Question 1

(a) I picked questions 2 and 4 because I struggled with min-cut/max-flow and residual graphs as a whole. I got through both of them but had to use online resources to ensure I was correct.

(b) I picked questions 7 and 10 because I wanted to work on problems that were not similar to those from the first 5. I got through them but they were hard and I was time crunched.

(c) I definitely need to continue practicing the flow and residual graph problems more. I should also practice some more NP reduction proofs so I am more comfortable with them.

(d) I am pretty comfortable with generic algorithm design (like problem 9) and dynamic programming (like 7). I will be prioritizing the flow and NP type problems because they are most difficult to wrap my head around.

Question 2.1

(a)

$$dist(v,i,j) = \begin{cases} 0 & \text{if i=0 and v is source for iteration j} \\ -\infty \\ \text{if i=0 and v isn't source for iteration j} \\ max(max(dist(u,i-1,j)) + weight(u,v), dist(v,i-1,j)) \\ \text{otherwise for each edge(u, v) for iteration j} \end{cases} \tag{1}$$

(b) This recurrence represents an algorithm where we update the distance to a node v from a connected node u if this distance is greater than the current distance stored to v. The i represents the total number of iterations we have to run the recurrence with i=0 being the first where j represents running the algorithm once with every vertex being the source. During iteration j, if i=0 and v is the source vertex, we set that distance to 0, and set all other non-source vertices to $-\infty$. Otherwise, we check the largest distance to each node that has an edge connecting to u and take the largest value between that and the value calculated for dist(v) during the previous iteration. We repeat for the entirety of j=0, and then switch to another vertex being the source for the next iteration when j=1 where we repeat the process.

(c) We would have to look up the distance to the node u and the distance between nodes u and v for every step in the recurrence. We would also have to access the maximum distance calculated to a node during the previous iteration i during the entirety of recurrence j. The return statement would be returning the maximum value calculated during any possible run for dist(v, i, j). We can do this by returning the maximum distance calculated in each iteration j and then compare all the value, or look through j return statements, to find the largest possible one.

(d) We can run j=0...n (where n is the total number of vertices) in any order and every iteration i=0...n-1 for each iteration j in any order as long as the source is specified.

(e) We have not shown P=NP because we only performed this on a directed, acyclic graph. We would need to show it on a general graph in order to prove that P=NP.

Question 2.2

(a) We can first eliminate every edge from contention that has $x_e = 0$ because while these could be a part of the matching, they cannot increase our total matching. Next, we can split the edges into two groups of magnitude 1 and magnitude 0. We can then create a network flow to represent this where edges are now represented as nodes. We can connect a dummy source node to every node that has $x_e = 1$ and every node that has $x_e = 1/2$ to a dummy sink node. Each edges would have capacity 1 for the source-1 edges and capacity 1/2 for the 1/2-sink edges. We then create edges between all the 1 nodes and all the 1/2 nodes representing if they share a common vertex. Each of these edges will have capacity 1. We can then calculate the maximum flow of the graph and that will be equal to the maximum integer flow. If this ends up being a multiple of 1/2 and not 1 then we can remove one of the 1/2 edges to be left with an integer flow.

(b) Eliminating all 0 edges only takes polynomial time at the beginning. Identifying every edge with magnitude 0 and magnitude 1 can be done in polynomial time, as can generating dummy edges and nodes for the max flow. We can calculate max flow with Floyd-Fulkeron which we know runs in polynomial time, thereby giving us a final polynomial runtime.

(c) If there is a fractional result in the end, we remove one edge that has weight 1/2 which can be done without disrupting the rest of the edges in the matching. Therefore, we cannot have a fractinal result. We will have a real matching because due to the capacity constraint of each edge in the flow graph, edges of capacity 1 can only connect with edges of capacity 1/2, meaning we will never place multiple $x_e = 1$ edges in the matching. Since each of these edges has capacity 1, it will reach a maximum of 2 $x_e = 1/2$ nodes, which add up to 1 total. Therefore, we will never place more than 2 $x_e = 1/2$ edges together, thereby ensuring we have a real matching.

(d) If the LP finds k edges, then in the worst case, at least 2/3 of the graphs' edges have $x_e = 1/2, not 1$. This means that 2/3*1/2 = 1/3 and the remaining 1/3*1=1/3, so (1/3+1/3)*k = 2k/3. Therefore, our algorithm has at least 2k/3 edges.

(e) If the graph only has edges of $x_e = 1$, then this will not work because the LP will return a value that is greater than any real matching as it is not a bipartite graph, resulting in a 'flow' of 0 in our algorithm.

I AM RECORDING MY PRACTICE EXAM ANSWERS HERE FOR PERSONAL REFERENCE. NO NEED TO LOOK HERE

Question 2

(a) The runtime of finding the min-cut is equal to that of finding the max-flow because once we have found the minimum cut we know what the maximum flow will be.

(b) Stable marriage runs in $O(n^2)$ time where n is the total number of elements in each set.

