1. This video-on-demand problem can be fairly easily cast as an undirected bipartite matching problem. We create the bipartite graph $G = (U, V, E)$ as follows. We first add a vertex for each unit of capacity offered by each server. So server s_i with capacity c_i would be represented in G by setting $U = U \cup \{s_i^1, s_i^2, \ldots, s_i^{c_i}\}$. The graph need not distinguish between vertices representing different servers, though we should label vertices so that we can later map back from a vertex to a server. It will help us to think of all vertices representing a single server as belonging to a logical group. After all servers are represented in U, we represent each request r_j as a vertex in V, simply setting $V = V \cup \{r_j\}$, again making sure to label the vertices so that we can later map back to the correct request. Each request r_j is for a particular movie, so as we add each request to V we also add edges, setting $E = E \cup \{(s_i^x, r_j) \mid$ server s_i contains the movie requested by r_j and $x = 0..c_i\}$. This adds an edge from all vertices which represent a server that contains the movie requested by r_j to the vertex representing r_j. Note that since all vertices in a logical group represent the same server, that either all vertices from a logical group will have edges to a particular request vertex, or none will have edges to that request vertex (which happens in the case that server does not have the movie requested).

   Using the bipartite graph G as constructed above, we can run the Hopcroft and Karp unweighted bipartite matching algorithm, which is given in the course text. This algorithm finds a maximum matching M in time $O(m\sqrt{n})$. Once we have the maximum matching M, the solution to the video-on-demand problem is given by letting the edges in M specify assignment. That is, for each edge $(s_i^k, r_j) \in M$, assign request r_j to server s_i.

   proof of correctness: Another way to think of the maximum matching M is that each edge $(s_i^k, r_j) \in M$ corresponds to a movie streaming from server s_i to satisfy request r_j. Since each edge represents one stream, and each server is represented by a number of vertices whose cardinality equal its capacity, and in a matching each vertex can only be adjacent to at most a single edge, it is impossible for a solution given by M to specify that a server provides more streams than its capacity allows. Since the only edges in G to vertices representing requests are from vertices representing servers which contain the movie that is being requested, it is impossible for a solution given by M to specify that a server provide a movie which it does not have.

   Assume that M is a maximum matching which provides a solution to a video-on-demand problem represented by $G = (U, V, E)$, but this solution does not serve a maximum number of requests. That is to say that there is another solution to this video-on-demand problem which serves more requests. If we map this solution which serves more requests to a bipartite graph, we will get a graph $G' = (U, V, E')$. U and V are the same in both G and G' since the vertices are determined by the problem, not the solution. $E' \subseteq E$, since the only servers which can satisfy a request are those servers which contain a movie being requested and all of these edges are in E. E' must be a matching, because a single unit of capacity from any server can only satisfy a single a request and a single request can only be assigned to one server. Since the solution represented by E' serves more requests than the solution represented by M, the solution represented by E' contains more streams and thus $|E'| > |M|$, which contradicts our original assumption that M was a maximum matching. Thus, if M is a maximum matching, then the solution it specifies must serve a maximum number of requests.

   complexity analysis: For a given video-on-demand problem, let a be the number of servers, b be the number of requests, c be the maximum capacity among all servers, and m be the total number of distinct movies available from any server.

   For the first phase we must map the problem to a bipartite graph, which requires creating at most $(c * a) + b$ vertices and at most $c * a * b$ edges. Creating a vertex takes a constant amount of work, and creating an edge involves the work to check whether the server has a given movie, which in the absolute worst case (brought about by laziness) is m. Thus the whole first phase takes time $(c*a)+b+c*a*b*m$, or $O(c*a*b*m)$.

For the second phase we run the Hopcroft and Karp algorithm on the created graph G. Since G has at most $(c*a)+b$ vertices and at most $c*a*b$ edges, this algorithm will take time $O(c*a*b*\sqrt{(c*a)+b})$. The maximum matching M produced by the algorithm will have size at most b, as the matching cannot be larger than the number of vertices in either partition of G.

