Question 1. Minimum Spanning Trees [12 MARKS]

Let $G = (V, E, w)$ be a undirected, connected, weighted graph and assume all weights are distinct (i.e., $w(e) \neq w(f)$ for any pair of edges $e, f \in E$ with $e \neq f$). Suppose C_0 is a given simple cycle in G, and $e_0 = (u_0, v_0)$ is an edge on C_0.

Given the above assumptions, for each of the questions below circle the most specific correct answer (i.e., if something "always" happens or "never" happens, then "sometimes" will be marked wrong).

Part (a) [2 MARKS]

Suppose e_0 and C_0 are as above and e_0 is the maximum weight edge on C_0, then e_0 is
Soln: never
in an MST of G.

Part (b) [2 MARKS]

Suppose e_0 and C_0 are as above and the weight $w(e_0)$ is neither the minimum nor the maximum of the weights of the edges on C_0, then e_0 is
Soln: sometimes
in an MST of G.

Part (c) [2 MARKS]

Suppose e_0 and C_0 are as above and e_0 is the minimum weight edge on C_0, then e_0 is
Soln: sometimes
in an MST of G.

Part (d) [2 MARKS]

Suppose $e_1 \in E$ is the maximum weight edge over **all of *E***. Then this e_1 (which may be different than the e_0 above) is
Soln: Sometimes.
in an MST of G.

Part (e) [2 MARKS]

Suppose e_0 and C_0 are as above then it is
Soln: Always
true that $|E| \geq |V|$.

Part (f) [2 MARKS]

Consider an edge $a = (a_1, a_2) \in E$ such that there is no path $P = (p_1, p_2, \ldots, p_n)$ in G such that $p_1 = a_1$, $p_n = a_2$, and all the edges on P satisfy $w((p_i, p_{i+1})) < w(a)$ for $1 \leq i \leq n-1$. Then a is
Soln: Always
in an MST of G.

Question 1. Minimum Spanning Trees [12 MARKS]

Let $G = (V, E, w)$ be a undirected, connected, weighted graph and assume all weights are distinct (i.e., $w(e) \neq w(f)$ for any pair of edges $e, f \in E$ with $e \neq f$). Suppose $S_0 \subset V$ and $D_0 = \text{cutset}(S_0)$. Suppose $e_0 \in E$ is an edge in D_0.

Given the above assumptions, for each of the questions below circle the most specific correct answer (i.e., if something "always" happens or "never" happens, then "sometimes" will be marked wrong).

Part (a) [2 MARKS]

Suppose e_0 and D_0 are as above, with $|D_0| > 1$, and e_0 is the minimum weight edge in the cutset D_0, then e_0 is

Soln: always

in an MST of G.

Part (b) [2 MARKS]

Suppose e_0 and D_0 are as above, with $|D_0| > 2$, and the weight $w(e_0)$ is neither the mininimum nor the maximum of the weights of the edges in the cutset D_0, then e_0 is

Soln: sometimes

in an MST of G.

Part (c) [2 MARKS]

Suppose e_0 and D_0 are as above, with $|D_0| > 1$, and e_0 is the maximum weight edge in the cutset D_0, then e_0 is

Soln: sometimes

in an MST of G.

Part (d) [2 MARKS]

Suppose e_0 and D_0 are as above, with $|D_0| = 1$, then e_0 is

Soln: always

in an MST of G.

Part (e) [2 MARKS]

Suppose e_0 and D_0 are as above, with $|D_0| > 1$, then G

Soln: sometimes

has a simple cycle C with two or more edges in D_0.

Part (f) [2 MARKS]

Define $D(v)$ to be the cutset formed from $S(v)$, where $S(v) = \{v\}$ for each $v \in V$, and define

$$B = \{e \mid e \text{ is the minimum weight edge in } D(v) \text{ for some } v \in V\}$$

then the graph (V, B) is

Soln: always

acyclic.

Question 2. Divide and Conquer: Count Duplicates [18 MARKS]

Given an unsorted list of integers, L, we want to return both a sublist D which contains every integer in L but without any duplicates, and a list N of the same length as D for which $N(k)$ equals the number of times the element $D(k)$ appears in L. For example, given $L = (5, 3, 4, 3, 4, 3, 3)$, one possible output would be $D = (3, 5, 4)$ and $N = (4, 1, 2)$ (i.e., D can be given in any order, but the entries in N must correspond to D's).

Part (a) [15 MARKS]

Finish the pseudo-code function `countDup` below for computing D and N. To get any marks at all your algorithm must make essential use of the output from the recursive calls of `countDup`, and must have a runtime of $O(|L| \log |L|)$. Be precise, some marks will be taken off for incorrect details such as off-by-one errors. Include comments to assist the marker.

```
// For a list L of length n, the solution D and N is given by
[D, N] = countDup(L, 1, n)

[D, N] = countDup(L, a, b)
// Input: Array L(1..n) of integers, and indicies 1 ≤ a ≤ b ≤ n.
   // Output: The lists D, N described above for the sublist L(k), k = a, ..., b.
   // Include any other important properties of the output lists D and N.
   //
   //
   //
   if a == b
      D = L(a); N = (1)
      return D, N
   else
      m = floor((b + a - 1) / 2)
      [D1, N1] = countDup(L, a, m)
      [D2, N2] = countDup(L, m+1, b)
```

Sketch of Soln: Add the following comment: The output list D is in sorted (increasing) order upon return and $D(k)$ occurs $N(k)$ times in the sublist from a to b.

