+1} and $P(i+1)$ is proved.

Now we show that if we output "no", G doesn't have a unique topological ordering, and if we output "yes", then G has a unique topological ordering.

Proof:

Suppose we output "no". Then after some i^{th} iteration, the number of sources is > 1 (the iteration where we return). This implies that G_i has at least 2 topological orderings. Now we show using induction that G has at least 2 topological orderings.

Proof:

Inductive hypothesis: $P(k) : G_{i-k}$ has at least 2 topological orderings for $k \leq i$.

Base case: for $k = 0$, this is true by assumption.

Inductive step: suppose $P(k)$ is true for some $k < i$, we need to show $P(k+1)$ is true.

Note that G_{i-k} has at least 2 topological orderings, so G_{i-k-1} also has at least 2 topological orderings, by the application of the claim proved above (the contrapositive of the backward direction). Hence $P(k+1)$ is true, which completes the induction.

Thus $P(i)$ is true, and this implies that G, which is G_0, has at least two topological orderings.

Now suppose we output "yes". This means that at all iterations, the number of sources is at most 1.

We show using induction that G has exactly 1 topological ordering.

Proof:

Induction hypothesis: $P(k) : G_{n-k}$ has exactly 1 topological ordering for $k \leq i$.

Base case: for $k = 0, 1$, this is true since a graph with 0 or 1 vertices has exactly 1 topological ordering.

Inductive step: suppose $P(k)$ is true for some $0 < k < n$, we need to show $P(k+1)$ is true.

Note that by the claim above, if G_{n-k} has exactly 1 topological ordering and G_{n-k-1} has exactly one source, then G_{n-k-1} also has exactly 1 topological ordering. However, using the inductive hypothesis, G_{n-k} has exactly 1 topological ordering and G_{n-k-1} has exactly one source since we never returned "no" (which means $sources \leq 1$ and $sources \neq 0$ since the graph is non empty as $k > 0$). Thus $P(k+1)$ is true and we are done. In particular, G_0 which is G also has a unique topological ordering, so we are done.

Therefore, our output is correct and we are done.

Running time analysis: Initialising takes $O(V)$ time, the first loop takes $O(V+E)$ time due to iteration over the adjacency list, the second loop takes $O(V)$ time, linearizing takes $O(V+E)$ time, and the final loop takes $O(V+E)$ time due to iterating over the adjacency list and doing $O(1)$ work in the inside and the outside loop. Thus the overall complexity is $O(V+E)$ which is linear.

3. (8 points) There are n cookies with $s_1 \geq s_2 \geq ... \geq s_n$ and there are n cookie monsters with cookie size requirements $r_1 \geq r_2 \geq ... \geq r_n$. The problem is to distribute the n cookies to the cookie monsters such that:

   1. every cookie monster gets exactly one cookie, and
   2. the number of satisfied cookie monsters is maximised. A cookie monster is satisfied iff it gets a cookie that has size at least its cookie size requirement.

   Design an algorithm for this problem. The output can be given as a set of tuples $\{(i_1, j_1), ..., (i_n, j_n)\}$. This means that, as per this distribution, cookie number i_t should be given to monster number j_t for all $t \in \{1, 2, ..., n\}$.

   (For example, consider an input with two cookies with size $5, 3$ and two cookie monsters with size requirements $4, 2$. For the assignment $\{(1, 2), (2, 1)\}$, the number of satisfied cookie monsters is one. For the assignment $\{(1, 1), (2, 2)\}$, the number of satisfied cookie monsters is two.)

   Here is a greedy strategy for this problem.

   Greedy strategy: Give the cookie number 1 to the first cookie monster (in order $1, ..., n$) that is satisfied by this cookie. Let j be this cookie monster. That is, $s_1 \geq r_j$ and $s_1 < r_1, s_1 < r_2, ..., s_{j-1} < r_{j-1}$. Then recursively distribute cookies $2, ..., n$ to cookie monsters $j + 1, ..., n$. At the end of this process, arbitrarily distribute the remaining cookies to the remaining monsters.

   (a) Prove correctness of the above greedy strategy.

   Solution:

   We prove this using an exchange argument.

   Define $value(OA)$ = number of satisfied monsters in a given assignment OA.

   Exchange lemma:

   If there exists a monster which can be satisfied by the first cookie, for any valid assignment of cookies OA, there is an assignment of cookies OA' such that the first cookie is given to the monster with the largest requirement which can be satisfied using it, and $value(OA') \geq value(OA)$.