For the third phase we use the matching M to assign requests to servers. This phase merely assigns request r_j to server s_i for each edge $(s_i^k, r_j) \in M$. Thus this phase is linear in the number of edges in M, or $O(b)$.

The total time complexity is the sum of the time complexities for the three phases and is thus $O(c*a*b*(\sqrt{(c*a)+b}+m))$, which is ugly but is polynomial in the size of the input.

2. The KERNEL decidability problem, that is, given a directed graph $G=(V,E)$, determine if there exists $K \subseteq V$ such that K is a *kernel* of G, is in NP as a proposed solution can be checked in polynomial time. Given a proposed solution K, iterate through each $v \in V$. If $v \in K$, verify that $\forall u \mid (v,u) \in E$, $u \notin K$. If $v \notin K$, verify that $\exists u \mid (v,u) \in E$, $u \in K$. If all verifications are successful, then K is a *kernel*. An array of size n can be kept with each vertex having an entry which specifies the membership of that vertex in K. This allows for constant time membership checks of any vertex. The verifications will take a step for each outgoing edge from each vertex, thus there will be m steps in all.

   KERNEL is in NP-complete as 3CNFSAT can be reduced to it in polynomial time. Given an instance of 3CNFSAT, which is a boolean expression in 3CNF (the expression is the conjunction of clauses, each of which is the disjunction of at most 3 variables), create G as follows: First, add all variables which appear the expression to V. Then add the negated version of each variable to V. Finally, add a new clause vertex for every clause to V. Now encode the relationship between variables and their negated instances, so add an edge to E from each variable vertex to the vertex for the negated version of that same variable. Also add an edge to E from each negated variable vertex to the vertex of the normal version of that variable. Finally, encode each clause by adding an edge to E from each clause vertex to each literal that appears in the clause. Finish this by adding an edge to E from the vertex for the negation of each literal that appears in the clause to the vertex for the clause (to be clear, if the literal in the clause is negated, then the negation of the literal will be the normal, non-negated variable). In this constructed version of KERNEL, membership in K can be mapped back to truth, so if the vertex for a variable is in K then that variable should given the value *true* and vice versa. Thus if K exists, then the given 3CNF is satisfiable, and if K does not exist, the given 3CNF is not satisfiable.

   proof of correctness: To be a member of K, a vertex can not have an edge to another member of K. Thus a variable and the negated instance of that variable can never both be in K. Also, for the *kernel* to exist, every vertex not in K must have an edge to a vertex which is in K. If we ignore any other edges for the moment, this means that either the positive or negative version of a variable must be in K, otherwise the other form of that variable would not have an edge to a vertex in K.

   Now assume for a moment that the clause vertex is chosen to be in K. The negation of the literal in the clause could not be in K since it has an edge to the clause vertex. Also, the literal which appears in the clause could not be in K, since the clause vertex has an edge to it. However, this cannot be a *kernel* as the vertex for the literal which appears in the clause will not have an edge to any vertex in K. Thus without restriction, for a *kernel* to exist, either a variable or the negated instance of the same variable must be in K, and the clause vertex can never be in K.

   Since each clause vertex only has edges to the vertices representing the literals in that clause, at least one of those vertices must be in K for the clause vertex to have an edge to a vertex in K. Since these properties must hold for the entire graph, if a *kernel* exists then the vertex for at least one literal from every clause must be in K, so setting instances in K to be *true* and instances not in K to be *false* is a satisfying assignment.

