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. node k
3. $S = \varnothing$
4. int $min = \infty$
5. $OPT_{in}[v] = OPT_{out}[v] = 0$ for all $v \in V$
6. **foreach** node u of G in postorder
    7. **if** u is not a leaf
        8. **foreach** child edge (u, w)
            9. **if** $OPT_{in}[w] - OPT_{out}[w] < min$
                10. $min = OPT_{in}[w] - OPT_{out}[w]$
                11. $k = w$
        12. $OPT_{in}[u] = 1 + OPT_{out}[k] + \sum_{v \in children(u), v \neq k} max\{OPT_{out}[v], OPT_{in}[v]\}$
        13. Store the following information in $B_{in}[u]$: edge $e = (u, k)$; for all children v of u (except k), 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 k, store $B_{out}[k]$
        14. $OPT_{out}[u] = \sum_{v \in \text{children}(u)} max\{OPT_{in}[v], OPT_{out}[v]\}$
        15. 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 $B_{in}[v]$
        16. $min = \infty$
17. **if** $OPT_{in}[r] > OPT_{out}[r]$
    18. Backtrack starting with $B_{in}[r]$; for each $B_{in}[x]$ encountered, add the stored edge e to S
19. **else**
    20. Backtrack starting with $B_{out}[r]$; for each $B_{in}[x]$ encountered, add the stored edge e to S
21. 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 5, of which their are order $|E|$ many of, and set some values. This is clearly within the time bounds. Then, with the combined for-loops of lines 6 and 8, we iterate through every edge in the graph and set/compare some variables for each edge. This is also clearly within the time bounds. Then we can see that in lines 12 and 14, we perform a sum for each non-leaf node. For each node, these sums sum over its child edges. Thus, overall, each of these sums ends up summing $O(|E|)$ many elements, each of which are of magnitude $O(|E|)$ (since the value of any OPT value is bounded by the total number of edges). So, the time it takes to do both of these sums is $O(|E|)$ operations. Next, we have that in lines 13 and 15 we store values from those sums; thus,

since we showed the sums are $O(|E|)$, so are these lines (overall, each set of B values - B_{in}s and B_{out}s - stores information about every vertex, and storing each thing takes constant time since at most we need to compare two values to store something). 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 $O(|E|)$ (since this means we backtrack through the B values of all the nodes, of which there are order $|E|$ many). 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 the optimal score of adding a child edge e of u to the matching and the optimal score if we don't add any of its child edges to the matching. The former part occurs in lines 8-12. We choose to add the edge between u and the child vertex w such that $OPT_{in}[w] - OPT_{out}[w]$ is minimized. This is to minimize the loss of score when adding that edge to the matching, which is the same as maximizing the score of the matching. The latter part occurs in line 14, 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 non-leaf node (it will be the maximum of OPT_{in} and OPT_{out}); for leaf nodes the value is always 0 since they have no children. 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, when we are backtracking, we add the edges stored by B_{in}'s, which we know are optimal, we get the maximum matching. Note that our backtracking never adds an edge that breaks the matching, since when we include a child edge (u, w), the B value ensures that w does not have a child edge added to the matching (because we store B_{out} for w). So overall, we maximize the number of edges added to S while still keeping it a 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 and update some values. If it is not greater than max but greater than 0, we maintain our current subsequence and update some values. Else, if sum is less than or equal to 0, we start a new subsequence on the next element and update some values. 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 and location of the max subsequence before that move. Thus, overall, our algorithm keeps track of how high a value a subsequence can have (while keeping track of the position of the maximum value subsequence) 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 $m + 2 + m$ rows and $m + n + m$ columns. The first m rows will represent the m edges (first m rows will be $e_1, \ldots, e_m$), the next two rows will be for the middle two constraints, and the last m rows will represent the m edges again (same order as first m edges). The first m columns will represent edges as well (same order as rows), the next n columns will represent all the vertices, and the last m columns will represent all the edges again (same order as rows). Looking at a row in the first m rows that represents edge $e_i = (u, v)$ where $u \neq s$, there will be a 1 in the column (from the first m columns) 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. If $u = s$, then the -1 in the column that represents u will be replaced with a 0. Next, the first row in the middle two rows will have a 1 in the column that represents vertex s (0s everywhere else), and the second row will have a 1 in the column that represents vertex t (0s everywhere else). Finally, looking at a row in the last m rows that represents edge $e_i = (u, v)$, there will be a 1 in the column (from the last m columns) that also represents edge e_i and 0s everywhere else. We can then see that multiplying this by the vector $x_1, \ldots, x_m, y_1, \ldots, y_n, x_1, \ldots, x_m$ and having a vector c that constrains the results of the first m rows to be greater than or equal to 1 for the edges that go out from s and 0 for those that don't, the result of the $m + 1$ row to be equal to 1, the result of the $m + 2$ row to be equal to 0, and the results of the last m rows to be greater than or equal to 0 (where a "result" is the number you get by multiplying a row of A by x) gives us all our constraints. 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 four 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. 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. Note that these first four cases cover every square submatrix that includes a row from the $m + 1$st row down. This is because all those rows only have one 1. Also note that these first four cases cover every square submatrix that includes a column from the first m columns from A and/or from the last m columns of A. This is because all those columns only have one 1. This means we have covered all submatrices except for those from the top middle $m \times n$ submatrix.

