Order  Say that \( f(n) \in O(g(n)) \) are functions from non-negative integers to non-negative integers. Then which of the following follows:

1. \( f(n) + g(n) \in O(f(n)) \)
   False. \( f(n) \in O(g(n)) \), means that \( f(n) \leq cg(n) \) for large enough \( n \), but it doesn’t mean that \( f \) cannot be much smaller. So \( g(n) \) might not be \( O(f(n)) \), in which case \( f(n) + g(n) \) won’t be \( O(f(n)) \). For example, if \( f(n) = n \) and \( g(n) = n^2 \), \( f(n) + g(n) = n + n^2 \notin O(n) \).

2. \( f(n) + g(n) \in O(g(n)) \)
   True. \( f(n) \in O(g(n)) \), means that \( f(n) \leq cg(n) \) for all large enough \( n \geq n_0 \). Then \( f(n) + g(n) \leq cg(n) + g(n) = (c + 1)g(n) \) for all \( n \geq n_0 \), so using the same constant \( n_0 \) and the constant \( c' = c + 1 \) in the definition of \( O \), \( f(n) + g(n) \in O(g(n)) \).

3. \( 2^{f(n)} \in O(2^{g(n)}) \)
   False. Exponentiation can turn a constant factor difference into a polynomial difference. For example, if \( f(n) = 2n \) and \( g(n) = n \), \( f(n) \leq 2g(n) \), so \( f(n) \in O(g(n)) \). But \( 2^{2n} = 2^n * 2^n \) is not \( \leq c2^n \) for any fixed \( c \), so \( 2^{f(n)} \notin O(2^{g(n)}) \).

4. \( f(n)g(n) \in O(g^2(n)) \)
   True. \( f(n) \in O(g(n)) \), means that \( f(n) \leq cg(n) \) for all large enough \( n \geq n_0 \). Then \( f(n)g(n) \leq cg(n)g(n) = c(g(n))^2 \) for the same \( n \). So using the same constants, \( f(n)g(n) \in O((g(n))^2) \).

5. \( f(n^2) \in O(g(n^2)) \)
   True. Since \( f(n) \leq cg(n) \) for all \( n \geq n_0 \), in particular it is true when \( n = (n')^2 \). Thus, \( f((n')^2) \leq cg((n')^2) \) for all \( n' \geq \sqrt{n_0} \). So \( f(n^2) \in O(g(n^2)) \).

Next larger integer  Consider the problem of, given an array of integers \( A[1...n] \) (not sorted), finding for each \( I \), the least \( J > I \) so that \( A[J] > A[I] \) (or “NIL” if no such \( J \) exists). Find and analyze an efficient algorithm for this problem.

Say that an element \( A[J] \) with \( J > I \) is blocked at point \( I \) if there is a \( I \leq K < J \) with \( A[K] \geq A[J] \). First, if an element is blocked at point \( I \), it is blocked at all smaller \( I \). And if it is blocked at point \( I \), it will not be the next larger integer for \( I \), since if it were larger \( A[K] \geq A[J] > A[I] \), so \( K \) would be an earlier larger element. So if \( A[J] \) is blocked at point \( I \), it will not be the answer for any \( I' \leq I \). So we can delete it from consideration for smaller values of \( I \). Second, note that the elements that
are not blocked are in increasing order in the array, since if \( J < J' \) and \( A[J] \geq A[J'] \), \( J \) is blocking \( J' \).

We’ll keep an array of answers, with \( \text{Ans}[I] = J \) if the \( J \)th element is the next larger one after \( A[I] \), or \( \text{Ans}[I] = \text{nil} \) if no later element is larger. Starting with \( I = n \) and going down to \( I = 1 \), we’ll keep track of the set of unblocked elements \( A[J] \) with \( J \geq I \). We’ll use a stack of pairs \((J, A[J])\) to store them, which will stay sorted from smallest to largest in both co-ordinates. We start with an empty stack and \( I = n \). For each \( I \), we first pop elements from the stack until we reach an element \((J, A[J])\) with \( A[J] > A[I] \) or the stack is empty. If the first, we set \( \text{Ans}[I] \) to \( J \), otherwise it is set to \( \text{nil} \). Then we push \((I, A[I])\).

At all times, the stack is sorted in both co-ordinates, since when we insert an element, the \( I \) is the smallest seen so far, and we have deleted all elements less than \( A[I] \) before inserting. Also, all unblocked elements are on the stack, since we only delete an element when \( A[I] \) would block it. Say we set \( \text{Ans}[I] = J \). Then by definition, \( A[J] > A[I] \) and \( J > I \), so \( A[J] \) is a later larger element. Say \( A[K] \) were the next larger element, and \( K < J \). Then since the next largest element is unblocked, and all unblocked elements are on the stack, \((K, A[K])\) was on the stack at the start. Since the stack is ordered by both coordinates, this element was examined before we got to \((J, A[J])\). Since we continued, it must be that \( A[I] \geq A[K] \), a contradiction to \( K \) being the next larger element. Thus, in this case \( A[J] \) is the next largest element. Similarly, if we assign \( \text{Ans}[I] = \text{nil} \) and there were a next larger element \( K \), \((K, A[K])\) would have been in the stack and we would not have deleted it. Thus, our algorithm assigns the correct answer for each \( I \).

