COMP2123 Final Exam

Problem 1.

a) The time complexity for line 2 is O(1) and the return step is also O(1). The priority queue takes O(nlogn). The steps inside the while loop takes O(1) each, but the worst case is that all the keys inside the priority queue are smaller than x^2, which means that it would take O(n) time. Therefore, the time complexity of running FOO is O(nlogn).

b) Let $S_1=\{i_1, i_2, i_3\}$, $S_2=\{i_1, i_2, i_3\}$, $S_3=\{i_1, i_2, i_3\}$, $S_4=\{i_1, i_4\}$, $S_5=\{i_2, i_4\}$. According to this algorithm, i_1 and i_2 appear 4 times which is the most frequent, so we add i_1 to T first. Then i_2 appeared in S_5 and no item in T is an element of S_5, so we add i_2 to T.

Problem 2.

a) T = {AB, BF, FE, BC, FG, CD, GH}
b) AB, BF, FE, BC, FG, CD, GH

Problem 3.

a) To solve this problem we use a 2D-array adjacency matrix to store all the vertices. As shown in the lecture, the 2D-array adjacency matrix has vertices as the rows and columns.
To initialize the data structure, we have to create the matrix by recursing through the rows and columns. Create n rows and n columns, while each part of the matrix is empty.
For insert-edge(i, j), if inserting this edge creates a cycle, then don't do anything (return None). If inserting this edge does not create a cycle, recurse through the rows for the vertex i, then recurse through the columns for vertex j. Add the edge to this place. Similarly, recurse through the columns for i, and then recurse through the rows for j. Also add the edge to this place. Then perform a DFS and add all the vertices traversed to a new list {}.
For remove-edge(i, j), recurse through the rows for the vertex i, then recurse through the columns for vertex j. If there is an edge here, then remove the edge. Otherwise, don't do anything. Similarly, recurse through the rows for the vertex i, then recurse through the columns for vertex j, if there is an edge here, then remove the edge, otherwise don't do anything.
In-same-tree(i, j) would just look at the list that has i in it. If the list also has j in it, then return true, else return false.

b) By creating a n x n matrix, we successfully constructed a data structure for n vertices without any edges. Then inserting edges we recurse through all the rows and columns to find the exact position we want to insert the edge. The edge would be added at (i, j) and also (j, i) if no cycles are produced. Then by performing DFS, we can get all the vertices of the same tree, which is used in in-same-tree function. Removing edges would also recurse through all the rows and columns to find the exact position. Therefore the algorithm is correct.

c) The time for initializing is O(n^2), as it needs to create n rows and n columns, with O(1) each position. Inserting and removing edges would need O(n^2) to recurse through n rows and n columns. For insert, we also performed a DFS, which needs O(n) time, but since the total time is n^2+n , it is still O(n^2). For in-same-tree, it has to find the list that has i in it, then recurse through the list to find j. This needs O(n) time.

Problem 4.

a) First we need to sort the locations of the shepherds in non-decreasing order. Then divide the list A into two halves. Then work on the two halves separately.

For the left part, start with the first location and make it the current location. Recurse to the right to the next location. If this location minus the current location is smaller or equal to d, then increment ***count*** by 1 and then move on to the next location. Continue until the distance between the new location and current location is greater than d. Then move the current location to the second location and do the same thing until the current location becomes the last index.

Similarly, for the right part do the same thing and loop through all the locations.

Now go to the last element of the left list. This location is **a**. Then find all the numbers that are less than or equal to **(a + d)** in the right list. Whenever there is a location less than **(a + d)** ***count*** is incremented by 1. Recurse to the left of **a** until **(a – d)** and do the same thing.

Finally, return ***count***.

b) First sorting the locations in non-decreasing order helps us compare the distances easier. By recursing through all the locations, we compare the distance between the locations. This would make sure that no combinations would be missed. If the distance is smaller or equal to d, then we increment count by 1 to record the number of pairs that can communicate with each other. Then we have to find all the combinations that cross through the left part and right part. Therefore the algorithm correctly finds all the combinations and return the number of counts.

c) Sorting the list takes O(nlogn) time. Then finding the centre position of the list to divide takes O(n) time. Looping through the two separate lists takes 2T(n/2) time. Finally conquering takes O(n) time, as the worst case is all shepherds can communicate with each other.

By using master's theorem, a=2, b=2. $\log_2 2=1$. $n^1=n logn^0$ which is case 2 of master's theorem. Therefore the time is O(nlogn) for divide and conquer.

Adding the time of sorting and conquer, the total time is 2nlogn which is still O(nlogn).