Solution 3.

(a) False.

Proof by contradiction. Assume this is possible. Then you can make use of the following steps to achieve an O(n) routine for sorting in comparison-based model

i). Convert an array into a max-heap
ii). Convert the max-heap to a BST
iii). Run inorder traversal on the BST

Each of these steps take O(n) time. But we already proved that it is not possible to achieve this. Hence a contradiction.

(b) False

In a binary tree, the minimum (and second) element can be anywhere.

[if it was BST, the answer was "**True**"

Since it is a nearly complete binary tree, the minimum element (*keep going left if you can*) can not have any child. Also, this node will be at the maximum level l. Hence, the second minimum (successor of this element) will be an ancestor. It is not difficult to see that it will be its parent (leftmost element at level l-1), and you have an O(1) access to this.]

(c) You have to make use of select / order-statistics (A,k)

```
t=n/5;
BreakGroups (A)
        If A. size <= t
                Print A; return; // Base case
        x = Select (A,t+1); // Find the (t+1)th order statistics in the array
        Partition(A,x);//Partition around x
        Print A[1...t];//The left part will have the set of smallest t elements; print this set
        BreakGroups (A[t+1; A.size]);//Recursively call this
        return;
```

Since BreakGroups is called only finite times, it is ok if a simple pseudocode is provided that does not use any loop.

(d) Make use of Counting sort idea with some change

C[ ]: Array storing the count of elements for range [1,k] / [0,k-1]

Get the cumulative sum

For j=2 -> n

C[j] + = C[j-1]

Preprocessing takes O(n+k) = O(max(n,k))
Now this array C[ ] can be used to answer the query:

To find the students with marks in the range [x,y], output C[y] - C[x-1] {Take care of the case when x=0. But no marks need to be deducted if not done}

This is an O(1) solution -- [3 marks]

If the solution depends on the actual range [x,y], it will not be O(1), 1.5 marks may be given.

Solution 4.

Yes, this is possible. See the algorithm below.
X_L<X_R can be replaced by string comparison
Probe == looking at / only using values of

// 3 probes //

FIND-LOCAL-MIN(T) // *T is the root of the Tree* //

- If T has children
  - Let L and R be T's children
  - Probe the values of x_L, x_R, and x_T
  - If $x_L < x_T$
    - Return **FIND-LOCAL-MIN**(L) // *find local minimum on the left subtree* //
  - Else If $x_R < x_T$
    - Return **FIND-LOCAL-MIN**(R) // *find local minimum on the right subtree* //
  - Else
    - Return T
- Else // *T is a leaf (no children)* //
  - Return T

The proof below shows the correctness. This is not required in the solution. Also, the analysis on the number of probes need not be explicitly mentioned. Give full marks if the algorithm is correct and O(logn) is mentioned.

Claim: At any point in the execution of the algorithm, the parent (if any) of **T** has a greater value than **T** itself.
Proof: Consider first the case in which **T** is the root of the entire tree. In this case **T** has no parent and the claim holds vacuously. Otherwise consider the execution of the method for the parent of **T**. The only way for the algorithm to continue recursively down to **T** required one of the highlighted conditions to hold true. If **T** was a left child of its parent then the first condition had to

hold true which states that the value of **T** is less than that of the parent. Similarly for the case where **T** is a right child.
Given this fact consider now the conditions in which **T** is returned by the algorithm. The first is when both its children have a greater value than itself. **T** is certainly a local minimum here as the parent was shown to have a greater value as well. The other way to return **T** is when it is a leaf (no children). In this case the only node **T** could be connected to is its parent and again we know that the parent's value is greater than that of **T** and hence **T** is a local minimum.

Number of probes:
Remember that the number of nodes in a complete binary tree is $n=2^d-1$ and thus its depth is d which can be represented as $\lg(n+1)$ by solving for d (note lg is same as $\log_2$).

Now notice that every time a recursive call is made in the algorithm, it is called with a node at a level 1 lower than the currently considered node. Thus the algorithm will recursively traverse a path down the tree which will take at most d steps. At each point three probes are performed hence the total number of probes will be $O(3*d) = O(3*\lg(n+1)) = O(\lg(n))$.