   Assume that there is a satisfying assignment for the original expression, but there does not exist a *kernel* for the constructed graph. However, if we put all vertices which correspond to true literals in the satisfying assignment into K, then K will be a *kernel*, as each clause vertex will have an edge to at least one vertex in K and one of each variable and its negation will be in K while the other is not. As covered previously, these vertices and edges will satisfy the property of a *kernel*, and since the only other edges and vertices represent other clauses, the same argument applies. Thus K is a *kernel*, so we have reached a contradiction, and thus the original assumption can not be true.

complexity analysis: For the mapping from 3CNFSAT to KERNEL at most 7 vertices are created for each clause. These vertices are joined together by a total of 12 edges. This is a constant amount of work. Since each clause must be represented in the graph, the mapping takes time linear in the number of clauses.

3. The *two-dimensional discrete Fourier transform* given is:

$$F(A)_{ij} = \sum_{k=0}^{M-1} \sum_{l=0}^{N-1} A_{kl} \omega_1^{ik} \omega_2^{jl}$$

Note that the product is of scalars, so we can factor the ω_1^{ik} term out of the inner summation, since it does not depend on l, giving:

$$F(A)_{ij} = \sum_{k=0}^{M-1} \omega_1^{ik} \sum_{l=0}^{N-1} A_{kl} \omega_2^{jl}$$

Now note that if we define a matrix $F2_{xy} = \omega_2^{xy}$ for $x = 0..N$ and $y = 0..N$ then $\sum_{l=0}^{N-1} A_{kl}\omega_2^{jl}$ defines the multiplication of the N length row vector a_k by the N length column vector $f2_j$, resulting in the value $f2(a_k)_j$.

$$F(A)_{ij} = \sum_{k=0}^{M-1} \omega_1^{ik} a_k * f2_j$$

The course text defines the one dimensional *discrete Fourier transform* as $f(a) = FN * a$, where $f(a)$ and a are both column vectors of length N. Since FN is a symmetric matrix (as are $F1$ and $F2$), it would be equivalent to define this transform as $f(a) = a * FN$, where $f(a)$ and a are row vectors of length N. Thus if we allow j in the previous formula to range $0..N$, then the term $a_k * f2_j$ will produce a row vector of length N which is $F2(a_k)$, or the one dimensional *discrete Fourier transform* of a_k.

The second sum now has k range from $0..M-1$ which causes the multiplication to calculate $f2(a_k)_j$ for all k, which describes a column vector of length M that can be calculated via the MxN matrix A times the N length column vector $f2_j$. Allowing j to range $0..N$ in the previous computation gives us $A * F2$, which can be seen to be M distinct one dimensional *discrete Fourier transform*s of each row in A via the transform $F2$.

If we define a matrix $F1_{xy} = \omega_1^{xy}$ for $x = 0..M$ and $y = 0..M$ then $F(A)_{ij} = \sum_{k=0}^{M-1} \omega_1^{ik} A * f2_j$ defines the multiplication of M length row vector $f1_i$ times the result of the previous summed vector multiplication which is an M length column vector, which results in a single scalar value $f1(f2(A)_j)_i$.

$$F(A)_{ij} = f1_i * A * f2_j$$

Note that in the previous expression, if value of i is allowed to range over $0..M$, then the term $f1_i * A$ gives us $F1 * A$, which can be seen to be N distinct one dimensional *discrete Fourier transform*s of each column in A via the transform $F1$

If the value for i ranges $0..N$ and the value for j ranges $0..N$, then previous defines the matrix multiplication

$$F(A) = F1 * A * F2$$

Since matrix multiplication is associative, the two multiplications may be done in either order with the result the same. Thus the *two-dimensional discrete Fourier transform* F can be described as a sequence of two matrix multiplications. Multiplication by the matrix $F1$ describes the one dimensional *discrete Fourier transform* of each column in A by the transform $F1$. Multiplication by the matrix $F2$ describes the one dimensional *discrete Fourier transform* of each row in A by the transform $F2$. The *two-dimensional discrete Fourier transform* F is the composition of the $F1$ transforms by $F2$ transforms or vice versa, so $F = F1(F2(A)) = F2(F1(A))$. Since both $F1$ and $F2$ can be described as multiple one dimensional *discrete Fourier transforms*, we can define the whole two dimensional transform F in terms of multiple one dimensional transforms.

