I completed this without a study group.

1 MSTs

a. Have Kruskal's initially add all the edges to the MST (and thus the union-find data structure keeping track of connected components) before sorting the edges. At the same time, remove these edges from the original graph's edge set. Then proceed as normal. Justification: we satisfy the requirement that all the edges in E' Adding these edges essentially creates a new graph of connected components (think of each connected component as just another vertex). Because Kruskal's is guarenteed to work on any graph, this will now find the MST connecting these components.

b. Search through the graph and find the minimum weighted edge. Add the absolute value of this edge to every other edge making every other edge positive. Run Kruskal's on the new graph. The resulting MST is the correct MST in all cases. Justification: every spanning tree has the same number of edges (because trees do - —V— - 1 edges). Adding weight to each edge therefore has no difference because you aren't punishing or rewarding using fewer edges (like if you did this in Dijkstra's).

c. Negate every edge weight. Run Kruskal's. The resulting MST is the correct MST for the original graph. By negating edges you are making the biggest edges the smallest and the smallest edges the biggest. Because Kruskal's finds the smallest spanning tree, by inverting the weights it will now find the largest spanning tree. The total weight of the actual MST is the total weight of the negated MST * -1. By minimizing the value of the negated MST we get the largest possible value for the actual MST.'

2 Prim's Algorithm

3 Huffman Coding

a. $mlog(n)$ - It takes log(n) bits to represent each character that way and there are m of them.
b. The character with the lowest Huffman encoding occurs m - n + 1 times. The other n - 1 characters occur once. If you do it any other way, you would be able to switch the character with the smallest Huffman encoding with a character that occurs multiple times. This can only make the total number of bits smaller (or stay the same if their is a tie for best). Therefore, no other file description can be optimal.
c. Every character has the same distribution. The efficiency will approach 1 as n gets very large. This is because Huffman encoding will produce an encoding very close to one that assigns log(n) bits to each character.

4 Horn Formulas

Main Idea: Use a hashmap to map every variable to the implications that it occurs in on the left hand side (these implications can be stored as a set). When it is determined that a variable must be true, go through each implication in which it occurs in and remove the variable. If there are no variables left on the left hand side you know the right hand side must be true as well (the left hand side is really equivalent to 1 and 1 and 1 ...). Keep going until there are no implications that have no variables on their left side. Check if all the negative ors are still satisfied. If they are, return the values, otherwise return not possible.

Pseudocode:

```
procedure hornSolver(implications, ors)
    create map<vertex, set<implication>> map
    trues = {}

    toRemove = {} // variables needed to be removed
    for implication in implications
        for var in implication
            if var in map
                add implication to set in map.var
            else
                create new set and put in map.var
                add implication to map.var

    for implication in implications
        if implication.left.size == 0
            add implication.right to toRemove
            trues.add(implication.right)

    while toRemove is not empty:
        currVar = toRemove.pop()
        for implication in map.get(currVar)
            implication.left.remove(currVar)
            if implication.left.size == 0
                add implication.right to toRemove
                trues.add(implication.right)

    if check_not_ors(ors, trues)
        return trues // everything else can assumed to be false
    else
        return not possible
```

Justification: Most of this was proven in lecture. All I have to justify is that this correctly evaluates whether an element must be true. The only way the left side is true is if every element is true. true and any other element is just the

other element, so removing trues is equivalent. When there is nothing left everything must be true, so the right side of the implication must additionally be true.

Runtime: For the rest of this part let n be the number of literals that occur in all the clauses. This algorithm runs in time $\Theta(n)$. Creating my map takes $\Theta(n)$ time because you visit each literal in the implication exactly once. Scanning the size of the implications takes time proportional to the number of implications which must be less than the number of literals. During the last while loop, I will visit every literal that must be true exactly once (after I remove it I obviously can't visit it again), so if every literal must be true it will take n time. Removing a literal from an implication can take constant time if the left side is implemented with a hashset (or anything with constant lookup) and getting the size can also take constant time. Comparing to the ors only requires you to look at each literal once, so the overall time is linear.