Question 1

For each $m \leq k$ and each v in V, we let $N_m(v)$ be the number of $s - v$ paths with exactly m edges. We note that

$$N_0(v) = \begin{cases} 1 & \text{if } v = s \\ 0 & \text{otherwise} \end{cases}$$

and that for $m > 0$, $N_m(v)$ is given by the recurrence

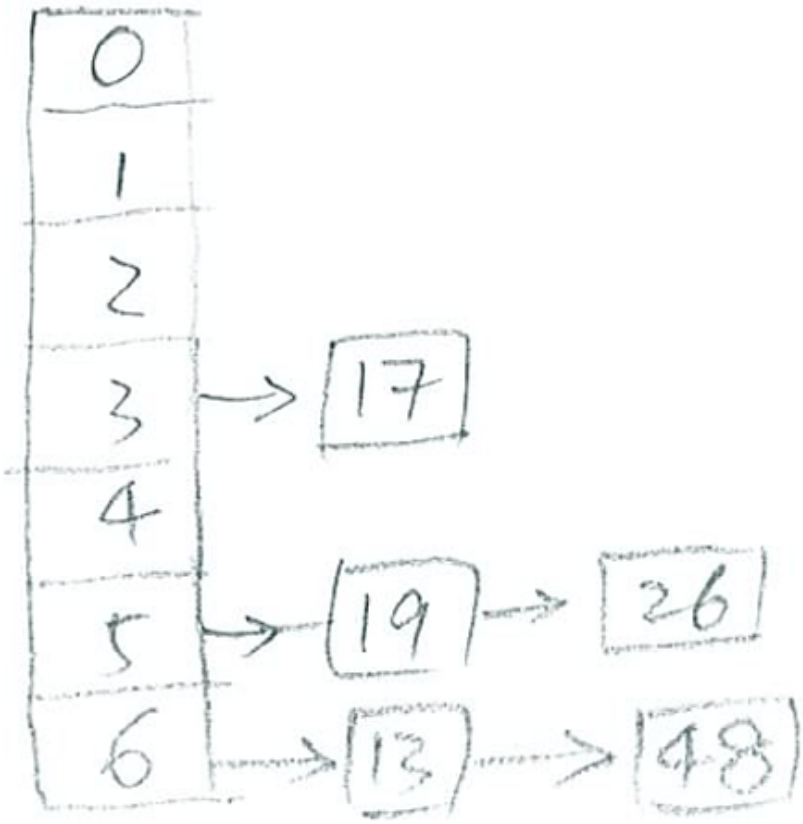
$$N_m(v) = \sum_{(u,v) \in E} N_{m-1}(u).$$

This is because a path of length m ending at v is given by a path of length $m-1$ ending at some u with an edge to v followed by the edge (u, v). Thus, we have the following dynamic program:

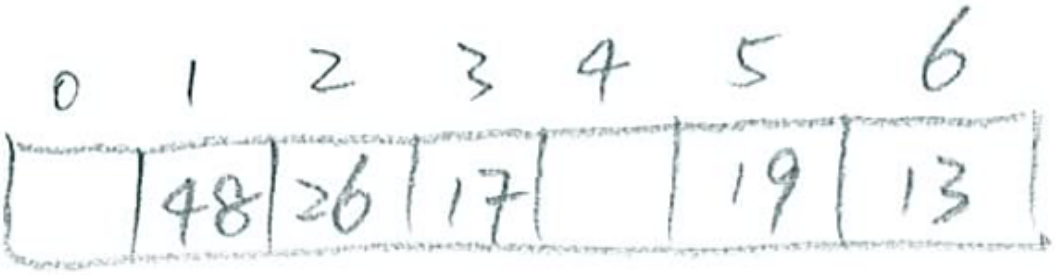
```
Initialize an array N indexed by a number m at most k and a vertex of G
For v in V:
  N[0,v] <- 0
N[0,s] <- 1
For m = 1..k
  For v in V
    tot <- 0
    For (u,v) in E
      tot <- tot + N[m-1,u]
    N[m,v] <- tot
return N[t,k]
```

To analyze the runtime, not that for each m from 1 to k we need to analyze each vertex v, and need to sum over each edge going into v. Therefore, the runtime is $O(k(|V| + |E|))$.

Question 2


0
1
2
3
17
4
5
19
26
6
13
48


0 1 2 3 4 5 6
13 48 17 19 26


0 1 2 3 4 5 6
48 26 17 19 13