Problem 1. (a) We will construct a graph G for this problem as follows:

- For each element d_{ij} in the matrix, create a corresponding node n_{ij}.
- For each row i in the matrix, create a corresponding node p_i. Also, for each element n_{ik} in row i, create an edge (p_i, n_{ik}) with infinite capacity.
- For each column j in the matrix, we will create a corresponding node q_j. Also, for each element n_{kj} in column j, create an edge (n_{kj}, q_j) with infinite capacity.
- Create a source S with outgoing edges to all row nodes p_i. Set the capacity of (S, p_i) to r_i, the sum of the elements in row i.
- Create a sink T with incoming edges from all column nodes q_j. Set the capacity of (q_j, T) to c_j, the sum of the elements in column j.
- Lastly, for each element $d_{ij} \in Y$, remove the node n_{ij} from the graph and subtract its value from the capacities of both (S, p_i) and (q_j, T).

With this graph, we can then determine the existence of a configuration that will produce the disclosed values in $\{r_i\}$, $\{c_j\}$, and Y. The flow through each element node n_{ij} can be thought of as the value of that element in the matrix; then, because each element is connected only to its corresponding row and column nodes, the flow through a row node p_i is equal to that row's sum in the present configuration, and likewise, the flow through a column node q_j is equal to that column's sum. Since all flow goes from the source to the sink, we know that the sum of all rows will be equal to the sum of all columns. Thus, we can use a standard max-flow algorithm to determine the max flow f through the graph. If $f = \sum r_i$, then this indicates there is a configuration that produces the provided values for the row and column sums since all of the edges from source to row node are saturated, meaning the row sums in the configuration are equal to the provided values of $\{r_i\}$ (and likewise for the column nodes and $\{c_j\}$). Otherwise, if $f \neq \sum r_i$, then there is no such satisfying configuration for the elements of the matrix that produce the given values.

The construction of the graph takes $O(pq)$ time, since there are $O(pq+p+q)$ nodes and edges to create.

(b) Given a max-flow f through G, there are two ways to assign a different flow value to an edge $e = (v, w)$. The first is to redirect flow away from (v, w); that is, find an alternative path from v to w with remaining capacity. We can do this by looking at the residual graph of G and finding a route from w to v in the residual graph (since it only contains edges with remaining capacity in reverse). Similarly, another way to alter the flow value of e is to increase the net flow through (v, w), i.e. by redirecting flow away from (w, v) via searching for a path in the residual graph from v to w. In both methods, we have to search for a route between v and w (or vice versa) in the residual graph using BFS. If the BFS does not return a valid path for either direction, it follows that there is no way to redirect flow through e, and the value for f_e is unique. This algorithm takes $O(2(m + n)) = O(m) = O(pq)$ time, since we perform BFS on the residual graph twice.

(c) We can combine parts (a) and (b) to solve the stated problem as follows. Using the algorithm in part (a), we determine if there is *some* configuration of cell values that matches the values given in $\{r_i\}$, $\{c_j\}$, and Y. If there is not, then we are done (since no such matrix exists); and if there is, then we proceed to use the algorithm from part (b) on each element node, or more precisely, on each edge (p_i, n_{ij}). If there is any element node n_{ij} for which the incoming

flow cannot be altered whilst preserving the max-flow, then this means d_{ij}'s value in the matrix is fixed for the given parameters, and therefore that element is unprotected. Thus, the combined algorithm can traverse all the element nodes to determine which elements are unprotected. Since the algorithm from part (b) is run for each of the element nodes in the graph, the overall runtime is $O(p^2q^2)$.

Problem 2. (a) Suppose that there are n left nodes and n right nodes. We can connect a source to all nodes on the left with capacity 1 and similarly connect a sink to all nodes on the right, also with capacity 1. Additionally, we add an edge from each node on the left to each node on the right with capacity 1, for a total of n^2 edges. Then, we can use max-flow on this graph to determine a maximum bipartite matching. Observe that the min-cut of the augmented graph (with the source and sink) is n, which can be obtained by cutting off either the source or sink. From the integrality property, since all the capacities are integers, it follows that there is a max-flow n with all integral flow values. Since each node on the right can have a maximum of 1 outgoing flow, we know that it also has exactly 1 incoming flow from a node on the left, so that every node on the left is "matched" with exactly one node on the right, and there are n such matchings coming from n max-flow. Thus, if we take the edges between the left and right subsets with flow value of 1 as our "pairs", then we have found a maximum bipartite matching for all nodes in the graph.

