Solutions for Exam problems

1. The first part was straightforward. Find the value that occurs the maximum number of times in the array. If this occurs k times and $k > n/2$, the whole sequence has a majority, otherwise the longest subsequence with majority has length $2k-1$. Since the numbers were given to be at most n, this can be done in $O(n)$ time, but $O(n \log n)$ would also have been okay.

For the second part, for each i, find the longest substring with i as the majority element, in time proportional to the actual number of occurrences of i. Suppose $i_0 < i_1 < \cdots < i_k$ are the indices at which i occurs. These lists can be constructed in $O(n)$ time total for all i. Consider the substring from the pth to the qth occurrence of i. This has length $i_q - i_p + 1$ and the number of occurrences of i is $q-p+1$. For this to be majority, $2(q-p+1) > i_q - i_p + 1$ or $i_p - 2p \geq i_q - 2q$. Consider the numbers $i_j - 2j$ for $0 \leq j \leq k$. For this element at index q, we need to find the smallest index $p < q$ such that $i_p - 2p \geq i_q - 2q$ to get the longest majority string ending at q. This can be done in $O(k)$ time by sorting these values and finding the smallest index to the right, for each number. Since these numbers are also 'small', sorting can be done in $O(k)$ time, though using the standard sort function makes little difference. This gives the maximum number l of occurrences of i in a substring with i as majority, and the actual substring has length $\min(n, 2l-1)$. Taking maximum for all i gives the answer.

2. Computing the edges was straightforward, but quite a few have not handled the $n = 1$ case properly. Very few have got the correct algorithm for the second part. The basic idea is to use recursion, but for each cograph corresponding to a subtree, we need more than just the longest path. This is the maximum number of edges that can be contained in disjoint paths in the graph. In the case of cographs, this number is obtained by taking any longest path, deleting it, taking a longest path in the remaining graph, and so on. This is not true for all graphs, for example, the graph that looks like an H . So each node of the tree will maintain the lengths of these paths, and we use a priority queue to access the elements in sorted order. It is enough to keep only the non-trivial paths, since others can be found from the number of vertices.

For a `0` node, we have to simply merge the two queues corresponding to the two children. This needs to be done carefully as explained later. Consider a `1` node. If there are equal number of nodes in the left and right subtree, the graph contains $K_{n/2,n/2}$ as a spanning subgraph, and there exists a Hamilton path in the graph, irrespective of what the left and right subtrees are. In this case, the queue will contain a single number, the number of nodes - 1. The same holds if the number of nodes in the two subtrees differ by at most 1. Suppose number of nodes in the right subtree is at least 2 more than the number in the left. Then try to use as many of the longest paths in the right subtree as possible and otherwise alternate between nodes in left and right subtree. If left subtree has n_l nodes, at most $n_l + 1$ paths from the right subtree can be used, and we use the longest ones. If the number of edges in them is at least the difference in number of nodes - 1, we again get a Hamilton path. Otherwise combine these paths with the n_l vertices on the left to get the longest path, and leave the other paths as they are. This can be done by popping elements from the priority queue of the right tree, till it becomes empty or one of the conditions is satisfied. The length of the longest path is inserted in the queue.

An implementation detail, that a few people who got the correct algorithm missed, is that merging and copying queues can take time, and would end up taking $O(n^2)$ time. Instead, maintain a pointer to a priority queue at each node, and when combining two queues, copy the elements from the smaller queue to the larger, and set the pointer of the parent to point to the larger queue. Every time an element is copied, the size of the graph containing that path at least doubles, so this can happen at most $O(\log n)$ times. Taking into account the time for the queue operations, the worst case can be $O(n \log^2 n)$. There are implementations of priority queue that allow merging in $O(1)$ time on the average, which would reduce this to $O(n \log n)$, but this was not needed. The worst case is when the tree has large height with lots of `0` nodes.

Only one person seems to have done a correct implementation of this.