To cover this part, we get 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. 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 ± 1s 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 and exactly one 1 and one -1 in each row, 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) First, we will show that the dual LP of this given LP is the max-flow LP, and that its maximum value is the value of the max flow. So, given our LP, we can say the following. We already defined A^T. Now we have that y is a column vector of height $m + n + m$ that just contains the edge variables (x_es) and the vertex variables (y_vs) and the edge variables again, in that order. We have that c is a column vector of height $m + 2 + m$ that, in the first m rows, contains a 1 for each edge that goes out from s and 0 everywhere else, in the $m+1$ row contains a 1, in the $m+2$ row contains a 0, and in the last m rows contains all 0s. Finally, we have that b is a column vector of height $m + n + m$ that contains all the edge capacities in the first m rows, then all 0s in the next $n + m$ rows. Turning this into the dual gives us the following. We have that A has $m + n + m$ rows and $m + 2 + m$ columns. The first m rows represent edges, and will have a 1 in the column (from the first m columns) that represents the same edge as the row. The next n rows represent vertices; each row will have a -1 for each edge going out from it and a 1 for each edge going into it (only in the first m columns). The two exceptions are the rows that represent s and t; the row that represents s will have an extra value of 1, and the row that represents t will have an extra value of 1 as well. These two values are in the middle two columns. The last m rows again represent edges; each of these rows will have a 1 in the column (from the last m columns) that represents the same edge as the row. For all these rows, all values that were not specified are 0. So, that is A. We can have x be a column vector of height $m + 2 + m$. The first m rows will indicate how much flow is sent through each edge, the next two rows will be 0, and the last m rows will again indicate how much flow is sent through each edge. b is just the same as in the dual (but we will add slack variables), as is c. Clearly, we maximize the right thing, as $c^T x$ is the sum of the flows coming out from the source vertex s. Then, we can then see that constraining $Ax = b$, where we subtract non-negative slack variables to the first m rows of b, subtract a non-negative slack variable from the row that represents vertex s, add a non-negative slack variable to the row that represents vertex t, and add non-negative slack variables to the last m rows of b gives us all the constraints we want. This is because multiplying the first m rows by x and setting them equal to the corresponding values in b gives us the constraint that the flow in an edge never exceeds the edge's capacity, for each edge. Then, multiplying the next n rows by x and setting them equal to the corresponding values in b gives us the constraint that the flow going into a vertex equals the flow going out of a vertex, for each vertex that is not s or t. For vertex s we just get the constraint that there is a non-negative net outflow, and for vertex t we just get the constraint that there is a non-negative net inflow. Finally, multiplying the last m rows by x and setting them equal to the corresponding values in b gives us the constraint that all the flows are non-negative. Thus, we have shown that the dual of the given LP is the max-flow LP, and that the value the max-flow LP gives us is indeed the value of the max flow.

Now we can use what we just proved in the following way. We have that the minimum/optimum value given by this LP equals the maximum value given by the max-flow LP (which is just the value of the max-flow). This is by the strong duality theorem given in lecture 15, slide 49. We can apply this because A/A^T and c are real-valued (both just contain integers). b is also clearly real valued, as we have integer capacities for each edge. And clearly the dual and primal are nonempty. Therefore we have that the optimum value of this LP equals the value of the max-flow. Then, 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.