Stack operations are constant time. Since we push each \((I, A[I])\) at most once, we pop the same at most once. Thus, there are at most \( n \) pushes and \( n \) pops total throughout the algorithm, although one iteration might have almost all of the pops, and there are \( n \) iterations. In addition to the stack operations, we spend constant time per element. So the total time is \( O(n) \), since the total stack operations take \( O(n) \) time, and so does the remaining part of the algorithm.

**Graph theory** A Hamiltonian path in an undirected graph is a simple path that visits every node exactly once. Show that if every node in a graph on \( n \) nodes has degree greater than \( 2/3n \), then the graph has a Hamiltonian path.

Let \( G \) be a graph where every node has degree greater than \( 2/3n \). Assume for contradiction that \( G \) did not have a Hamiltonian path. Then it has some longest path \( p = u_1, u_2, \ldots, u_k \) with \( k < n \). If \( k \leq 2n/3 \), then \( u_k \) must have some neighbor \( v \) not on the path. But then appending \( v \) to the end would create a longer path, a contradiction.
If $k > \frac{2}{3}n$, let $v$ be any node not on $p$. Then $v$ cannot have edges to $u_1$ or $u_k$ (or we could extend the path as above.) Also, for each pair $u_{2i}$ and $u_{2i+1}$, $v$ cannot be adjacent to both elements of the pair, or else the path $p' = u_1..u_{2i}vu_{2i+1}..u_k$ would be larger than $p$. Thus, $v$ has less than $k/2$ neighbors on the path. It can have at most $n - k$ neighbors not on the path. So in total, $v$ has less than $n - k + (k/2) = n - k/2 < \frac{2n}{3}$ neighbors total, since $k > \frac{2n}{3}$. This is a contradiction to the assumption that every node has at least $\frac{2}{3}n$ neighbors.

As was pointed out by several students, Dirac’s Theorem is strictly stronger than what we just proved. However, the point is for you to work it out yourself, not look up what is known.

**Triangles** (20 points) Let $G$ be an undirected graph with nodes $v_1...v_n$. The adjacency matrix representation for $G$ is the $n \times n$ matrix $M$ given by: $M_{i,j} = 1$ if there is an edge from $v_i$ to $v_j$, and $M_{i,j} = 0$. The adjacency list representation is an array, which for each node, contains the head of a list of (pointers to) all of the node’s neighbors in the graph. A triangle in the graph is a set of three nodes that are all adjacent. Give and analyze two efficient algorithms for listing all triangles in a graph, one if the graph is given as an adjacency matrix and the other if the graph is in adjacency list format. Analyze your algorithms in terms of both the number of nodes $n$ of the graph and the number of edges $m$ of the graph. You can assume $n \leq m$.

The obvious method is to look at all triples of nodes and check whether they are all connected. This would be $O(n^3)$ to check, and if the graph were fully connected, is also a minimum time, since we would need to report $\Omega(n^3)$ triangles, all triples. So we can’t do better for dense graphs $m \in \Omega(n^2)$, and in particular can’t do better just as a function of $n$. But we can do better for sparse graphs, $m \in \omega(n^2)$. The high level strategy is, instead of checking for pairs of nodes, check for each edge and each node not on the edge, whether the two endpoints and the additional node form a triangle. In the matrix representation, this would work as in the following pseudo-code.

**Triangles** [M:[1..n][1..n]: matrix of Booleans]: list of triples of nodes (integers in the range 1...n)

1. Initialize List as an empty list of triples.
2. FOR $I = 1$ TO $I = n - 2$ do:
3. FOR $J = I + 1$ TO $J = n - 1$ do:
4. IF $M[I,J] = 1$ THEN do:
5. FOR $K = J + 1$ TO $K = n$ do:
6. IF $M[I,K]$ and $M[J,K]$ then append $(I, J, K)$ to List;
7. Return List;

While there are three nested loops, each taking \(O(n)\) time, we can do better than \(O(n^3)\) in our time analysis. The loop in lines 5 and 6 is only performed if \(M[I, J]\), i.e., at most once per edge \(\{I, J\}\). Thus, it is really an \(O(n)\) loop executed \(m\) times, so the total time spent on lines 5 and 6 is \(O(mn)\). The time spent on the rest of the algorithm is just \(O(n^2)\) from the two nested loops, with the \(O(1)\) time to check whether \(M[I][J]\) in line 4. Thus the total time is \(O(mn + n^2)\). Since you are told \(m > n\), this is \(O(mn)\).

In the adjacency list format, there are ways to solve it directly in about the same time. However, the cleanest approach is simply to first compute the adjacency matrix and then use the above algorithm. Given the array of lists \(L\), we can compute the matrix \(M\) as follows:

CreateMatrix(L: array of lists of nodes)

1. Initialize \(M[1...n][1...n]\) by setting \(M[I, J] = False\) for each \(I, J\).
2. FOR \(I = 1\) TO \(n\) do:
   3. FOR each \(J \in L[I]\) do: \(M[I, J] \leftarrow True\);
4. Return \(M\).

The above algorithm is time \(O(n^2)\), since Initialization takes time proportional to the size of the matrix, and the two nested loops each go through at most \(n\) steps.

Combining with the other algorithm gives total time \(O(n^2 + n^2 + nm) = O(nm)\) as before.