As such, $F(A)$ can be inverted using a similar technique to the one dimensional case—$A = F1^{-1} * F(A) * F2^{-1}$. Since $F1$ and $F2$ as we have defined them are both Vandermonde matrices, we know that their inverses exist and so we know that F itself is invertible. In addition we may use the simple formulas given in the course text to easily arrive at the inverses of $F1$ and $F2$, so F is (relatively) easy to invert.

$F(A)$ may be calculated very efficiently by applying the NC FFT algorithm given in the course text to each of the one dimensional components of the two dimensional transform. The one dimensional algorithm takes $O(\log N)$ steps and N processors.

On the first stage, we use MN processors to simultaneously calculate M different one dimensional transforms—we transform each row in A by $F2$, which takes $O(\log N)$ steps to do, and results in the MxN matrix $F2(A)$.

On the second stage, we again use MN processors to simultaneously calculate N different one dimensional transforms—we transform each column in $F2(A)$ by $F1$, which takes $O(\log M)$ steps to do, and results in the MxN matrix $F1(F2(A))$, which is the same as $F(A)$.

The entire algorithm will thus take $O(\log M + \log N)$ steps and $O(MN)$ total processors, so it is in NC.

4. (a) $C = \bigcup_t C_t$. For some number of iterations, calculate M_t, a maximal matching. For each edge in M_t, randomly choose one of its endpoints to be a member of C_t. Once C_t is chosen, delete all vertices in C_t from the graph. When a vertex is deleted, delete all edges incident to that vertex. If edges remain in the graph, increment t and start the next iteration. When this finishes, no edges remain. Each edge could only be deleted because a vertex which covered it was also deleted, which only happened when that vertex was added to a C_t. Since C is the union of all sets C_t, then it contains at least one vertex that covers each edge that was deleted. Since all edges have been deleted, C must be a vertex cover.

   (b) $M = \bigcup_t M_t$, so M contains all edges which were part of a matching. An edge can only be part of one matching, because all edges in M_t are deleted at the end of iteration t. For each edge in M_t, one of its two endpoints is chosen randomly with equal probability. Thus each endpoint of each edge in M_t is chosen with probability $\frac{1}{2}$. All chosen vertices are deleted at the end of iteration t. Once deleted, a vertex v obviously can no longer be an endpoint for a new matching, thus no new edges adjacent to v will be added to M. Also, it is possible that v is never deleted, but all edges incident to v are deleted because they are adjacent to other vertices that are deleted. Let X be a random variable whose value is the number of edges in M adjacent to v. X is then given by the number of times v is the endpoint of an edge in a matching but v is not chosen in that iteration, but before all edges adjacent to v are deleted by other vertices. We now define a random variable X' which is also the number of edges in M adjacent to v, but when the algorithm is run in a slightly restricted environment in which we can assume that there are an unlimited number (as many as we can use) of edges adjacent to v such that other vertices never cause all edges adjacent to v to be deleted, and vertices which are not adjacent to an edge in the matching are not counted. Under these conditions, X' is equal to the number of iterations for which v is the endpoint of an edge in M_t before v is chosen (and thus deleted). As is proved in the Kleinberg/Tardos boook (as well as our homework), the expected number of independent trials before an event happens is equal to the reciprocal of the of probability of that event occurring. We know that when v is the endpoint of an edge in M_t, it is chosen with probably $\frac{1}{2}$, so in this situation, $\mathcal{E}X' = 2$. Since less than 2 edges being adjacent to v in the beginning, or other vertices deleting edges adjacent to v

before that edge is chosen to be part of a matching are the only other factors that determine the value $\mathcal{E}X$, and allowing either of these to happen would decrease the value of $\mathcal{E}X'$, we know that $\mathcal{E}X \leq \mathcal{E}X'$, and thus $\mathcal{E}X \leq 2$.

