Problem 1 - Sketching

a. The algorithm for sketching is essentially the following: define two variables, an upper bound and a lower bound, and set them to infinity (or some NULL value). Then, for each pair (x_i, y_i), if $y_i - \epsilon >$ the upper bound, or if $y_i + \epsilon <$ the lower bound, the error tolerance will be breached and a new interval is started (i.e. the upper bound and lower bound return to infinity/NULL), with the data of the old one (start can be stored in a value, finish will be $y_i - 1$, height will be (upper bound + lower bound) / 2) stored in a data structure. Otherwise, if the error tolerance isn't breached, set the upper bound to $min(upperbound, y_i + \epsilon)$, and the lower bound to $max(lowerbound, y_i - \epsilon)$. This essentially guarantees that every new y value being examined will either not touch the bounds at all - keeping them the same - or bring them closer together, denoting as the interval goes on where points will be out of the error range. In pseudocode, the algorithm should look like:

$upper, lower = NULL$
$i, s = 1$
while $i \leq n$ **do**
  if $y_i - \epsilon > upper$ OR $y_i + \epsilon < lower$ **then**
    Store an interval from s to $i - 1$, with height $\frac{upper+lower}{2}$.
    $s = i$
    $upper = y_i + \epsilon$
    $lower = y_i - \epsilon$
  else
    $upper = min(upper, y_i + \epsilon)$
    $lower = max(lower, y_i - \epsilon)$
  end if
  $i = i + 1$
end while
Store the last interval from s to n, with height calculated as above.

b. I'm going to take this very atomically and piece by piece, so I apologize in advance, because it will be wordy.

First, let's prove that the algorithm terminates - the only loop has a finite, positive, unchanging bound and the condition is guaranteed to increase every iteration. Regardless of what the bounds are doing, the algorithm goes one point at a time and eventually does finish its calculations when it reaches the end.

Next, let's prove that the lower bound is always less than or equal to the upper bound while each interval is being computed (why not? It might even come in handy). Every interval starts out with the upper bound and lower bound being assigned to the first point's value $+ - \epsilon$, as evidenced with $s = i, upper = y_i + \epsilon, lower = y_i - \epsilon$. From that point onward (until the interval ends, that is), in each step the upper and lower bounds stay the same, or they are replaced with a new value - for the upper bound, always less than the current value, and for the lower bound, always more than the current value. This replacement can only happen if the condition that $y_i - \epsilon > uppper$ OR $y_i + \epsilon < lower$ is not met - essentially, it must be the case to change values that $y_i - \epsilon \leq upper$ and $y_i + \epsilon \geq lower$. Additionally, the condition on the actual assignment specifies that to change the value of upper, $y_i + \epsilon \leq upper$, and to change the value of lower, $y_i - \epsilon \geq lower$. Phrased together, this means that the assignment of a new upper bound is $y_i + \epsilon$ such that $y_i - \epsilon \leq y_i + \epsilon \leq upper$ (ϵ is always a positive/zero value) and that the assignment of a new lower bound is $y_i - \epsilon$ such that $lower \leq y_i - \epsilon \leq y_i + \epsilon$. Without loss of generalization this means that at any point where *upper* could possibly be assigned to be lower than *lower*, it would also need to be lower than $y_i - \epsilon$ - the condition for ending that interval, thus preventing such a situation from ever happening. As this can't happen in any given step, it can't happen at all, and the same applies for lower overstepping upper.

Third, let's show that each interval ends with a height from which no point is greater than ϵ away. Intervals are created immediately after the condition that $y_i - \epsilon > upper$ OR $y_i + \epsilon < lower$, meaning that for every step in this interval preceding, the opposite has been true - as seen in the above paragraph, $y_i - \epsilon \leq upper$ AND $y_i + \epsilon \geq lower$. Additionally, we know that each interval begins with $upper = y_i + \epsilon$, $lower = y_i - \epsilon$ - as such, the height of the interval at this point $(y_i + \epsilon + y_i - \epsilon/2)$ is equal to the actual height of the point itself, y_i (this is fairly intuitive). Assume the inductive hypothesis that for every point in the interval following the first one for which $y_i - \epsilon \leq upper$ AND $y_i + \epsilon \geq lower$, the height calculated using the new upper and lower bounds adjusted from that most recent point is no more than ϵ away from any previous point in the interval. I've just proved the basis, for the first point, and we can assume via the inductive hypothesis that every point between the first point and the one currently

