Problem 1, CS38 Final, Matt Lim

The greedy algorithm will be as follows. At each step, we will pick the set covering the largest number of remaining uncovered items. Now, we will show that this achieves an approximation ratio of $\frac{e}{e-1}$.

Let OPT be the cardinality of the set of elements covered by a *maximum k-cover* - that is, the cardinality of the set with the maximum number of elements of U that can be covered by k of the subsets. Let r_i be the number of the OPT elements remaining (not yet covered) after iteration i; that is, if x elements have been covered after iteration j, then $r_j = OPT - x$. This means $r_0 = OPT$. Then we claim that

$$r_i \leq (1 - \frac{1}{k}) \cdot r_{i-1}.$$

The proof for this statement is as follows. We have that k subsets cover OPT elements. Then, at each step, k subsets cover all remaining elements (out of the OPT elements, not the universe). Thus, at each step, *some* subset must cover at least a $\frac{1}{k}$ fraction of those remaining elements out of the OPT elements. That is, at each step, there must exist some subset that covers a fraction $\frac{1}{k}$ (at least) of the r_{i-1} elements remaining to be covered. This basically means we can cover at least $\frac{r_{i-1}}{k}$ uncovered elements at each step i. And since our greedy algorithm picks the set covering the largest number of remaining uncovered items, we will cover at least $\frac{r_{i-1}}{k}$ uncovered remaining elements at each step i. Then, using this claim, we have that

$$r_i \leq (1 - \frac{1}{k})^i \cdot OPT$$

$$r_k \leq (1 - \frac{1}{k})^k \cdot OPT$$

$$r_k \leq \frac{OPT}{e} \text{ (since } (1 - \frac{1}{x})^x \leq \frac{1}{e})$$

So we have that, after k iterations of our algorithm (which means we have picked k sets) there are less than or equal to $\frac{OPT}{e}$ elements remaining of the OPT number of elements that are possible to be covered with k subsets. This means that c, the number of elements we have covered out of the OPT possible, is bounded below as follows:

$$c \geq OPT - \frac{OPT}{e}$$

$$c \geq OPT(1 - \frac{1}{e})$$

$$c \geq OPT(\frac{e-1}{e})$$

$$c(\frac{e}{e-1}) \geq OPT$$

Thus we get our approximation ratio of $\frac{e}{e-1}$.

Problem 2, CS38 Final, Matt Lim

Here is our algorithm. Note that a "child edge" (u, w) of a vertex u is an edge that goes from u to one of its children w.

Maximum-matching-tree(tree $G = (V, E)$)

1. Root the tree G at a node r
2. $S = \varnothing$
3. $OPT_{in}[v] = OPT_{out}[v] = 0$ for all $v \in V$
4. **foreach** node u of G in postorder
    5. **if** u is not a leaf
        6. **foreach** child edge (u, w)
            7. int $new = 1 + OPT_{out}[w] + \sum_{v \in \text{children}(u),\ v \neq w} max\{OPT_{out}[v], OPT_{in}[v]\}$
            8. **if** $new > OPT_{in}[u]$
                9. $OTP_{in}[u] = new$
                10. Store the following information in $B_{in}[u]$: edge $e = (u, w)$; for all children v of u (except w), if $OPT_{out}[v] \geq OPT_{in}[v]$ store $B_{out}[v]$ and if $OPT_{out}[v] < OPT_{in}[v]$ store $B_{in}[v]$; for w, store $B_{out}[w]$
        11. $OPT_{out}[u] = \sum_{v \in \text{children}(u)} max\{OPT_{in}[v], OPT_{out}[v]\}$
        12. Store the following information in $B_{out}[u]$: for all children v of u, if $OPT_{out}[v] \geq OPT_{in}[v]$ store $B_{out}[v]$ and if $OPT_{out}[v] < OPT_{in}[v]$ store $B_{in}[v]$
13. **if** $OPT_{in}[r] > OPT_{out}[r]$
    14. Backtrack starting with $B_{in}[r]$; for each $B_{in}[x]$ encountered, add the stored edge e to S
15. **else**
    16. Backtrack starting with $B_{out}[r]$; for each $B_{in}[x]$ encountered, add the stored edge e to S
17. return S

Note that these following sections depend on the fact that G is a tree.

