Rubric: The following standard rubric applies to all maximum flow reductions:

- 4 points for correct reduction
  - 1 for vertices
  - 1 for edges
  - 1 for capacities, demands, and/or costs
  - 1 for remaining details
- 4 points for proof of correctness
  - 2 points for optimal flow $\Longrightarrow$ optimal solution
  - 2 points for optimal solution $\Longrightarrow$ optimal flow
- 2 points for time analysis
  - The time bound *must* be stated in terms of input parameters, not just parameters of the flow network.
  - No penalty for slower polynomial-time algorithms.
  - 1 point partial credit for 'polynomial time' but no explicit bound

Similar rubrics apply to other reductions (for example, to minimum-cost circulations, maximum matchings, k-ary assignment, or other homework problems).

1. Suppose we are given a set of boxes, each specified by their height, width, and depth in centimeters. All three side lengths of every box lie strictly between 10cm and 20cm. As you should expect, one box can be placed inside another if the smaller box can be rotated so that its height, width, and depth are respectively smaller than the height, width, and depth of the larger box. Boxes can be nested recursively. Call a box is *visible* if it is not inside another box. Describe and analyze an algorithm to nest the boxes so that the number of visible boxes is as small as possible.

Solution (reduction to maximum matching): We define a bipartite graph G as follows:

- G has $2n$ vertices $u_1, u_2, \ldots, u_n, v_1, v_2, \ldots, v_n$. Vertices u_i and v_i correspond to box i.
- G contains an edge $u_i v_j$ if and only if box i can be placed inside box j.

Now consider any valid nesting of the boxes. Say that box i is *directly* inside box j if box i is inside box j but no boxes are nested between boxes i and j. Each box is either visible or directly inside exactly one other box. The bounds on the side lengths imply that at most one box can be directly inside any other box. Thus, any legal nesting of the boxes corresponds to a *matching* in G, where an edge $u_i v_j$ appears in the matching if and only if box i is directly inside box j.

Conversely, let M be any matching in G. This matching assigns each box to lie directly inside at most one other box. For any edge $u_i v_j$ in G, the smallest dimension of box i is strictly smaller than the smallest dimension of box j. Thus, the assignment induced by M cannot include a *cycle* of nested boxes, so M corresponds to a valid nesting.

For any valid nesting of the boxes, a box i is visible if and only if no edge in the corresponding matching touches u_i. So the number of visible boxes is exactly n minus the number of edges in the matching. Thus, the nesting with the fewest visible boxes corresponds to the maximum-cardinality matching in G. We can construct G in $O(n^2)$ time and then compute the maximum matching in $\boldsymbol{O(VE) = O(n^3)}$ ***time*** as described in the lecture notes. ■

Solution (reduction to flow with vertex demands): Consider the following problem: Can the boxes be nested so that at most k nodes are visible? If we can solve this decision problem quickly, we can find the minimum possible number of visible boxes by performing binary search over all possible values of k.

To solve this decision problem, we compute a maximum flow in a directed graph G with edge capacities and vertex demands, as follows.

- G has a vertex v_i for each box i, plus three additional vertices s, s', and t. Each vertex v_i has a *demand* of 1.
- G has an edge $s \to s'$ with capacity k; this is the only edge with finite capacity.
- G has an edge $v_i \to v_j$ if and only if box i fits inside box j.
- For each box i, the graph G has edges $s' \to v_i$ and $v_i \to t$.

Any nested sequence of boxes corresponds to a directed path from a to t in this graph, and vice versa. Thus, we can nest the boxes into k groups if and only if there is a flow in G with value k that meets all the vertex demands. Conversely, if the maximum feasible flow in G has value k, it can be decomposed into k directed paths from s to t, which describe a nesting of the boxes into k groups.