(c) Any vertex cover must cover all edges in M. This means that a minimum vertex cover must include at least one endpoint of each edge in M. However, some edges in M may share endpoints. Since the expected number of edges in M adjacent to any vertex is at most 2, the expected number of edges in M which share any given endpoint is at most 2. So the expected number of edges in M which can be covered by a single vertex is at most 2. This means the expected size of M is at most 2 times the size of the minimum vertex cover. Since $|C| = |M|$, the expected size of C is at most twice the size of the minimum vertex cover.

5. Assume that G is a mixed graph which contains no directed cycles for which it is impossible to orient an undirected edge to form the directed acyclic graph, $G^{\rightarrow}$. Since there were no previous directed cycles, some directed edge had to be oriented in such a way as to form a directed cycle. However this would only occur if orienting the undirected edge in the other direction would also produce a cycle, as the only criteria for the direction an edge is oriented is avoiding cycles. We will call this undirected edge $u = (v, t)$. The oriented version in one direction is $u_l = (t, v)$ and in the other direction is $u_r = (v, t)$. Thus there must exist a cycle C_1, composed of the edges $u_r, d_0, d_1, ..., d_i$, where d_0 starts from vertex t and edge d_i goes to vertex v. There must also exist a second cycle, C_2, composed of the edges $u_l, d_{i+1}, d_{i+2}, ..., d_n$, where edge d_{i+1} starts from vertex v and edge d_n ends at vertex t. As defined however, there must also be a cycle composed of the edges $d_1, d_2, ..., d_n$. However, these are all originally directed edges, which contradicts the original assumption that G contained no directed cycles. Thus the proposition that it impossible to orient an undirected edge in G such that a directed acyclic graph, $G^{\rightarrow}$, is formed, must be false.

The previous can be applied inductively if the mixed graph contains multiple undirected edges—if the j^{th} undirected edge cannot be oriented without causing a directed cycle, then a directed cycle must already exist before the j^{th} undirected edge was oriented. Since it is possible to orient a single undirected edge such that a directed cycle is not created as long as no directed cycle exists previously, and assuming that edges are oriented correctly if possible, the previous $j - 1$ undirected edges did not create the cycle, so the graph must have originally contained a directed cycle.

(a) First, while ignoring all undirected edges in G (simply do not travel over them) use the directed DFS algorithm for topological sort from the course text to label each vertex with its post ordering number, which we call # (if vertices are then put in descending order according to their #, a topological ordering results). The DFS may form a forest, rather than a tree, but the # for each vertex will be unique, thus forming an implicit ordering between connected components. If at any point during the DFS a back edge is encountered, then we know that G contains a directed cycle, so we can stop because it is impossible to orient G in such a way that it is acyclic. If this step succeeds, then it is possible to orient G so as to produce a DAG. This entire step takes time linear in the size of G, or $O(m + n)$.

Now, we consider all edges in G. Note that at this point vertices which are adjacent to a previously directed edge are labeled with a #, and vertices adjacent only to undirected edges are unlabeled. Begin a DFS from the first vertex in the previous topological ordering (which will have the largest # of any vertex). Note that in a graph labeled as ours, a back edge will always go from a smaller # to a larger #, while all other edges will go from a larger # to a smaller #. When visiting a vertex, mark it as such. When finished with a vertex, mark it as such. When an undirected edge is encountered, use the following guidelines:

i. If the edge leads to a vertex with a smaller # than the # of the current vertex, then orient the edge towards that vertex (this should allow the DFS to follow this edge normally).

ii. If the edge leads to a vertex with a larger # than the # of the current vertex, then ignore the edge (following it would create a back edge).

iii. If the edge leads to a vertex with the same # as the # of the current vertex, then use the normal DFS logic for an undirected graph on whether or not to follow the edge—choose to follow the edge (and thus orient it towards the other vertex) only if the other vertex has finished being visited by the DFS, because if it is currently being visited, orienting the edge toward that vertex would create a back edge (so in this case ignore the edge for now).

iv. If the edge leads to an unlabeled vertex, then orient the edge towards that vertex (this should allow the DFS to follow this edge normally) and label that vertex with the same # as the current vertex.

