Homework #6

Megan Ku
Data Structures and Algorithms

March 6, 2020

1. Olin-avirus

Here is my solution taken from Homework 4 for reference:

> *We can cure the Olin population in a similar way to how we found whether a path existed in the maze problem. We can use a queue to keep track of who we need to search, and have a list to store all the potentially sick students. We start by enqueueing patient zero. Now, while the queue is not empty, we can dequeue one student from the queue and add that student to our final list. We then enqueue any classmates of the current student that aren't already in the final list. This process repeats until the queue is empty, which signifies that every potential student has been added to the final list. The final list can be represented as a python list, stack, DLL, queue, or any other structure that is mutable and can hold multiple values as order doesn't matter and we are only adding values to these structures.*
>
> *By using a queue to keep track of students not yet checked for their classmates, we are making sure that no students are checked more than once. We are also making sure that all potential students are checked by enqueueing them all.*

(a) Runtime Analysis

Let's first do a runtime analysis of the solution that I gave in Homework 4. The longest that the queue could be is n, or all of the students at Olin. The outer while loop ("While the queue is not empty...") will run at most n times. Within that loop, any students that aren't already in the final list are added if they share a class with the student most recently dequeued. Since we are assuming that there are 25 students in a class and every student takes at most 5 classes, at most 120 students will be added to the list in the loop (24 classmates * 5 classes). But checking if a student exists in the final list will take at most O(n) time per sweep. Therefore, the runtime of this algorithm is, at worst, $O(120\ n^2) \implies O(n^2)$.
Given a hash map of the classes at Olin and the students in each of those classes, and a list of all the students at Olin, we can create two new hash maps. The first hash map has the students as the keys and the students' classmates as the values. The second hash map keeps track of whether students have been already considered or not; the key is the student, and the value is a Boolean that symbolizes whether the student has been checked or not. For our purposes, let's have all values initialize as False, so students already visited will return

True. This will reduce the runtime of the algorithm by a factor of n, as we don't have to scan a list to see if a student exists in it; hash map lookup has a runtime of O(1).
The new algorithm starts by adding patient zero to the queue. Then, patient zero is dequeued, added to the final list of potentially infected students, and all of patient zero's classmates are enqueued. We then set patient zero's value in the boolean hash map to True. We dequeue students, one at a time, and check in the boolean hash map to see if they have already been checked. If so, we add them to the final list of potentially infected students and enqueue their classmates. The structure of this algorithm is extremely similar to what was designed in Homework 4, but the runtime reduction discussed earlier drops the total runtime down from $O(n^2)$ to O(n).

(b) Potential Patient Proof

We will prove the correctness of the above algorithm by contradiction.

Suppose that a student caught academitis but was not added to the list. Since the algorithm adds each of a student's classmates to the list, if a student wasn't added, that means that they didn't interact with any student that could have caught academitis, which means that they didn't catch academitis from another student. This means that the student didn't catch academitis. Contradiction! [QED]

2. Merge Sort Proof

For this proof, we want to show that Merge Sort always returns a fully sorted list using a proof by induction.

Inductive hypothesis: For a given list with n or less than n elements, Merge Sort returns a fully sorted list.

Base case: The list contains a single element (n=1).

Inductive step: Suppose that Merge Sort returns a sorted list given a list of n elements. We want to prove that Merge Sort returns a sorted list given a list of n + 1 elements.

For a list with n+1 elements, Merge Sort divides the list in half, then sorts each of those lists (which each have a length of (n+1)/2). The length of those sub-lists will be less than or equal to n, and we know that those sub-lists will be sorted because of the inductive hypothesis. The final constructed list is sorted by comparing the first value in each sub-list and appending the lower one and removing it from its sub-list. Since the sub-lists are sorted from left to right, we compare the first element in each and append the lesser of the two to the final list. We know that the first element in the left sub-list is less than all the other elements in the left sub-list, and that the first element in the right sub-list is less than all the other elements in the right sub-list, so the smaller of the two is the smallest number in both lists. This ensures that the final list is always sorted as elements are added to it, because

only the current minimum value is added. Therefore, the final list returned will always be sorted.

3. Union-Find

(a) Construction using Doubly Linked Lists

The Union-Find data structure (DS) can be implemented using doubly linked lists. The Union-Find DS is comprised of two classes: the `Node` class and the `Union-Find` class.

The `Node` class has the following attributes:

- `val` - the value of the node
- `prev` - the previous node
- `next` - the next node
- `head` - the head node

The `Union-Find` class has the following attributes:

- `head` - the head node
- `tail` - the tail node
- `len` - the number of elements in the list

The Union-Find class also supports the following operations:

- `make_set(i)`: This function returns a Union-Find DS whose head and tail are `i` and length is 1. Because the function simply initializes an instance of the Union-Find class, the runtime is O(1).
- `union(A,B)`: This function first compares the `len` of the two sets. Then, for each node in the smaller set, the value of `head` is assigned to the head node of the other set. Finally, the sets are linked together (the tail of the longer set is linked to the head of the shorter set). The DS `tail` is now the tail of the shorter set, and `len` is now the sum of the lengths of the sets. The only operation in this function that isn't constant time is the updating of the `head` value for each `Node` in the shorter set. As a result this function runs in $O(\min(n_A, n_B))$.
- `find(i)`: This function simply returns the `head` attribute of the given node. This runs in O(1) time as it is just returning an attribute.

(b) Improved Runtimes

We can construct a Union-Find DS using a tree structure. Let us define the `Node` class as having the following attributes:

- `val` - the value of the node
- `parent` - the parent node

The `Union-Find-Tree` class has the following attributes:

- `root` - the topmost node in the tree
- `len` - the number of elements in the tree

With this new structure, the operations mentioned in part (a) are implemented differently, and with altered runtimes:

- `make_set(i)`: Returns a `Union-Find-Tree` object with the i as `root`. This simple assignment runs in O(1) time.
- `union(A,B)`: Assigns `parent` of the root of the smaller tree to the root of the larger tree and updates `len` to be the sum of `A.len` and `B.len`. This operation, similar to `make_set()`, runs in O(1) time, especially since the `len` of the trees are stored as attributes in the tree class.
- `find(i)`: For a given node, i, the function traverses through the parents of the node until it arrives at a node with no parent. This node (the root node) is then returned. This operation runs in O(log(n)) time, where n is the number of elements in the tree. This is because in the worse case scenario, the node is a leaf, so there must be log(n) traverses up to the root node.