The network G has $O(n)$ vertices and $O(n^2)$ edges and can be constructed in $O(n^2)$ time. Maximum flows in networks with vertex demands can be computed in $O(VE \log V) = O(n^3 \log n)$ time using the most efficient max-flow algorithms. Thus, the overall running time of this algorithm is $\boldsymbol{O(n^3 \log^2 n)}$. ■

Rubric: 10 points max: standard rubric for max-flow reductions. These are not the only correct solutions.

2. Suppose we are given an array $A[1..m][1..n]$ of non-negative real numbers. We want to *round* A to an integer matrix, by replacing each entry x in A with either $\lfloor x \rfloor$ or $\lceil x \rceil$, without changing the sum of entries in any row or column of A. Describe an efficient algorithm that either rounds A in this fashion, or reports correctly that no such rounding is possible.

Solution (reduction to greedy): First, we check whether every row and every column of A sums to an integer; if not, no legal rounding is possible.

Next, we split A into two $m \times n$ arrays I (integer) and F (fractional) by setting $I[i,j] := \lfloor A[i,j] \rfloor$ and $F[i,j] := A[i,j] - I[i,j]$ for all indices i and j. If F' is a legal rounding for F, then $I + F'$ is a legal rounding for A. So we just have to compute a legal rounding of the fractional matrix F; any legal rounding of F consists entirely of 0s and 1s.

But rounding F is precisely the problem considered in the first question in Homework 3. Although that homework asked only about square matrices (with $m = n$), the algorithm also works for rectangular matrices with only trivial modifications to either the code or the proof of correctness. Assuming without loss of generality that $m < n$, the algorithm described in the solutions runs in $\boldsymbol{O(m^2 n)}$ ***time***.[1] ■

Solution (reduction to flows): First, we check whether every row and every column of A sums to an integer; if not, no legal rounding is possible.

Next, we split A into two $m \times n$ arrays I (integer) and F (fractional) by setting $I[i,j] := \lfloor A[i,j] \rfloor$ and $F[i,j] := A[i,j] - I[i,j]$ for all indices i and j. If F' is a legal rounding for F, then $I + F'$ is a legal rounding for A. So we just have to compute a legal rounding of the fractional matrix F; any legal rounding of F consists entirely of 0s and 1s.

To find a legal rounding of F, we create a graph G with a vertex r_i for each row i, a vertex c_j for each column j, and two additional vertices s and t. The graph has the following edges:

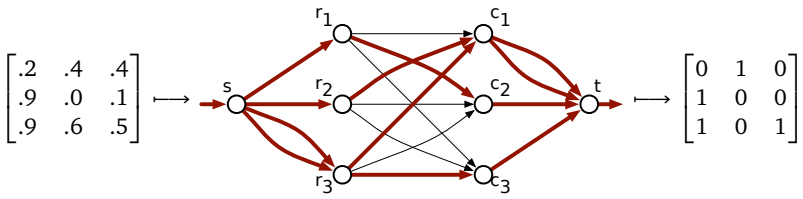
- An edge $s \to r_i$ with capacity $\sum_k F[i,k]$ for each row i
- An edge $c_j \to t$ with capacity $\sum_k F[k,j]$ for each column j
- An edge $r_i \to c_j$ with capacity 1 for each row i and column j

Now we compute a maximum flow in G from s to t. Every edge capacity is an integer, so Ford-Fulkerson computes an *integer* maximum flow. Finally, for each i and j, let $F'[i,j]$ denote the flow through the edge $r_i \to c_j$.

The maximum flow value cannot exceed $\sum_{i,j} F[i,j]$, because this is the total capacity of the edges leaving s. We claim that F' is a legal rounding of F if and only if the value of the maximum flow in G is $\sum_{i,j} F[i,j]$. We prove this claim as follows:

- Suppose the max flow value is $\sum_{i,j} F[i,j]$. Then every edge out of s must be saturated, so any row of F' has the same sum as the corresponding row of F. Similarly, every edge into t must be saturated, so the column sums are maintained. Finally, because we find an integer max flow, every $F'[i,j]$ is either 0 or 1.
- On the other hand, if F' is any legal rounding of F, then we can construct a valid flow f in G with value $\sum_{i,j} F[i,j]$, by setting $f(r_i \to c_j) = F'[i,j]$. Thus, if the maximum flow value is less than $\sum_{i,j} F[i,j]$, no legal rounding is possible.