being examined keep two things true - the first is that none of them exceeded the error range set by the (variable and based on all the points between) height, and the second is that the lower bound has only moved up and the upper bound has only moved down, as shown by the previous paragraph. Thus, we can use the truth of the fact that last step's calculated height was within ϵ of the furthest point to say that one of two things will happen in the current step - either the bounds will change or they will not. If the bounds don't change (this would require the new point to be exactly on the line of the height calculated by the previous points, but this is certainly possible), then certainly there isn't any risk of having a point run over the height-ϵ difference - the upper bound is not greater than the point's value + ϵ, and the lower bound is not lesser than the point's value - ϵ, and we showed last paragraph that the upper bound can never be less than the lower bound. The other case is that the bounds do change, and in this case they can only change as long as - as we have seen - $y_i - \epsilon \leq upper$ AND $y_i + \epsilon \geq lower$. The previous paragraph showed that $upper$ always decreases and $lower$ always increases, and it is never the case that $upper < lower$, as the height at the start of each interval is the height of that first point, and $upper$ and $lower$ are ϵ away from the height, the height of the interval can only move closer to $upper$ and $lower$. The height is calculated as the average of the $upper$ and $lower$ bounds, and so the difference between the height and either bound is always $\leq \epsilon$. That is to say, $upper \leq height + \epsilon, lower \geq height - \epsilon$. Either of these taken in conjunction with the statement that $y_i - \epsilon \leq upper$ AND $y_i + \epsilon \geq lower$ must be true to change bounds results in the pair of inequalities $y_i - \epsilon \leq upper \leq height + \epsilon$ and $y_i + \epsilon \geq lower \geq height - \epsilon$. If y_i is greater than $height + \epsilon$ or less than $height - \epsilon$, one of these equations fails, and thus the interval would end. As such, y_i must be in between those bounds and so can't become greater than ϵ away from the height at any given step. The inductive hypothesis is thus true, and the statement that a new point's height can't be greater than ϵ away from the already-calculated height without starting a new interval is thus also true.

Fourth, and last, let's prove that the optimal solution has k intervals, the number produced by the above algorithm. Demarcate any solution to the set of points as $x_1, x2, ...xm$. If $k \leq m$, the algorithm is optimal. Greedy algorithms stay ahead - this algorithm has the property that it is using the minimum number of intervals at every point throughout the execution of the algorithm. Using induction, we can show that if the algorithm has been using the minimum number of intervals required up to a certain point, it will continue to do so in the current step. To take a page from the solution to last week's sample problem, we can state as the basis to an inductive hypothesis that the first entry in any given set of data points, no matter what, will be the first point in the first interval in any solution. We can prove here that for every interval $1...k$ in the algorithm's solution, there is at least one interval in any solution x for the data. Let point a be the one currently under examination, such that all previous points have already been placed into intervals, and such that it is the first point in a new interval, interval i. This indicates that a had a difference greater than ϵ from the height of the previous interval. The inductive hypothesis indicates that up until this point, there has been at least as many intervals in x as in my solution, meaning that x's intervals have to begin with points with x-values at most equivalent to those of my intervals. a is thus a point that has not yet fit into any of x's intervals, indicating that x must have an interval here to fit a - that matching one of my intervals. Inductively, this implies that any solution x has at least k clusters, and as such the algorithm creates the optimal solution.

c. My algorithm can more or less run in $O(n)$ - each point in the set of points is examined once, for $O(n)$, and within each examination a set of $O(1)$ operations is run: the y value of the point is checked against the error tolerance, variables are set or incremented, etc. The only thing that gives a snag here is storing the intervals themselves, but even then, any data structure that can hold at most n triples of the form $(start, finish, height)$ will do - you could use an array, although you'd be wasting a bit of space most of the time (most runs will end up with less than n intervals). Setting an array of length n up to contain NULL triples at initialization takes, of course, $O(n)$.

