The algorithm is to split into halves, find a majority element of the first sample and one of the second (if they exist). Then you need to check the up to 2 candidate majority genes against all n samples to see if any is 50% + 1. Then return that element or indicate that there is no majority element.

The run time is $T(n) = 2T(n/2) + cn$ for some constant c, and by the Master Theorem (or by tracing the recursion tree or by noticing this is the same recursion as mergesort) the runtime is $\Theta(n \log n)$.

The proof is by induction. The key part is to show that a majority element of the whole set must be a majority in one of the two halves.
Assume the algorithm correctly returns a sample containing the majority gene or indicates that there is no majority gene on a set of up to n genes. Consider a set of n + 1 genes. Split it in half, and by induction the algorithm correctly returns a sample containing the majority gene of each half. Since we are comparing the sample returned against all the n samples, clearly if any of the two have a majority gene, we are done. Suppose there is a majority gene that is not in one of the (up to) two samples returned. Since that sample is not majority in the first half, it occurs in at most 1/2*(n+1)/2 of the first half samples. Since that sample is not majority in the second half, it occurs in at most 1/2*(n+1)/2 of the second half samples. So, that gene will occur in at most 1/2*[(n+1)/2+(n+1)/2] samples and n+1 is not a majority gene.

Quiz 8:
1) A simple reduction to Dijkstra.
For each edge, if the edge is compromised, give it weight 1 and if not compromised give it weight 0. Use Dijkstra to find the shortest path from s to t. The run time of Dijkstra is $O(m \log n)$ and the reduction is $O(m)$ to give $O(m \log n)$.

The path uses K compromised edges if and only if the cost of the shortest s - t path in the new graph is K. Assume the path has K compromised edges, then it used K edges of weight 1 and the rest of weight 0. Assume the shortest path is length K, then it used K edges of weight 1, and these are the K compromised edges.

(Technically this solution also requires them to show that Dijkstra works with 0 cost edges. In class we claimed the edge costs had to be positive, but you can ignore that issue.)

2) The same idea but more rigorous.
Let W = m.
Any edge that is compromised gets weight W while an uncompromised edge gets weight 1. Run Dijkstra. Since summing the weights is O(log m) which is the same as the cost of maintaining the sorted costs, so the run time is still O(m log n). The path from s-t uses K compromised edges if and only if the length of the s-t path is greater than KW and less than (K+1)W. The rest of the proof is identical to above.

3) Or they do a greedy algorithm/proof based on Dijkstra.
Let L[v] = infinity where L[v] will store the number of compromised edges on the best path found so far. Let P[v] = null where P[v] stores the previous vertex to v on the path from s. Set L[s] = 0 and T = empty.
Until t is added to T, get the vertex v not in T with smallest L value. That is the least compromised path to v. For each vertex a adjacent to v, if edge (a,v) is not compromised and L[a] > L[v] then set L[a] = L[v] and P[a] = v. If edge (a,v) is compromised and L[a] > L[v] + 1 then set L[a] = L[v] + 1 and P[a] = v.

Now, the proof is almost identical to the one from last week. Here is how the swap version should start:

Assume we have an optimal solution T* that gives the least compromised path from s to all vertices. Suppose our algorithm matches T* on the path to the first k vertices added to T. Now we find a different path that T* to vertex k+1.

Quiz 7:
Let V = emptyset, and for each vertex v of G, let L[v] = infinity for v != h and L[h] = t_0. Let P[v] = null for all v.

Repeat :
  Let v be the vertex of V(G) - V that has the smallest L value.
  Add v to V.
  For each edge (v,a) of G, set L[a] = min{L[a], L[v] + f_{v,a}(L[v])}, and if this changed L[a], we set P[a] = v.
Until (v = w) or until all vertices chosen.

We can get the path by following P[w], P[P[w]], P[P[P[w]]], ..., h.

The running time of the algorithm is identical to Dijkstra's. Every edge is considered at most twice, and we have to maintain a sorted list of L. We can use a heap for that and so $O(m \log n)$.

Proof 1: (Greedy stays ahead)
We prove that each time we add a vertex v to V, then L[v] is the earliest/least cost we can reach v from h when starting at time t_0.

Let O[v] be the true earliest we can reach v and order the vertices so that O[v_1] <= O[v_2] <= ....

Consider the first vertex in this ordering where O[i] < T[i]. The optimal path to i reaches i by travelling an edge (j,i). Since O[j] < O[i], we know by our assumption that T[j] = O[j]. However, that means that j was added to V before i, and when we added j, we set L[i] <= L[j] + f_{j,i}(L[j]) = O[i]. This contradicts the assumption that O[i] < L[i].

Proof 2: (Swap with optimal)
Suppose that the tree of least cost paths from h to all other vertices matches an optimal tree for the first k vertices, in the order our algorithm added them, but then the optimal algorithm uses a different path to vertex k+1. Let x be the vertex immediately preceding vertex (k+1) on the optimal path and y be the vertex immediately preceding (k+1) on the greedy path. Either x < k+1 or x > k+1 (ordered by how they were added to V by greedy). If x < k+1, then by the induction hypothesis, L[x] = the optimal time we can reach O, and then greedy considered L[x] + f(x,k+1) and L[y] + f(y, k+1) and found that L[y] + f(y, k+1) <= L[x] + f(x, k+1). Thus we can change O to have y precede k+1 instead of x, and the time to k+1 can only decrease, and thus the time to any vertex after k+1 also can only decrease. Suppose x > k+1. Then there are two vertices x' and x'' with x' < k+1 and x'' > k+1 where the edge (x',x'') is on the optimal path to k+1. By the same argument as above, we know L[y]+f(y,k+1) <= L[x'] + f(x',x''). By the induction hypothesis, L[x'] is the optimal time to x', and so changing O so the optimal path to k+1 goes from y can only decrease the time to k+1, and thus also only decrease the time to any vertex reached from k+1.