[1] With more care, the running time can be reduced to $O(mn)$.



Rounding a fractional matrix via maximum flows; every edge in the graph has capacity 1.

The network G has $O(mn)$ edges, and the maximum flow value is at most $\sum_{i,j} F[i,j] < mn$. Thus, Ford-Fulkerson computes the maximum flow in $O(E|f^*|) = \boldsymbol{O(m^2n^2)}$ ***time***. ■

Rubric: 10 points max: standard rubric for max-flow reductions. These are not the only correct solutions.

3. Describe and analyze an algorithm to compute a donation schedule, describing how much money each voter should send to each candidate on each day, that guarantees that every candidate gets enough money to win their election. The schedule must obey both Federal laws and individual voters' budget constraints. If no such schedule exists, your algorithm should report that fact.

Solution: We solve the problem by computing a maximum flow in a directed graph G with the following vertices and edges:

- G has a source vertex s, a vertex d_i for each day between today and election day, a vertex p_j for each party member, a vertex c_k for each candidate, and a target vertex t.
- For each day i, there is an edge $s \to d_i$ with infinite capacity.
- For each day i and party member j, there is an edge $d_i \to p_j$ with capacity 100.
- For each party member j and candidate k, there is an edge $p_j \to c_k$, whose capacity is the total amount party member j is willing to donate to candidate k. (To simplify the solution, I will assume this number is an integer.)
- Finally, for each candidate k, there is an edge $c_k \to t$ whose capacity is the donation target for candidate k. (To simplify the solution, I will assume this number is an integer.)

If there are D days until the election, P party members, and C candidates, this graph has $O(D+P+C)$ vertices and $O(DP+PC)$ edges.

Say that a donation schedule is *successful* if all laws and budget constraints are satisfied and every candidate receives *exactly* their donation target. A successful donation schedule can be described by a three-dimensional array D; each entry $D[i,j,k] \geq 0$ describes how many dollars party member j should donate to candidate k on day i. We can transform any such schedule into a feasible flow f in G as follows:

- For each day i, we have $f(s \to d_i) = \sum_{j,k} D[i,j,k]$.
- For each day i and party member j, we have $f(d_i \to p_j) = \sum_k D[i,j,k]$, which is at most 100 by law.
- For each party member j and candidate k, we have $f(p_j \to c_k) = \sum_i D[i,j,k]$, which is at most the amount party member j is willing to give candidate k, because budget constraints are satisfied.
- Finally, for each candidate k, we have $f(c_k \to t) = \sum_{i,j} D[i,j,k]$, which is equal to candidate k's donation target, because the donation schedule is successful.

We easily confirm that this flow f is non-negative everywhere, satisfies all capacity and conservation constraints, and saturates every edge into t. In particular, f is a maximum flow in G.

On the other hand, let f be a feasible flow in G that saturates every edge into t. Then f must be a maximum flow, so without loss of generality, f is integral. It follows that we can decompose f into the weighted sum of directed paths in G. Every directed path in G has the form $s \to d_i \to p_j \to c_k \to t$. Let $D[i,j,k]$ denote the weight of $s \to d_i \to p_j \to c_k \to t$ in the path decomposition of f. Straightforward definition-chasing implies that D is a successful donation schedule.

Thus, there is a successful donation schedule if and only if the maximum flow in G saturate every edge into t. We can compute the maximum flow in G in $\boldsymbol{O(VE \log V) = O((D+P+C) \times (DP+PC) \log(D+P+C))}$ ***time*** using the most efficient combinatorial maxflow algorithms described in the lecture notes. ■

Rubric: 10 points max: standard rubric for max-flow reductions. These are not the only correct solutions.