The pseudo-code here is similar to that of merge sort. Since both sublists D1 and D2 are sorted, we can perform a standard merge to compute the new D. The only difference from merge sort is when there is a common item in the two half lists. That is, when there are k1 and k2 such that $D1(k1) == D2(k2)$. In this case, only one of these items is appended to D, and the number N of these items is set to $N1(k1)+N2(k2)$. We omit the details, although these details were marked on your exam.

Part (b) [3 MARKS]

Analyze your algorithm's running time.

Initially, the list length is n. **Soln:** Note that the two recursive calls have lists of length $floor(n/2)$ and $ceil(n/2)$. (The details here can be skipped.) Moreover, the merge step requires $O(1)$ computation for each element added to D, and at most n elements are added to D. Therefore, the runtime of the rest of this function is $O(n)$. Therefore we have

$$T(n) \;=\; 2T(n/2) + O(n).$$

The Master Thm applies with a=2, b=2, and d = 1, so this algorithm runs in time $O(n \log(n))$.

Question 2. Divide and Conquer: More Than a Third [18 MARKS]

We are given a list of $n > 0$ objects, say $X = (x_1, x_2, \ldots, x_n)$. The only operation we can do on these objects is equality testing (therefore we can not sort it). We must find all objects which occur in X strictly more than $n/3$ times. (We define $|X|$ to be this n.)

Part (a) [3 MARKS]

Define M to be the maximum number of different objects (e.g., x_i and x_j with $x_i \neq x_j$) that can each appear in X **strictly more** often than $|X|/3$ times. (Note the minimum number of such frequent objects is zero.) Give the value of M and prove (carefully) that it is correct.

Soln: Suppose M objects appear strictly more than $|X|/3$ times. Then just these objects must together appear strictly more that $M|X|/3$ times. But the list is length $|X|$ so we must have $|X| > M|X|/3$, so $M < 3$. That is, the maximum number is $M = 2$.

Part (b) [12 MARKS]

Finish the pseudo-code of the function `freqThird` below for computing a list, D, of items which occur strictly more than $|X|/3$ times in X. Your algorithm:

- Must (to get any marks) make essential use of the two recursive calls in the code below;
- Can return additional values beside D (but see the next point);
- All returned values must be either constants or lists that all have $O(1)$ length (such as, D, which we showed in (a) is a list of length at most $|M|$);
- Should have a runtime that is as small as possible (part marks are offered here).
- Should include enough comments to assist the marker.

```
// Return the list D of items appearing in X strictly more often than |X|/3 times:
D = freqThird(X, 1, n)

[D, _____] = freqThird(X, a, b)
   // Input: Array X(1..n) of objects, and indicies 1 ≤ a ≤ b ≤ n.
   // Output: The list D described above for the sublist X(k), k = a, ..., b.
   // Describe any additional output here:
   //
   if b > a
      m = floor((b + a - 1) / 2)
      [DL, _____ ] = freqThird(X, a, m)
      [DR, _____ ] = freqThird(X, m+1, b)
      // Note to student: Remember to complete the else clause below.
```

Soln: You actually do not need to return anything else (but it might be simpler to recognize the solution this way).

The key idea is that you can afford to simply count the number of items equal to each item in DL or DR in the sub-list from a to b. This will take $O(b - a + 1)$ for at most 4 items.

For efficiency, you do return counts CL and CR in the two recursive calls above. You then do not need to recount items that are in both lists DL and DR (the number of items in the combined list is just $CL(k1) + CR(k2)$ where $DL(k1) == DR(k2)$). Otherwise, if an item only appears in DL, then you need to count how often it appears in the right sublist (from $m + 1$ to b). And similarly for items that only appear in DR. We omit the details. These counts take at most 4 times $ceil(n/2)$ time, i.e., $O(n)$.

The base case is for $b == a$, and to return $D = (X(a))$, and $C = (1)$.

It is ok in the body of this function to use a mapping data structure for the sub-lists DL and D (like a dictionary in python), or to use a set data structure. But it is not ok to use a mapping data structure (or

hash table) for the whole list of items from a to b (or from 1 to n). The reason is that this does not make essential use of the two recursive calls since we could simply solve this problem with hashing in $O(n)$ time.

Part (c) [3 MARKS]

Analyze your algorithm's running time.

Soln: As explained above, there are two recursive calls on sub-lists of length $floor(n/2)$ and $ceil(n/2)$, and the necessary counting takes at most $O(n)$ time.

Therefore, in the Master Thm, $a = b = 2$ and $d = 1$, so the runtime is $O(n \log(n))$.