We will now show that our algorithm runs in time $O(|E|)$ operations, where an operation is as defined in the problem. We will assume that $|E| \geq |V|$ (otherwise just reading in the graph would be greater than $O(|E|)$). First we iterate through all the nodes in line 3, of which their are order $|E|$ many of, and set some values. This is clearly within the time bounds. Then, we can see the combination of lines 4 and 6 runs through every edge in the graph. So, for every edge, we calculate a sum (line 7); then if the value of this sum is greater than the current value of $OPT_{in}[u]$, we set some variables. According to Lecture 17 slide 18 and our definition of an operation, we can can treat the sum in line 7 as a constant number of operations. Then, since line 10 just takes information from that sum and stores it, line 10 is also just a constant number of operations. Then for each non-leaf node (order $|E|$ many), we also calculate another sum in line 11, and set another variable in line 12. For the same reasons as lines 7 and 10, these two lines are also just a constant number of operations each. Finally, the backtracking at the end just goes through all the B values for each node and sometimes adds edges to S, which is clearly $O(|E|)$. Then, we have that we perform a constant number of operations for each of order $|E|$ many things (all nodes, edges, non-leaf nodes). Thus this algorithm runs in time $O(|E|)$ operations.

Now we will show that our algorithm works. We can see that we visit the nodes in postorder. This is to ensure a node is visited after all its children. For each node u, we consider what happens, each child

edge e at a time, if we add e to the matching and what happens if we don't add any of its child edges to the matching. The former part occurs in lines 6-10. In line 7, we consider adding an edge (u, w) to the matching. This would mean that our matching cardinality would go up 1, that w could not have any of its child edges in the matching, and the other children of u (not w) can have one of their child edges in the matching or not. After calculating each such possibility (one possibility for each child edge (u, w)) we make it the new value of $OPT_{in}[u]$ if it is greater than the current value. Doing this in the for-loop ensures we get the maximum score for including one child edge of u. The latter part occurs in line 11, where we calculate the maximum score of when we don't include any of u's child edges in the matching. In this way, we compute the value of a maximum matching for the subtrees rooted at each node (it will be the maximum of OPT_{in} and OPT_{out}). Thus, we compute the value of a maximum matching for a tree rooted at r, which is just the value of the maximum matching for the whole tree. Now we must explain why our backtracking works. We have that, for each node u, we store $B_{in}[u]$ and $B_{out}[u]$. $B_{in}[u]$ contains the optimal child edge (u, w) of u to add to the matching, and the optimal in/out choice for all child vertices of u (for w, the choice is by default "out") stored as B values. $B_{out}[u]$ contains the optimal in/out choice for all child vertices of u stored as B values. Thus, when starting from the root node r, we have the optimal choices (depending on whether or not we include a child edge of r) for all its child vertices contained in $B_{in}[r]$ and $B_{out}[r]$. Then, we know which one is better to start with from the values of $OPT_{in}[r]$ and $OPT_{out}[r]$. Once we pick the correct B to start with, it is clear that we can continue optimally down to the bottom of the tree; at each level, we have the B values for all its nodes (obtained from the previous level, unless we are at the top level), from which we know the optimal choices/B values for the next level. And since we add the e values stored by B_{in}'s, which we know are optimal, we get the maximum matching.

Problem 3, CS38 Final, Matt Lim

Here is the pseudo-code of our algorithm to find a maximum sum subsequence. Note that it does not use divide-and-conquer; **I asked Professor Umans and he said it is OK as long as it runs in the desired time bounds.** Let the size of the list be n, and let L_k denote the kth element of the list. Note that we will let the "value" of a list be the sum of its elements.