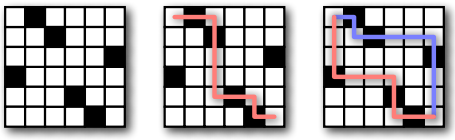
4. Consider an $n \times n$ grid, some of whose cells are marked. A *monotone* path through the grid starts at the top-left cell, moves only right or down at each step, and ends at the bottom-right cell. We want to compute the minimum number of monotone paths that cover all the marked cells.

   (a) One of your friends suggests the following greedy strategy:

   - Find (somehow) one "good" path π that covers the maximum number of marked cells.
   - Unmark the cells covered by π.
   - If any cells are still marked, recursively cover them.

   Prove that this greedy strategy does *not* always compute an optimal solution.

   Solution: Here is a 6×6 counterexample:



   The greedy strategy is not optimal.

   The middle figure shows a greedy path; in fact any greedy monotone path covers the same four marked cells. If we start with a greedy path, we need two more monotone paths to cover the other two marked cells. The right figure shows that the marked cells can all be covered with just two monotone paths. ■

   Rubric: 3 points max = 1 for input + 1 for greedy output + 1 for correct output

   (b) Describe and analyze an efficient algorithm to compute the smallest set of monotone paths that covers every marked cell. The input to your algorithm is an array $M[1..n, 1..n]$ of booleans, where $M[i, j] =$ True if and only if cell (i, j) is marked.

   Solution (reduction to flow with demands): Consider the following decision problem: Can the marked cells be covered using at most k monotone paths? If we can solve the decision problem efficiently, then we can find the minimum number of paths efficiently by performing binary search over the possible values of k.

   We construct a directed flow network G, with both capacities and demands on the edges, as follows.

   - G has two vertices $u_{i,j}$ and $v_{i,j}$ for each pair of array indices i and j, plus an additional source vertex s. The target vertex is $v_{n,n}$.
   - Each vertex $u_{i,j}$ has an outgoing edge to its partner $v_{i,j}$. Each partner edge has *demand* 1 if $M[i, j] =$ TRUE and demand 0 otherwise; partner edges do not have capacities.
   - Each vertex $v_{i,j}$ has outgoing edges to its right neighbor $u_{i+1,j}$ and its downward neighbor $u_{i,j+1}$ (if they exist). Neighbor edges have neither demands nor capacities.
   - Finally, there is an edge $s \to u_{1,1}$ with capacity k.

   Each monotone path through the grid is represented by a directed path in G. Conversely, every directed path in G represents a monotone path through the grid. Thus, if we can cover the marked cells with k monotone paths, then there is a feasible flow through the network from s to $v_{n,n}$. Conversely, if the network admits an (s, t)-flow with value k, that flow decomposes into k monotone paths from $(1, 1)$ to (n, n), which must cover all the marked cells. So we can solve the decision problem by computing a maximum flow in G.

The network G has $O(n^2)$ vertices and edges, and we can easily construct it in $O(n^2)$ time. Computing a maximum flow in a network with both capacities and demands requires $O(VE \log V) = O(n^4 \log n)$ time using the most efficient max-flow algorithms. Thus, the overall running time of this algorithm is $\boldsymbol{O(n^4 \log^2 n)}$. ■

Solution (reduction to minimum-cost flows): We construct a directed flow network G where edges have demands and costs, but no capacities.

- G has two vertices $u_{i,j}$ and $v_{i,j}$ for each pair of array indices i and j, plus an additional source vertex s. The target vertex is $v_{n,n}$.
- Each vertex $u_{i,j}$ has an edge to its partner $v_{i,j}$. Each partner edge has *demand* 1 if $M[i,j] = \text{TRUE}$ and demand 0 otherwise. Partner edges have cost 0.
- Each vertex $v_{i,j}$ has edges to its right neighbor $u_{i+1,j}$ and its downward neighbor $u_{i,j+1}$ (if they exist). Neighbor edges have demand 0 and cost 0.
- Finally, there is an edge $s \to u_{1,1}$ with *cost* 1 and demand 0.