   Proof:

   OA satisfies the constraints by definition. Consider the first cookie c. Suppose it has been assigned to a monster m. Suppose the monster with the largest requirement which can be satisfied using it is M, and M is assigned C. Consider the solution $OA' = OA - \{(c, m), (C, M)\} \cup \{(c, M), (C, m)\}$. Note that M is satisfied by c as per the definition of M. OA' meets constraints since exchanging still upholds the property that the cookie-to-cookie-monster mapping is a bijection. Now we make cases as follows:

   1. M is satisfied by C and m is satisfied by c. m is also satisfied by C as the requirement of m is $<$ requirement of M, so no change in number of satisfied monsters.
   2. M is not satisfied by C and m is satisfied by c. M is satisfied by c and m may or may not be satisfied by C, so the number of satisfied monsters remains the same or increases by 1.
   3. m is not satisfied by c. This case isn't possible since requirement of m can't exceed that of M by definition of M, and M is satisfied by c, so m must be satisfied by c as well.

   Hence by the analysis above, we see that OA' is at least better than OA and we have proved the exchange lemma.

   Now we show using induction on the number of cookies that the greedy algorithm gives an optimal solution.

Base case:

For the number of cookies being 0 or 1, we are done because there can be exactly 1 assignment for each of these cases.

Induction step:

Now suppose number of cookies is $c > 1$.

If there is no monster which can be satisfied by the largest cookie, we can see that no monster can be satisfied by any cookie, so an optimal solution in this case would be an arbitrary assignment, which is what the greedy algorithm does in this case.

Now suppose the set of cookie monsters which can be satisfied by the largest cookie c is non-empty, and hence the cookie monster with the largest requirement in this set exists, say M. Let the greedy solution to this problem be GA.

Consider any solution OA to this problem, and let OA' be defined as in the exchange lemma. Then $OA' = (c, M) \cup$(solution on the instance where cookie c and monster M are removed, say OS), and by the exchange lemma, we have $value(OA') \geq value(OA)$.

Let GS be the greedy solution corresponding to the instance where cookie c and monster M are removed. By the inductive hypothesis, we have $value(GS) \geq value(OS)$. Then we have, from the above, that

$$value(GA) = 1 + value(GS) \geq 1 + value(OS) = value(OA') \geq value(OA)$$

Hence, we have shown that in any case, value of the greedy solution is greater than or equal to the value of any other solution for c cookies. Specialising to the case where the "any other solution" is an optimal solution Opt, we see that the value of the greedy solution (say $Greedy$) is greater than or equal to $value(Opt)$. Since Opt is optimal, $value(Greedy)$ can not exceed $value(Opt)$, and thus must be equal. Thus $Greedy$ is optimal for c cookies, and our induction is complete.

Now, taking the number of cookies to be n and the problem instance to be the one given to us, using the result above, we conclude that the greedy solution is optimal for our instance as well.

(b) Give an efficient algorithm implementing the above strategy, and give a running time analysis for your algorithm.

Solution:

```
function SOLVE(n, s, r)
    j := 1
    stop := n + 1
    unsatisfied := empty linked list
    assignment := empty linked list
    for i ∈ {1, ..., n} do
        while j ≤ n ∧ r[j] > s[i] do
            insert j into unsatisfied
            j := j + 1
        end while
        if j ≤ n then
            insert (i, j) into assignment
            j := j + 1
        else
            stop := i
            break
        end if
```

```
    end for
    while stop ≤ n do
        pop j from unsatisfied
        insert (stop, j) into assignment
        stop := stop + 1
    end while
    return assignment
end function
```

Running time analysis:

The first four lines of the function take $O(1)$ time. Now consider the loop. For each iteration, let $j_{i_{old}}$ and $j_{i_{new}}$ be the values of j before and after this iteration. The inner loop takes time $O(j_{i_{new}} - j_{i_{old}})$ and the outer loop takes a constant amount of extra time apart from this time. Thus the total time taken is $O(i_{final} + j_{i_{final,new}} - 1))$ where $i_{final}(= stop)$ is the value of i where we break the loop (or n if the loop ends without any break from our side). Since i and j can be at most $n + 1$, this takes time $O(n)$. The final loop which does assignment takes $O(n)$ time (more precisely $O(n - j_{i_{final,new}})$ time) since it takes $O(1)$ time inside each iteration. Thus the overall complexity is $O(n)$, which is linear.