DSA Homework

Katie Foster

Part 1

a)

In order to create an algorithm that runs in O(n) time, you would need to preprocess the data. The best way to preprocess the data is to create a dictionary that contains each student's name as a key, and a list containing all the classes that the student is in as a value. The runtime to preprocess the data would be *O(n)* where *n* is number of students * max classes that student takes (which is at most 5).

The algorithm itself could remain very similar.

1. Create an empty dictionary where each student is the key and the value is a boolean of whether they are on list of infected students *(O(1))*
2. Create an empty list (called i_list) that will contain all infected students *(O(1))*
3. Add patient 0 to list i_list and the dictionary *(O(1))*
4. Get all of the student's classes *(O(1))*
5. Iterate through each class *(O(5))*
6. For each class, look up all other students who have that class in their class list *(O(1))*
7. Iterate over each student in the class *(O(n))*
    a. Check if student has been added to list (tracked by dict) *(O(1))*
    b. If not, add student and set student's dictionary value to true *(O(1))*
8. Recurse until there are no more students to check *(O(n))*

The runtime of the algorithm would come out to O(n). The preprocessing takes O(n), and so does the loop that iterates through all of a student's classmates. Those O(n) runtimes, in addition to all the O(1) runtimes, simplify to O(n).

b)

Suppose a student got infected but was not added to the list.

The algorithm works by starting with patient zero and iterating over all students in patient zero's class, and setting them to infected. Then it iterates over all those students, checks what classes they are in, and sets all the students in those classes to infected.

In order for a student to become infected but not added to the list, they would have to

become infected without sharing a class with an infected person. Being in the same class as an infected person is the only way that students can become infected with academitis.

Contradiction. This completes the proof.

Part 2

Inductive Hypothesis:
Merge sort maintains a sorted list

Base Case:
List of length 1, which is always sorted

Inductive Step:
Suppose the list is sorted when the list has n elements elements (n>=1 and n<=k). We want to show that the inductive hypothesis holds for k+1 elements.

Because each list is sorted in the base case, and the lists maintain being sorted, we know that the lists are in correct order as they are merged.

We start by splitting our list into sublists recursively. By the inductive hypothesis, the two sublists are sorted. I will prove that the sorted pattern continues to hold as we merge the two sublists.

To merge the lists, the algorithm iterates through each list, comparing the current element of the first list to the current element of the other, and then adds the smaller element to a new list. Since we are always adding the smallest element to the new list, the new list will be sorted, which shows that the inductive hypothesis holds for a list of k+1 elements.

Part 3

a)

- Make set() would be implemented in O(1) using a doubly linked list of node classes. The nodes would store:
    - Their own value
    - A pointer to the next node
    - A pointer to the previous node
    - The head of its own set

- Union(A, B) would be implemented by finding the shorter list out of A and B, which would take O(1), since each list would store its own length. Then the program will iterate through each list element, and change the head of its own set to the head of the other set. This step would take O(n) where n is the number of elements in the smaller list between A and B. The last step is to actually join the two lists, which would be done by making the tail of list A point to the head of list B, and making the head of list B point to the tail of list A (or vice versa). This step would take O(1), which would make the overall time O(n) for making a union.
- Find(i) would be very easy to implement in this case. Since each node stores the head of its own set, we could access the necessary information through the node in O(1) time.

b)

- Make set() would be implemented in O(1) by initializing a tree containing nodes that store:
    - Their own value
    - A pointer to their parent node
    - A list of their children
- Union(A, B) could be implemented in O(1) time by making the top of tree B into a child of the top of tree A (or vice versa). Since this tree does not have to be a binary, there is no further moving of elements required.
- In order to implement find(i) in O(logn), you would set i to the current node, and go up from the current node to the parent node, setting the current to the parent over and over, until you have reached the top. You will know that you have reached the top when the current node has no parent. Once you have reached the top, you return the top node, which is equivalent to the head.