Each monotone path through the grid is represented by a directed path in G. Conversely, every directed path in G represents a monotone path through the grid. Thus, if we can cover the marked cells with k monotone paths, then there is a feasible flow through the network from s to $v_{n,n}$. Conversely, if the network admits an (s,t)-flow with value k, that flow decomposes into k monotone paths from $(1,1)$ to (n,n), which must cover all the marked cells. So we can find the minimum number of monotone paths that cover all marked cells by computing a minimum-cost feasible flow in G.

The network G has $O(n^2)$ vertices and edges, and we can easily construct it in $O(n^2)$ time. Computing minimum-cost flows takes polynomial time, so the running time of our algorithm is polynomial. Specifically, Orlin's algorithm computes the minimum-cost flow in $O(E^2 \log V + EV \log^2 V) = \boldsymbol{O(n^4 \log^2 n)}$ ***time***. ■

Solution (reduction to maximum matching): We construct a bipartite graph G with the following vertices and edges.

- G has vertices $u_1, u_2, \ldots, u_m, v_1, v_2, \ldots, v_m$, two for each marked cell. We can compute the set of m marked cells in $O(n^2)$ time by brute force.
- G has an edge $u_i v_j$ if and only if there is a monotone path from marked cell i to marked cell j. We can compute this set of $O(m^2)$ edges in $O(m^2)$ time by brute force.

Now consider any collection of monotone paths $\pi_1, \pi_2, \ldots$ that cover every marked cell in the grid. Assign each marked cell to the first path π_k that covers it; let M_k denote the sequence of marked cells assigned to π_k. Let M be the set of edges $u_i v_j$ such that marked cells i and j are assigned to the same path π_k, and no other marked cell between i and j is assigned to π_k. Each vertex u_i or v_j is incident to at most one edge, so M is a matching in G.

Conversely, let M be a matching in G. For each edge $u_i v_j$, draw a monotone path from marked cell i to marked cell j. Each marked cell is then an endpoint of at most two monotone paths. If we merge any pair of monotone paths that share an endpoint, the result is a set of monotone paths that cover every marked cell; we can easily extend these monotone paths to start at $(1,1)$ and end at (n,n).

Thus, there is a one-to-one correspondence between matchings in G and sets of monotone paths in the grid. Moreover, any set of ℓ monotone paths corresponds to a matching with

$m - \ell$ edges. Thus, finding the *minimum* number of monotone paths that cover every marked cell is the same as finding the *maximum*-cardinality matching in G. We can compute this matching in $O(VE) = O(m^3)$ time using the Ford-Fulkerson augmenting path algorithm, as described in the lecture notes.

Altogether, our algorithm runs in $\boldsymbol{O(n^2 + m^3) = O(n^6)}$ ***time***. ■

Solution (reduction to problem 1): We construct a collection of m boxes, one for each marked cell in the input grid. Specifically, for each marked cell (i, j), we construct a box $B(i, j)$ with the following dimensions (in centimeters):

$$\left(10 + \frac{i}{n} + \frac{j}{n^2}\right) \times \left(14 + \frac{i}{n^2} + \frac{j}{n}\right) \times \left(18 + \frac{i}{n^2} + \frac{j}{n^2}\right)$$

All boxes have height between 10cm and 11cm, width between 14cm and 15cm, and depth between 18cm and 19cm, so boxes can only fit in their given orientations.

There is a monotone path from cell (i, j) to cell (i', j') if and only if $i \le i'$ and $j \le j'$ but $(i, j) \ne (i', j')$. Box $B(i, j)$ fits inside box $B(i', j')$ if and only if the following inequalities must be satisfied:

$$in + j < i'n + j', \qquad i + jn < i' + j'n, \qquad i + j < i' + j'.$$

Because all indices are integers between 1 and n, the first inequality holds if and only if $i \le i'$ and $(i, j) \ne (i', j')$, the second inequality holds if and only if $j \le j'$ and $(i, j) \ne (i', j')$, and the third inequality is implied by the first two. Thus, $B(i, j)$ fits inside $B(i', j')$ if and only if there is a monotone path from cell (i, j) to cell (i', j').