(b) As in lecture, suppose we have already run d blocking flows on the bipartite graph. For the bipartite unit graph, each left vertex as indegree 1, and so the residual graph is 0 or 1 on every edge. This means that only n/d more blocking flows are required. If we set $d = \sqrt{n}$, then we obtain a total of $O(\sqrt{n})$ blocking flows required to reach max-flow. Since each blocking flow takes $O(m)$ time, it follows that maximum matching for the bipartite graph takes $O(m\sqrt{n})$ time.

(c) After running d blocking flows, there will be d layers in the layer graph from source to sink. Since there are a total of n nodes, there must be some layer for which the number of nodes in that layer does not surpass n/d by law of averages. Similarly, if we consider any two pairs of adjacent layers, there must be some layer for which the number of total nodes in both layers does not surpass $2n/d$. In the best case, each layer possesses $(2n/d)/2 = n/d$ nodes. If we consider the two layers as a cut, then there can be a maximum of $(n/d)^2 = n^2/d^2$ residual flow passing through the cut.

Then, suppose we have already run d blocking flows. Based on the above, we require $O(n^2/d^2)$ blocking flows for the residual flow, so the total number of blocking flows required is $O(d + n^2/d^2)$. We can choose the value for d that minimizes the expression by setting $d = n^2/d^2$, which gives $d = n^{2/3}$. Thus, the number of blocking flows required to find the max-flow is $O(n^{2/3})$.

(d) We know that the distance between source and sink in this graph is bounded by $n/\sqrt{f}$ using the same reasoning as in part (c). Since each of the shortest paths must be bounded by this length, and each path carries with it one unit of flow, it follows that the total number of edges used by f is $O(n/\sqrt{f} * f) = O(n\sqrt{f})$.

Problem 3. (a) We can set up a bipartite graph with students in U and faculty members in V. We augment the graph with a source S and add an outgoing edge from the source to every student $u_i \in U$ with capacity 1. Similarly, we also augment the graph with a sink T and add an incoming edge from every faculty member $v_j \in V$ to the sink with capacity 1 as well. Finally, we add an edge for every student-faculty pair (u_i, v_j) if student u_i has requested to meet with faculty member v_j with capacity 1. We observe that the min-cut of this graph is $|U| = |V| = n$ (by cutting either the source or sink off) which implies that the max-flow is also n, which is possible if each student splits their incoming flow value of 1 into d outgoing flow values of $1/d$, and each faculty member receives d incoming flow values of $1/d$ and combines them into 1 outgoing flow. By the integrality property, it follows that there also exists a max-flow with all integral flow values, which we can envision as a student sending all their flow through one outgoing edge to a single faculty member. Then, since there are n students and n faculty, and the max-flow shows that n meetings can happen simultaneously, it follows that it is possible to schedule a single slot wherein every student meets with a different faculty member.

(b) By symmetry, there is an equal number of mappings of 1-to-1 student-faculty pairs for each of the outgoing edges of any given student. In general, there should be d non-overlapping configurations of 1-to-1 mappings in which each student meets with a faculty member, where for each student $u_i \in U$, none of the configurations share a common edge of u_i. Thus, these d configurations correspond to d time-slots, and since none of the configurations share a common edge of u_i, it follows that after the d-th meeting, each student will have met with each of their requested faculty members.

(c) Without loss of generality, suppose there are s students and $t < s$ faculty members, and construct a bipartite graph similar to the one from part (a) where $|U| = s$ and $|V| = t$. In the worst case, each student will want to meet with each faculty member, such that each student $u_i \in U$ has t outgoing edges. We can augment the graph with $s - t$ "ghost" faculty members, so that there are now s total faculty members, and draw an edge between each student and each ghost faculty. Thus, we have created a complete bipartite graph, where each student would like to meet with s faculty members, and each faculty member has s students to meet with. We can then use the result from part (b) to schedule all the meetings with s timeslots. Lastly, for each meeting with a "ghost" faculty (as well as any student-faculty pairs which were not originally requested), we can simply throw away that meeting. Therefore, we have successfully arranged all the meetings.