Since no undirected edge or sequence of undirected edges is ever oriented from a smaller # to a larger #, we add no back edges (or back paths) to the graph composed of originally directed edges, thus there are no cycles created which contain an originally directed edge. For edges which are oriented toward a previously unlabeled vertex, we use the path taken by a DFS to orient the edge, so it is impossible to create back edges with respect to newly oriented edges. This algorithm uses two phases, each of which does a DFS which do constant amounts of work for each edge and vertex encountered, so the entire algorithm takes time $O(m+n)$.

(b) Assume that we have an adjacency matrix A which represents G. We will use several additional data structures for this algorithm—*directed* is an array of length n which contains an entry for each vertex, which contains a 1 if an originally directed edge is adjacent to that vertex and a 0 otherwise, *indegree* is an array of length n which contains an entry for each vertex which holds the indegree for that vertex, and B, an adjacency matrix representation of the graph G with undirected edges removed. First use n processors and constant time to initialize all entries in *directed* and *indegree* to 0. Then use n^2 processors to examine each entry in A—one processor per entry. If a processor's entry contains a 1 and the mirror entry contains a 0, then that processor sets the entries in *directed* to 1 for both vertices adjacent to the edge represented by this entry (we will refer to such vertices as directed vertices). The processor also copies this edge to the exact same entry in B. This all takes constant time.

Now calculate the transitive closure of B by constructing $(I+B)$ and then computing the product $(I+B)^n$ via repeated squaring. This uses $O((\log n)^2)$ time and $O(n^3)$ processors. Call the resulting matrix C. Next sum the columns of C—assign $O(n)$ processors to each column of C to do this in $O(\log n)$ time, storing the result in in corresponding entry of the array *indegree*.

Finally use $O(n^2)$ processors to examine each unique pair of vertices in A. Each processor has a unique assignment (i, j) such that $i < j$. i and j give the indexes of the two vertices this processor is to consider. If there is no edge at all between vertices i and j or if there is an already directed edge between vertices i and j, then the processor does nothing. Otherwise, the processor orients the edge between i and j using the following guidelines:

i. If both i and j are directed vertices, and $\mathit{indegree}[i] \neq \mathit{indegree}[j]$, then orient the edge toward the vertex of greater indegree.

ii. If both i and j are directed vertices, and $\mathit{indegree}[i] = \mathit{indegree}[j]$, then orient the edge toward the vertex of greater index, which would be j.

iii. If only one of i and j is a directed vertex, then orient the edge away from the directed vertex.

iv. If neither i or j are directed vertices, then orient the edge toward the vertex of greater index, which would be j.

The previous step takes constant time for each processor. Thus in total we have used $O(n^3)$ processors and $O((\log n)^2)$ time, thus the algorithm is in NC. The algorithm finishes with A in a state such that the graph G it represents is now a DAG.

Since there are no directed cycles in G, all directed edges in G go from vertices of lower indegree to higher indegree. The only way to form a cycle involving previously directed edges is if an undirected edge is oriented toward a lower indegree directed vertex from a higher indegree directed vertex, but this never happens, so no cycles involving previously directed edges are formed. However a cycle could be formed from oriented edges which connect directed vertices of the same degree. However since these edges are always oriented toward the vertex of greatest index (which is somewhat arbitrary but forms a total order over the vertices nonetheless), it is impossible for these edges to form a cycle among themselves. Since all oriented edges between a directed vertex and a non-directed vertex are oriented away from the directed vertex, it is impossible for oriented vertices to create path which connects a higher indegree directed vertex to a lower indegree directed vertex. Finally, all edges between two non-directed vertices are oriented towards the vertex of higher index, and since the vertex index is a total ordering, it is impossible for there to exist a cycle among these edges between non-directed vertices.