Problem 2 - Graph inference

a. The algorithm should look as follows: Assuming a connected graph, build a multi-rooted breadth-first-search tree where the roots are all the elements $x \in X$. Continue along each edge, marking each node with its shortest (i.e. its first, no need to re-mark nodes) distance from a root. Once the first $y \in Y$ is hit, take the length of that shortest $X - Y$ path (e.g. $X - node - node - Y$ has length 4) and set some $n = length - 1$. Continue marking the unmarked nodes in the layer until the layer is complete, then divide all markings by n. Continue along the breadth-first tree and mark all remaining nodes 1. A few edge cases: if X and $Y = \emptyset$, root at a random node and mark all nodes 0. If $X = \emptyset$, root at a random node and mark all nodes 1. If $Y = \emptyset$, mark all nodes 0.

b. The algorithm is correct because it terminates, because it gets a shortest path, because it reduces the maximum difference between vertex values (by finding the minimum distance = maximum difference), and because once it finds the maximum difference, it relegates all other vertex values to have the same or a lesser distance (in the process, guaranteeing that f is a function.)

Let's start with the easy bit. The algorithm terminates. Using a breadth-first search guarantees that each vertex is expanded once and only once - there's no going back, so to speak, and the number of vertexes and edges is definitively finite.

The algorithm gets the shortest path from a node in X to a node in Y. Again, the properties of breadth-first search all but guarantee this. The graph is unweighted, so BFS is fine to use (instead of, say, Dijkstra's or something) as it methodically expands each "layer" of vertexes n distance away from a root, one layer at a time. The Kleinberg-Tardos text makes quite clear in 3.2 that layer 1 is defined by any vertex sharing an edge with a root (in this case, $x \in X$), and that all following layers $j + 1$ are those vertexes not already defined by a layer (i.e. not already in the tree) that share an edge with a vertex in layer j. As such, the first $y \in Y$ that crops up during such an expansion will be closer to or as close to some x than any other vertex in the graph - or in effect, the shortest path between any x and y.

The algorithm's shortest $X - Y$ path is its maximum difference. This much can be derived from a sort of inductive proof: it's quite evident that if an X and a Y are adjacent and share an edge, the maximum difference will be 1 - the shortest path has only two nodes, X and Y - and 1 is the maximum difference possible in all runs of the algorithm. From this point, if we add one node between X and Y, our shortest path has length 2, and as such the difference between the X and Y can be split between the edges - the maximum difference becomes $1/2$. Inductively, we can assume that the shortest path is of length k and the maximum difference is $1/(k-1)$ - from here, we can say that if we were to introduce an extra node into the shortest $X - Y$ path, it would have length $k + 1$ and the maximum difference would be k. This is fairly intuitive - the algorithm is guaranteed to find the shortest path, and in doing so it is guaranteed to find the maximum difference.

The only hazard left in the algorithm's design is that in "counting up" from layer to layer, the algorithm may begin assigning values to nodes that unexpectedly reach $y \in Y$ too soon - e.g. the situation in which a shortest path of length 4, $n = 3$ is discovered, so that the nodes of layer 1 (adjacent with X) have value $\frac{1}{3}$, the nodes of layer 2 have value $\frac{2}{3}$, and Y is found in layer 3, but somewhere along the finishing completion of the algorithm, some node with value $\frac{1}{3}$ is adjacent to a Y, making the maximum difference $2/3$, not $1/3$. If this were the case, however, the breadth-first search property of the algorithm would have placed any Y not adjacent to an X and adjacent to a $\frac{1}{3}$ - a layer 1 node - into layer 2. Thus, the shortest path would be of length 3, $n = 2$, and to make a long story short, that's a contradiction. The first Y node reached must be the shortest and must define the maximum difference required, as we saw in the previous two paragraphs. All other paths from any X to any Y must be as long or longer, or, quite simply, they'd be the shortest path! As such, there's no way to get from X to Y while "missing" a step up, so to speak: all other nodes will be marked $\frac{n-1}{n}$ or 1.

All told, the algorithm thus takes the shortest $X - Y$ path, calculates the maximum difference based on that path's length, and applies it to the rest of the graph, only expanding each node once.

c. The algorithm runs in time $O(n + m)$. We know that this is the case for any BFS, as each node is expanded once, and each edge plotted once. The only real added cost is initializing and keeping an $O(n)$ table indexed by vertex and keeping track of marked layer assignment (or NULL, if not yet seen), meaning the maximum item here is the BFS.

