Question 1

1 1.1

(a) Let our greedy algorithm solution be the set C = $c_1, c_2...c_k$ where each element of the set is a coin; Also let the optimal solution where the total number of coins is minimized be the resulting set O = $o_1, o_2...o_n$ where each element of O is a coin.

The set O will never contain more than b elements because by definition, $b^1 = b(b - once), b^2 = b * b(b - twice), b^3 = b * b * b...(b - thrice)...b^k = bk - times$. If we ever have b of any coin (let us say coin valued b^m for some integer $m >= 0$), then we can replace it with one coin of b^{m+1} instead in the minimum coin set, since this gives us b^{m-1} less total coins. We can repeat this idea with every possible coin to confirm that in the end, there will be less than b of each coin.

In the greedy algorithm, we first select the highest value coin (let us call this c_x) that is less than n and add as many copies of it as possible without surpassing n. We then select the highest value coin that when added to c_x is less than n and add as many copies of it as possible again. We repeat this going down to b^0, or 1, until our total is equal to n. This means that there are no more than b of any one of these coins, since b of any coin b^k would have been replaced by b^{k+1} earlier. Starting from the highest-value coins results in less total coins because less higher-value coins are necessary to total the same price as multiple lower-value coins.

If we follow through with every coin until we sum to n, then we get the exact same number of coins in our set C and the set O since the pattern of addition results in the same quantity and value of coins. Therefore, our greedy algorithm produced the minimum number of coins to make n cents of change.

(b) I think the proof outline that mine mimics closest is the structural proof. This is because I am comparing the end result of my algorithm and the most optimal algorithm to ensure that they have the same total number of coins in the end. In other words, I am comparing them to make sure they have the same ending structure.

2 1.2

(c) In part a), we said that there are no more than b of any one coin, since b of any coin b^k would have been replaced by b^{k+1} earlier. The same does not hold true for this set of coins, $1, 10, 15$. For example, to make a total of n=20, the greedy algorithm would start with 15, then add 5 1s (6 coins total), when the truly minimum set is 10 and 10 (2 coins total). The flaw arises in assuming that starting with the highest value coin works best among sets like this. There is no guarantee that the highest priced coin will even be involved in the minimum coin set, so greedy does not work for this type of coin set.

(d) If we were to add a 80-cent coin, then we would not optimally create a total of n=160 using the greedy algorithm. The greedy algorithm would first add 100, then 50, then 10 (3 total coins).

However, the minimum total is actually two 80-cent coins (2 total coins).

Question 2

(a) To begin, we will iterate through the entire array of pizzas and create a map that keeps track of all the possible pizzas and the number of votes for each pizza type (0 votes at the beginning for all). This part will only happen once and will take O(n) time. Next, we begin the divide-and-conquer part of our algorithm by splitting the array in each step by 2 every time until every array is in its own 1-element sub-array. This part of the recurrence will take log(n) time. Next, for every one-element sub-array we will increment the count of whatever pizza type is in that array. Since we are comparing elements by our own criteria (not hashcode), we will increment the counts of all pizzas in the map with the same ingredients. Now we have finished dividing and have a count of all the pizzas in separate arrays, so we will recombine them. In every step, we can compare the total counts of every pizza together and return this new map of pizza to counts until we combine back to our n-size array. Since we have finished recombining now, we can iterate through the map to check for which pizza has the largest count. If this number is greater than n/2, then we return that pizza type. If not, we return cheese.

(b) We will prove that this algorithm works using proof by induction. We will assume that the number of elements in the array of pizzas is of size n where $n = 2^k$ from some integer k and we only have to identify this works for when a non-cheese pizza is ordered.

If we let k=0, or n=1, then our algorithm will return that the maximum count is 1 as the only vote is for the one pizza that is in our array. Since $n/2 = 1/2 and 1 > 1/2$, we will return this pizza. Therefore, since when we only have one pizza preference we would want to return that pizza, this statement holds true for k=0 by inductive hypothesis.

Let us assume that we return the correct pizza type for some integer k where $n = 2^k$. Then to show that we return the correct pizza type for when $n = 2^{k+1}$, we can do the following:

By the nature of our divide-and-conquer algorithm, we will begin by splitting the n-size array in half to n/2, then n/4, and so on until each sub-array is size 1. We know that the algorithm works correctly for k, and for when $n = 2^{k+1}$, we only need to split the array one more time so that each sub-array has 2^k elements, after which we can use the algorithm for 2^k. After we finish the algorithm for everything until $n = 2^k$, we have divided-and-conquered for the entirety of $n = 2^k$, so we are left with two sub-arrays each of size 2^k. We only need to combine this one more time to reach the final step in our algorithm for $n = 2^{k+1}$. At this stage, we will just return the total counts for each $n = 2^k$ array, which we know to be accurate, and simply add the results together. Since this combination repeats the same logic as the final step of $n = 2^k$, it is correct in calculating the total for each pizza. The pizza in question will then be that with the highest count as we will either order that or cheese, so we only have to compare this highest count to n/2. If it is greater than 2^k, or if more than half the TAs agree, then we return that pizza. If not, we will return cheese.

(c) -To begin, we will iterate through the entire array of pizzas and create a map that keeps track of all the possible pizzas and the number of votes for each pizza type (0 votes at the beginning for all). This part of the algorithm runs in O(n) time as we will iterate through and create all n elements (even if some of them turn out to be the same in the future) just once. -Next, we begin the divide-and-conquer part of our algorithm by splitting the array in each step by 2 every time

until every array is in its own 1-element sub-array. This part of the recurrence will take log(n) time as we only need to cut each array in half.

-Next, for every one-element sub-array we will increment the count of whatever pizza type is in that array. To do this, we have to compare the element with every element in the map which will take a worst-case of O(t*n).

-Now we have finished dividing and have a count of all the pizzas in separate arrays, so we will recombine them. This part of the algorithm takes O(t*n) as well time since we will need to combine the arrays by comparing elements a maximum of n times.

- Since we have finished recombining now, we can iterate through the map in O(n) time to check for which pizza has the largest count. If this number is greater than n/2, then we return that pizza type. If not, we return cheese (the returns and check take constant time).

The recurrence itself will be $T(n) = 2T(n/2) + O(t * n)$ if $n >= 2$, O(t*n) otherwise. Since by master theorem $log_2(2) = 1$ and the power of n and t*n are the same, this runs in O(t*n*log(n)) time.