Homework 6

Sam Daitzman // DSA Spring 2020

Exercise 1: Academitis

> This question builds on problem 2 on Homework 4, in which you designed an algorithm to find a list of all possible patients starting with patient zero.

a) O(n) Time Algorithm

> In this question, you will complete the runtime analysis for your algorithm. Suppose that you are given a list of all students at Olin, and for every class at Olin, you are given a list of students in that class. Further, suppose that at most 25 students are in a class and every student takes at most 5 classes (so these are constants that do not depend on the number of students at Olin). Update your algorithm from HW4 to pre-process these class lists so that your algorithm now runs in O(n) time, where n is the number of students at Olin (**Hint**: you may want to use hash maps).

As defined previously, my algorithm followed this procedure:

> Assuming we have a single affected student and a list of their classmates, and we are trying to find all classmates who are in a chain of classes where one student attended a class with patient zero, and another may have attended a class with that first student (and so on), we can use queues in a maze-search-like implementation to search the lists.
>
> Essentially, we treat each student as a location node in the maze problem we previously considered. Each *StudentNode* must have a property *Checked*. We will create a queue *Quarantine*.
>
> This is a depth-first search algorithm, which does not ensure that the *Quarantine* queue is ordered with the students most likely to be infected (with the shortest chain of classroom contacts from patient zero) towards its start. If trying to prioritize students, it would make sense to set a *Distance* key on each student to a value that represents the current depth in the tree. Then, the list could be sorted to prioritize who to reach out to or quarantine first.

There are several issues here in terms of the time-efficiency of this algorithm. For one, we did not previously know that constraints like the number of students at Olin, the maximum number of potential classmates, or the maximum number of classes each student takes. Now that these are all constrained, it becomes possible to design an algorithm to map the potential academitis patients in $O(n)$ time with pre-processing.

In pre-processing, I would construct a hashmap with every student's name as a key, and the values would contain a True/False `Checked` flag that indicates whether that student has been checked already. Additionally, each item in the hashmap would store a value that is a list of other students with shared classes. I would construct this by iterating through every class and, for each student in the class, adding every other student in the class to the list of classmates attached to their name, skipping duplicates. This would result in the following dict structure:

```
{
    "Sam Daitzman": {
        checked: False,
        classmates: ["Grace Hopper", ...]
    },
    "Grace Hopper": {
        checked: False,
        classmates: ["Sam Daitzman", ...]
    },
    "Alan Turing": {
        checked: False,
        classmates: ["Ada Lovelace", "Charles Babbage", ...]
    },
    ...
}
```

Now, the algorithm can behave the same way as before, but can take a shortcut directly to the classmates of every student. Since the number of classmates at maximum is a finite constant, and this algorithm will now only perform at most one constant-time operation for each of these students before setting the `checked` flag to `True` and never checking them again, the algorithm will perform `n` constant-time operations. This makes the algorithm's time efficiency $O(n)$.

b) Proof by Contradiction

> Previously, you argued why every student added to the list was a potential patient and why the returned list contains all such students. This question will formalize that last part using a proof by contradiction. That is, suppose that a student caught academitis but was not added to the list. Prove that this leads to a contradiction.

Suppose a student was affected by academitis but was not added to the list of patients. For a student to not have been added to the list of patients, each peer in their `classmates` list must also not be affected, since the `classmates` list is generated in a bidirectional fashion. If another student were in a class with the affected student, each of their names must appear in the list for that class, and must therefore appear in their respective classmate lists. Since this generalizes to a student separated by multiple class steps, every student who has shared contact (directly or indirectly) with the patient we are considering must appear in their list of `classmates`. Each of those students must also include a bidirectional link through their respective `classmates` lists, some series of which must lead to our patient. The algorithm follows all of these chains. This means that our patient was affected by academitis but did not attend class with anyone who could have been exposed to academitis in a class, which is impossible. So, every student who has potentially been exposed to academitis must be on the list of `patients`.

Exercise 2: Merge Sort Inductive Proof

> In class, we argued why Merge Sort always returned a fully sorted list. Formalize that argument using a proof by induction. Your proof should contain your inductive hypothesis, a base case, and an induction step.

Want To Show: merge sort will always return a fully sorted list when called on a list of items of any length, whose values can be compared.

Inductive Hypothesis: suppose our merge sort algorithm, as it compares at each index of the two lists being merged, will only merge values such that lesser values in the resulting list will be placed to the left of greater values.

Base Case: at the base of the merge sort tree, we have exactly one item at each tip of the tree. Since there is only one value in each list at this point, each list is sorted because it would be impossible for it to be unsorted.

Inductive Step: as we merge back up the tree at each stage of merge sort, we maintain sortedness. At each step, supposing the list passed to that stage was sorted, the list will maintain sortedness before being passed up.

Thus, we show that for a list of any length, mergesort will maintain sortedness at every stage running up the tree. Since the return value of mergesort is an $n + 1 + ... + 1$ step case, sortedness will be maintained to this final step and a sorted list will be returned.

Exercise 3: Union-Find Variants

Union-Find is an abstract data type and supports the following operations

- `make_set(i)`: Creates a set with a single node i.
- `union(A,B)`: Merges two sets A and B.
- `find(i)`: Given a node i, returns the head element of i's set.

a) Union-Find with Doubly Linked Lists

Explain how to construct a Union-Find DS using doubly linked lists with the following runtimes:

We can use a collection of doubly-linked lists to implement the union-find abstract data type. We'll track a list of all of the items in the data structure, and each node will be similar to that of a traditional DLL, but each node will contain an additional property that tracks the head (first element) of the set that each node is part of. To support our operations, we will amortize recording the length of each set across every operation on the DLL-like class we use. In other words, as each set grows or shrinks, we'll track its length so that fetching its length later can be $O(1)$.

- `make_set`: $O(1)$. To create a new set we create a single DLL node, with a `set` property equal to the head for this set. This is an $O(1)$ operation because we are only operating on/creating fixed constant data.
- `union(A,B)`: $O(min(nA, nB))$, where nA is the number of elements in A and nB is the number of elements in B. To union one set with another, we check the length of the two sets. We link the last element of the first set with the first element of the second (shorter) set, then we iterate through every item that was formerly part of the shorter set and adjust its `set` property to link to the head node for the first (longer) set. This will perform in $O(min(nA, nB))$ because we perform a few brief constant-time operations at the beginning, and then only perform one constant-time assignment operation for each item in the shorter set we are calling `union` on.
- `find(i)`: $O(1)$. To implement `find` we simply return the head node linked to by the `set` property on the node we are interested in. This is $O(1)$ because looking up one property is only one operation that takes constant time.

b) Union-Find with Tree Structure

Explain how to construct a Union-Find DS using a tree of elements with the following runtimes: Instead of linking our nodes linearly, we can link them in a tree-like structure to keep track of the union-find datastructure.

- `make_set`: $O(1)$. To run `make_set` with this form of union-find is very similar to the previous approach. We create a node, and link it to nothing to begin with.
- `union(A,B)`: $O(1)$. Union performs better in this implementation, with $O(1)$ time complexity, because to link in a new set, we simply add one set's head element as a child of the other set's head element. This can be chained infinitely. Performing this linking can be done in $O(1)$ because we are only doing two constant-time operations (linking in both directions to keep track of parent/child relations).
- `find(i)`: $O(log(n))$. To find the head of the set, we climb up the parent tree until we reach the top-level head node, which will have no parent set. Amortized across the tree, this will perform in $O(log(n))$ time complexity because it will need to climb $log(n)$ layers up the tree which dominates the brief constant time needed to implement the operation.