(d) As per lecture, bipartite matching can be done in $O(m\sqrt{n})$ using blocking flows on the unit-capacity case, so we can match students to faculty members in $O(sn^{3/2})$ time since there are $O(sn)$ edges in the graph. However, this only accounts for one timeslot, and from part (c), we know that we can finish all the meetings in s timeslots. Thus, we can begin with the original graph, find a bipartite matching, schedule the meetings, and then remove the corresponding edges and repeat on the remaining graph until all edges have been removed, at which point all requested meetings have occurred. In total, there will be s such iterations, and each iteration requires $O(sn^{3/2})$ time, leading to an overall runtime of $O(s^2n^{3/2})$.

Problem 4. (a) We can model this problem as min-flow problem by creating a graph G as follows:

- Initialize a source S.
- For each prefrosh, create a node p. Add an edge from the source to each prefrosh with capacity 1 and cost 0.
- For each student, create a node t. Add an edge from each prefrosh to each student with capacity 1 and cost $-f(p,t)$, as long as $f(p,t) \neq -\infty$.
- For each suite, create a node s. Add an incoming edge for each student that lives in that suite with capacity 1 and cost 0.
- For each floor, create a node g. Add an incoming edge for each suite s on that floor with capacity m_s and cost 0.
- For each dormitory, create a node d. Add an incoming edge for each floor g of that dormitory with capacity M_g and cost 0.
- Finally, create a sink T and add an incoming edge for each dormitory d with capacity $\mathcal{M}_d$.

This graph accurately incorporates the problem constraints into a network flow graph. We can use min-cost max-flow to determine the maximum flow through this graph that incurs the minimum cost. If the max-flow is not equal to the total outgoing capacity of S, then not all prefrosh are able to be assigned. Otherwise, there is some assignment that matches all prefrosh to students according to the constraints given, and since the only edges that have costs are the edges between prefrosh and students, the algorithm will try to maximize the suitability across all assignments (i.e. minimize the sum of negative suitabilities) when determining the assignments, thus solving the problem using min-cost flow.

(b) We can address the limits on suites with a slight modification to the approach from part (a), by introducing "overflow" nodes for each suite. That is, for each suite, we add an additional node s', and create incoming edges for each student that lives in the suite with capacity 1 and cost 0. Additionally, we create an edge from the new node s' to the corresponding floor g with capacity ∞ and cost $\sum_{p \in \text{Prefrosh},\, t \in \text{Students}} f(p,t)$.

Thus, the algorithm will function as before, except that now it is possible to bypass the suite limits using the overflow nodes. Like previously, if the max-flow determined by the algorithm is not equal to the total outoging capacity of S, then there is no such solution to the given problem. Otherwise, the algorithm will attempt to assign prefrosh to students abiding by the constraints, including minimizing overage, and bypassing the suite limits if necessary. However, due to the exorbitant cost of overflowing (which is greater than the sum of all preference combinations combined), the algorithm will still prioritize solutions that do not overflow, while still maximizing the suitabilities as in part (a). Thus, the algorithm minimizes the total overage and maximizes the total suitability efficiently.

Problem 5. The shortest augmenting path algorithm computes a min-cost max-flow in the unit-capacities graph. We can show this by first proving that the SAP algorithm never introduces a negative reduced-cost cycle in the residual graph by using induction as follows. For the base case (i.e. the beginning), there are no negative cost cycles yet because we have not augmented any paths. Then, assume that after the k-th augmentation, there are still no negative cost cycles. Using shortest path distances (i.e. Dijkstra's) from the source, we can compute feasible prices for each of the remaining nodes. From shortest paths, we know that for any edge (v, w), $d(w) \leq d(v) + c(v, w)$, which implies that $0 \leq d(v) - d(w) + c(v, w)$; in other words, the reduced edge costs are non-negative, so when we perform the $(k+1)$-th augmentation, we know that no new negative cost cycles will be introduced, closing the induction. Going back to the min-cost max-flow question, since SAP does not introduce any negative cost cycles, the algorithm does indeed compute the min-cut max-flow for unit-capacity graphs. A similar reasoning can be applied to show that Dinic's blocking-flow algorithm also does not introduce any negative cost cycles, and therefore computes the min-cut max-flow as well.