Maximum-sum-subsequence(List L)

1. check if L is all non-positive elements by iterating through every element. While iterating through, keep track of the maximum element's index. If L is all non-positive elements, return index of maximum element.
2. int $curr = max = 0$
3. int $begin = end = beginMax = endMax = 1$
4. **for** $1 \leq k \leq n$
    5. **if** $curr + L_k > max$
        6. $max = curr + L_k$
        7. $curr = curr + L_k$
        8. $beginMax = begin$
        9. $endMax = k$
    10. **elif** $curr + L_k > 0$
        11. $curr = curr + L_k$
    12. **else**
        13. $curr = 0$
        14. $begin = end = k + 1$
15. return $beginMax, endMax$

We will now see why this runs in the desired time bounds. We have that step 1 takes $O(n)$ operations, as it just iterates through the list. Then, the rest of the algorithm is basically just a for-loop that iterates through all n elements of the list. In the for-loop, we are just adding integers of magnitude $O(n|\max_i a_i|)$, comparing values, and/or setting integer values. And in each iteration of the loop, the number of these operations that happen is clearly bounded by a constant. So we have that overall, our algorithm only takes $O(n)$ operations, which meets the desired bound of $O(n \log n)$ operations.

Now we will explain why this algorithm works. The first line takes care of lists that have no positive values by simply returning the maximum value. Otherwise, we do the following. We start from the beginning and iterate through the whole list. We keep track of the starting and ending indices of the current subsequence $(begin, end)$, the starting and ending indices of the current maximum subsequence $(beginMax, endMax)$, and the values for the current subsequence and the maximum subsequence $(curr, max)$. For every element, we add its value to $curr$. Let the element we are considering be called L_k. So, we will add that element to $curr$ to get $sum = curr + L_k$. If sum is greater than max, we set the current subsequence to be the maximum. If it is not greater than max but greater than 0, we maintain our current subsequence. Else, if sum is less than or equal to 0, we start a new subsequence on the next element. We do this because once the value of a current subsequence is non-positive, we know that it is better or the same to start a new subsequence on the next element than to continue on with the current one, as non-positive values do not help the sum. So overall, our algorithm considers a subsequence until its value becomes non-positive (keeping track of the maximum along the way), and when its value does become non-positive, it considers a new subsequence starting on the next element. Thus, our algorithm only elongates the current subsequence if it is possible that it is an essential part of the maximum (that it actually adds to the value), all while keeping track of the current maximum subsequence - otherwise,

it starts a new one, since extending the previous one cannot help. In other words, we never make an outright bad move, only moves that are or have the possibility of being beneficial. And if we make a possibly beneficial move that hurts the current value, like adding a negative to our subsequence, we have stored the value of the max before that move. Thus, overall, our algorithm keeps track of how high a value a subsequence can have until its value hits zero or less, then starts again on the next element. So clearly, since we run this algorithm starting at the beginning of the list, it will not miss finding a maximum sum subsequence.

Problem 4, CS38 Final, Matt Lim

(a) Given the LP, our constraint matrix A will be as follows. Letting $|V| = n$ and $|E| = m$, we have that our constraint matrix A will have m rows and $m + n$ columns. The m rows will represent the m edges. The first m rows will also represent the m edges, and the last n rows will represent all the vertices. Looking at a row that represents edge $e_i = (u, v)$, there will be a 1 in the column that also represents edge e_i. Also, there will be a -1 in the column that represents vertex u, and a 1 in the column that represents vertex v. All other columns in that row will have a 0. Thus, with this constraint matrix in place, we can see that multiplying it by the column vector x with the edge variables on top (same order as the edges in the constraint matrix) and the vertex variables on the bottom (same order as the vertices in the constraint matrix) and constraining the product to be greater than or equal to a column vector of 0s, we meet our desired constraint for each edge. We will now prove that A is unimodular.

Now we will prove by induction that A is unimodular. To do this we will prove that every square submatrix of A has determinant 0, 1, or -1. First we will consider the base case. So, consider any submatrix of A that has dimensions 1×1. We showed above that every cell of A is either 0, 1, or -1. And since the determinant of a 1×1 matrix is just the value of the cell, we have that every 1×1 submatrix of A has determinant 0, 1, or -1. Now we will do our inductive assumption. So, assume that every $k \times k$ submatrix of A, where $1 \leq k \leq i - 1$ has determinant 0, 1, or -1. Now we must show that every $i \times i$ submatrix of A has such a determinant. So consider an arbitrary $i \times i$ submatrix B of A.