Problem 3 - Shortest-path tree checking

a. The first step is to go through the input tree with breadth-first search rooted at s (with running time $O(n+m)$) and gather the tree's calculated distances. Use a table indexed by node number ($O(n)$ to initialize a blank table $dist$ of nodes using $|V|$, place their distances at $dist[num]$, and retrieve the distance of vertex u from $dist[u]$ when examining e_{uv} to calculate the distance of vertex v, placing it at $dist[v]$, with the full operation's cost at $O(1)$). Assuming an adjacency list sorted by end vertex, should any vertex $v \neq s$ show up where v is not the endpoint of any edge, it will not be reached by the BFS and the number of nodes the BFS finds will be less than $|V|$ - this should automatically return "no".

Once this table is done, all the vertexes in the tree have immediately accessible distances. At this point, the algorithm must just go through each edge and (this being a directed graph, each edge only needs to be examined once), for all edges e_{uv}, make sure that $dist[v] + length(e_{uv}) \leq dist[u]$. If this test *fails* for any edge, the tree is not the shortest-path tree for graph G and the algorithm should return "no". Additionally, if some edge containing a node not in the table established in the previous paragraph crops up, the input tree is not spanning and thus not a proper shortest-path tree - the algorithm should return "no" in this case. If all edges are checked and neither of these conditions are tripped, the algorithm should return *yes*. As this checks each edge once, this section of the algorithm runs in $O(m)$.

The algorithm runs in $O(n+m)$ - the BFS on the input tree should contain all the nodes n and some of the edges m, but in the worst case, all of both. Regardless, creating the input table definitively takes $O(n)$ and examining each edge in the second step takes $O(m)$, for a full cost of $O(n+m)$ either way.

b. Let's prove the algorithm is correct. It is correct because it terminates, because it recognizes spanning trees as spanning correctly (and conversely, recognizes non-spanning trees as non-spanning correctly), and because it properly recognizes non-shortest paths as such. For what it's worth, our input *is* guaranteed to be a tree, meaning it automatically won't contain any cycles and is connected.

The algorithm, of course, terminates - it merely does a BFS (guaranteed to end) and then steps through the (finite) set of edges in the graph, without ever returning to them.

The algorithm can adequately tell a spanning tree from a non-spanning tree. Although all inputs to the algorithm will be trees, nothing guarantees that they will hold all the vertices in the actual graph itself. The algorithm accounts for this, implicitly keeping a record of the vertices found in the input tree (guaranteed by the process of a BFS to be all the vertices in that tree) in a table. If the algorithm in its second half - actually operating on the graph proper - runs into an unrecognized node, this would automatically indicate that the tree on which the table is based is not spanning. Alternatively (and probably a better defined way of expressing this concept) the table's size could be predetermined using $|V|$, and then in the process of being filled in the BFS from $1...n$, if at the end of this fill, $n \neq |V|$, then the algorithm should immediately stop and return "no". This implementation would also handle the weird happenstance of a tree containing vertices *not* in the graph, although I'm not sure why you'd ever write code so horrible/evil.

Lastly, and actually the crux of the way this works, the algorithm recognizes whether or not a node is actually in a shortest path (for what it's worth, shortest-path trees aren't actually unique - consider the situation in which the start node has edges to two nodes, with paths of equal length, each of which also has an edge to another node, again with equal length - the tree could be formed going in "either direction" so to speak). This is essentially done by taking the possible distances calculated by the BFS on the input tree and measuring these against the graph itself. As a tree is defined as a graph where any two vertices are connected by exactly one path, there is only one way in the input tree to get from s to a random vertex v - the distance value is thus dependent on each segment between the start vertex to v, and the input tree makes the claim that this distance value is the minimum possible. A successful shortest-path tree building algorithm has the property, then, that some vertex v has a set of nodes u such that $e_{uv} \in E$ - and that that successful algorithm chooses the shortest path, so that $dist[v] = dist[u] + e_{uv}$ is minimized. It thus suffices to say that in any particular node on a graph, if an assumed distance for v is known, the minimal $dist[u] + e_{uv}$ is the proper edge to include on a shortest-path tree. This assumed distance for v is thus *correct* if its sole predecessor t in the tree has the minimum $dist[t] + e_{tv}$. This can be exhaustively proved by examining all edges with v as their endpoint and showing that t, v's predecessor in the tree, is better suited as a predecessor than all potential predecessors u - essentially, that $dist[u] \geq dist[v] + e_{uv}$, implicitly stating that $dist[t] = dist[v] + e_{tv}$. Should this condition ever be false, that u would create a smaller distance for v (and all nodes to which v would be predecessor) - in short, there would be a better tree.