It follows that there is a one-to-one correspondence between valid nestings of the boxes $B(i, j)$ and sets of monotone paths in the grid that cover every marked cell. In particular, the marked cells can be covered with k monotone paths if and only if the boxes can be nested so that exactly k are still visible. The algorithm from problem 1 finds the nesting with the fewest visible boxes in $O(m^3)$ time. Thus, we can find the smallest set of monotone paths that cover every marked cell in $\boldsymbol{O(m^3) = O(n^6)}$ ***time***. ■

Rubric: 7 points max: standard rubric for max-flow reductions (3 for correct reduction + 3 for correctness + 1 for time analysis). These are not the only correct solutions. Only one solution is required for full credit!

5. Let G be a directed graph with two distinguished vertices s and t, and let r be a positive integer. Two players named Paul and Sally play the following game. Paul chooses a path P from s to t, and Sally chooses a subset S of at most r edges in G. The players reveal their chosen subgraphs simultaneously. If $P \cap S = \emptyset$, Paul wins; if $P \cap S \neq \emptyset$, then Sally wins. Both players want to maximize their chances of winning the game.

   (a) Prove that if Paul uses a deterministic strategy, and Sally knows his strategy, then Sally can guarantee that she wins.

   Solution: If Sally knows Paul's deterministic strategy, she can predict exactly which path he will choose in any given graph. Sally can win by choosing any single edge on Paul's path. ■

   Rubric: 1 point; all or nothing.

   (b) Let M be the number of edges in a minimum (s,t)-cut. Describe a deterministic strategy for Sally that guarantees that she wins when $r \geq M$, no matter what strategy Paul uses.

   Solution: Sally wins by choosing all M edges in the minimum (s,t)-cut, because by definition, every path from s to t contains at least one edge in this cut. ■

   Rubric: 1 point; all or nothing.

   (c) Prove that if Sally uses a deterministic strategy, and Paul knows her strategy then Paul can guarantee that he wins when $r < M$.

   Solution: First, Paul computes a set of M edge-disjoint paths $\pi_1, \pi_2, \ldots, \pi_M$ from s to t. (The maxflow mincut theorem implies that such a set exists.) If Paul knows Sally's strategy, he can predict precisely which r edges she will choose from any given graph. Each of these edges lies on at most one path π_i; thus, at least one path π_i contains none of Sally's edges. Paul can win by choosing that path. ■

   Rubric: 2 points max = 1 for strategy + 1 for proof

   (d) Describe a randomized strategy for Sally that guarantees that she wins with probability at least $\min\{r/M, 1\}$, no matter what strategy Paul uses.

   Solution: Sally computes the minimum (s,t)-cut, and then chooses r of its M edges uniformly at random. Any path from s to t contains at least one edge in the minimum (s,t)-cut. Thus, no matter what path Paul chooses, Sally chooses an edge on that path with probability at least r/M, as required. ■

   Rubric: 3 points max = 2 for strategy + 1 for proof

   (e) Describe a randomized strategy for Paul that guarantees that he loses with probability at most $\min\{r/M, 1\}$, no matter what strategy Sally uses.

   Solution: Again, Paul computes a set of M edge-disjoint paths $\pi_1, \pi_2, \ldots, \pi_M$ from s to t. (The maxflow mincut theorem implies that such a set exists.) Finally, Paul's strategy is to choose one of the paths π_i uniformly at random. No matter what r edges Sally chooses, each of those edges blocks at most one path π_i. Thus, at most r paths π_i are blocked by Sally's edges. Thus, Paul's random path hits one of Sally's edges with probability at most r/M, as required. ■

   Rubric: 3 points max = 2 for strategy + 1 for proof