(c) We reduce X to Y because we need to show that Y is at least as hard as X. If we can reduce X to Y in polynomial time, then we have shown that Y is at least as hard as X, thereby making it NP-Complete.

(d) 2-coloring algorithm is not NP-Complete because for a general graph G, we can check if it is bipartite in polynomial time. Therefore, the only way this algorithm can be NP-complete is if P=NP, which we have not shown.

Question 4

(a)

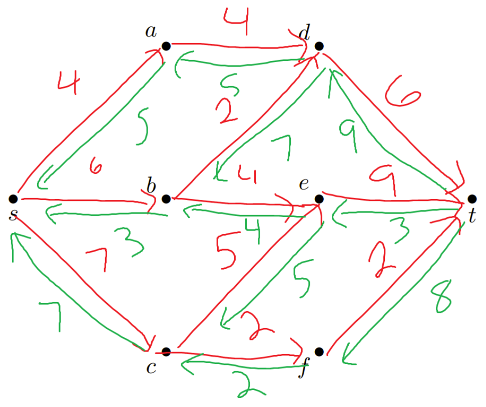

Figure 1: Residual graph for the current flow represented in the question

(b) We first set the flow in every edge to 0. Next, we identify a possible path from the source to the sink. Along that path, we calculate the maximum flow possible. For example, in the previous graph, the path (s, b), (b, d), (d, t) takes us from source to sink with a maximum flow of 9 through each edge. Next, we build the residual graph from this where each edge going to a node represents the flow going to it and we create a backwards edge that holds the leftover capacity. In this case, (s, b) = no forward edge and 9, (b, d) = no forward edge and 9, (d, t) = 6 and 9. The next path is (s, c), (c, e), (e, t). Each edge has capacity 10 because there are no backwards edges present. The residual graph has (s, c) = 4 and 10, (c, e) = no forward edge and 10, (e, t) = 2 and 10. The next path is (s, c), (c, f), (f, t). We can now maximize the edge (s, c) with capacity 14, so the residual graph will now set (s, c) = no forward edge and 14, (c, f) = no forward edge and 4, (f, t) = 6 and 4. The next path is (s, a), (a, d), (d, t). Edge (d, t) is no filled with capacity 15, so the other edges will have a flow of 9. The residual graph has (s, a) = 3 and 6, (a, d) = 3 and 6, (d, t) = no forward edge and 15. The next path is (s, a), (a, d), (d, b), (b, e), (e, t) which is formed by the back edge on (d, b). The residual graph has (s, a) = 1 and 8, (a, d) = 1 and 8, (d, b) = 2 and 7, (b, e) = 6 and 2, (e, t) = no forward edge and 12. We have now maximized the flow as we can no longer find a path from source to sink.

Question 7

(a) OPT(i, v) = 0 IF $i = 0$ AND $v = 0$

OPT(i, v) = ∞ IF $i = 0$ AND $v \neq 0$

OPT(i, v) = min(OPT(i-1, v), OPT(i-1, v-v_i)+p_i) IF $i \neq 0$ AND $v_i > v$

OPT(i, v) = min(OPT(i-1, v), p_i) IF $i \neq 0$ AND $v_i <= v$

(b) The base cases are when $i = 0$ and $v = 0$ or when $i = 0$ and $v \neq 0$. This is when we have no states and set it to either 0 or infinity depending on how much V is (0 or not 0).

(c) Our algorithm creates a 2-D array of size i+1 by V+1 where V is equal to the sum of every state's electoral votes divided by 2 plus 1 and i is the total number of states. We set the entire first row to ∞ except for at [0][0] which we set to 0. We then calculate each cell [i][v] using our recurrence and setting it equal to min(OPT(i-1, v) if $v_i <= v$ and min(OPT(i-1, v), OPT(i-1, v-v_i)+p_i) otherwise where v_i is the number of electoral votes for state i and p_i is the population of state i. We then return the value we calculate at the final cell, or [i][V] as the minimum population.

Question 10

To show that 4-SAT is in NP-Complete, we can show that 4-SAT is in NP-hard and assume that it is in NP. Let the following be an instance of 3-SAT: $N = x_1, x_2 ... x_n, C = c_1, c_2 ... c_n$ where N is the set of variables and C is the set of clauses. We can transform this into a 4-SAT instance by adding element y to N so $N = x_1, x_2 ... x_n, y$. If some element c_i in C is of the form l_1, l_2, l_3, to make this into 4-SAT we can set $c_i = l_1, l_2, l_3, yANDl_1, l_2, l_3, \neg y$. If 3-SAT is satisfiable, then 4-SAT will be as well because each c_i in the 4-SAT reduction is made up of the new element y and its complement $\neg y$. This means that one of the two will always be true. If 3-SAT is not satisfiable, that means one of l_1, l_2, l_3 was causing it to not be satisfiable, meaning regardless of y and $\neg y$, the c_i in 4-SAT will also not be satisfiable. Therefore, 4-SAT is NP-Complete as it is both in NP and NP-Hard.