Now we will consider two cases. The first case is when a column c in B is made of all zeroes. Then we have that the determinant of B is zero (expand by cofactors along c). The second case is when a column c in B has one 1 or one -1 and 0s everywhere else. Here we have that the determinant of B is either 0, 1, or -1. We can see this is true if we expand by cofactors along c. This is because by calculating the determinant this way, it is the same as if we calculate the determinant of the $(i-1) \times (i-1)$ matrix we get by crossing out column c and the row with the one $1/-1$ in column c and multiplying it by 1 or -1. And by our inductive assumption, the determinant of such a $(i-1) \times (i-1)$ matrix is 0, 1, or -1. So clearly the determinant of B is 0, 1, or -1. Note that these first two cases cover every square submatrix that has a column from the first m columns of A, since all those columns only have one 1 (since we don't include the same edge twice). Thus, when we take a submatrix that includes one of those columns, the corresponding column in the submatrix will have at most one 1.

The next cases cover submatrices that only take columns from the last n columns of A, since, as we noted above, the first two cases cover all cases when a column from the first m columns of A is included. So, we get the following cases. Similarly to the first two cases, we have a third case in which one row of B is made of all 0s and a fourth case in which one row of B is made of all 0s and one 1 or -1. These cases are proved true in the same way as the first two cases, except that instead of expanding along a column we expand along a row. Then, we have a final fifth case, in which every row of B has both one 1 and one -1 and every column has at least 2 non-zero values. Note that we never get a case where there are two 1s and one -1 in a row, because that would involve including a column from the first m columns (a case which we already took care of). We will now claim that in this case, it is possible to make a row of all zeroes by adding multiples of rows to other rows. We can see this is true because of the following reason. Since we have two non-zero values in each column (two 1s, two -1s, or one 1 and one -1) and exactly one 1 and one -1 in each row, we can do the following. Let us add rows to the first row. Then, we can just keep on adding/subtracting rows, getting rid of one ± 1 in a column at least one column at a time. Note that, every time we add/subtract a row to/from the first row, the overall value of that row stays 0, since we are adding a 1 and subtracting a 1 either way. Therefore, the first row will eventually become all 0s, since we have two non-zero values in each column that we can have cancel and we can always keep the value of the first row at 0. For an example, see the following:

$$\begin{bmatrix} -1 & 1 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 1 & -1 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & -1 & 1 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \rightarrow \begin{bmatrix} 0 & 0 & 0 & 0 \\ -1 & 0 & 1 & 0 \\ 0 & -1 & 0 & 1 \\ 0 & 0 & -1 & 1 \end{bmatrix}$$

First we subtract the second row, then we add the third row, then we subtract the fourth row. We can see that, since we have two ± 1s in each column, we can just cancel all of them out in the first

row. So, we have that it is possible to make a row of all zeroes by adding multiples of rows to other rows. Note now that it is a theorem of linear algebra that the determinant of a square matrix is unchanged if the entries in one row are added to those in another row (page 1224 CLRS). Then, since we have a row of all zeroes, we can expand along this row and get that the determinant of B is 0. So for this case, the determinant is 0. Now we have that all the cases are covered. Thus, we can conclude by induction that every square submatrix of A has determinant 0, 1, or -1, and thus that A is totally unimodular.

(b) We are given that this LP is the dual of the max-flow LP. Thus, the minimum/optimum value given by this LP equals the maximum value given by the max-flow LP. This is by the strong duality theorem given in lecture. We can apply this because, clearly, A and b are real-valued. c is also clearly real valued, as we have integer capacities for each edge. And clearly the dual and primal are nonempty. Then we have that the optimum value of this LP equals the value of the max-flow. By the max-flow min-cut theorem, the value of the max-flow equals the capacity of the min-cut. Thus we have that the optimum value of this LP equals the value of the min-cut.