5 Money Changing Redux

a. **Main Idea**: The main idea is to use dynamic programming to "build up" to the correct solution. We start at counting the number of ways to make 0 which is just 1. Then for each subsequent number, compare that number - each denomination to see which one requires the least number of coins. Do this until you calculate the least number of coins for A.

Pseudocode:

```
procedure count_change(A, denoms):
    array = make array of size A
    array[0] = 1
    for i = 1 to A
        min = INFINITY
        for denom in denoms
            if i - denom > 0 and array[i - denom] < min
                min = arr[i - denom]
        array[i] = min + 1
    return array[A]
```

Proof of Correctness: Consider a strong inductive argument. The base is obviously true (there is only 1 way to produce 0). Assume it's true for all numbers 0 to A. Then for A + 1, you have already calculated the min of A + 1 - each denomination. The minimum number of coins is just 1 + the of ways it takes to go 1 denomination smaller.

Runtime Analysis: In this algorithm we go through A numbers and check every denomination so it takes An iterations where A is the number and n is the number of iterations. Comparing to previously calculated values takes constant time so that is the justification for the An runtime.

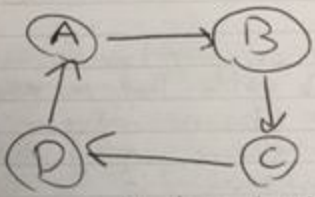
b. Because the number of bits to represent A is only log_2 the number of bits, the overall runtime is actually $2^A n$ because sA is much bigger than its actual size.

HW 4

Problem 2

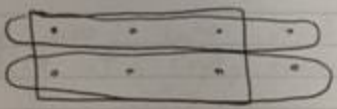
a. Yes. If there is a path from v_i to v_j then v_i must be visited before v_j in a topological ordering because it is either a prereq (meaning leads directly into) v_j or it is a prereq for one of v_j's ancestors (meaning it must come before the ancestor and thus before v_j)

b. No, consider:



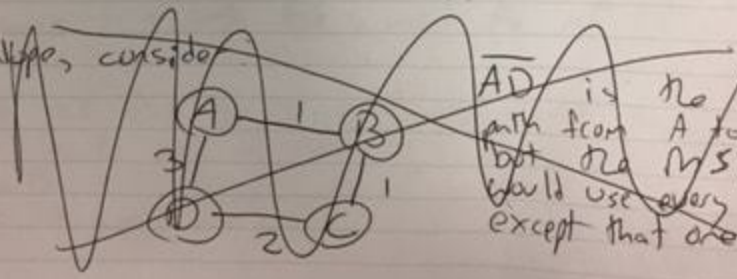
Removing any edge would break the cycle and turn this graph into a DAG (making it obviously impossible to reach every vertex from leaves).

c.



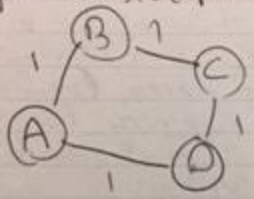
Greedy algo will take all 3 while the optimal is only both groups of 4

d. Nope, consider



$\overline{AD}$ is the shortest path from A to D but the MST would use every edge except that one.

2. Nope consider:



Any 3 ~~valid~~ edges are a valid MST. D minimizes dist(s,v), however $\overline{AD}$ doesn't need to be in MST because any 3 edges are valid (ie $\overline{AB}$, $\overline{BC}$, $\overline{CD}$).

c. Yes by the cut property. Because the edges are all positive, the vertex that minimizes d(s,v) must be a neighbor (because adding an additional edge only makes the distance worse). Because its a neighbor we are now asking if the shortest edge incident to s must be included. Consider the coloring in which every vertex is blue except s which is red, then the smallest edge crossing the cut is the one I described.

Problem 3