Homework 4

Sam Daitzman // DSA Spring 2020

Exercise 1: Dining Hall Foods

> Olin's dining hall is considering implementing a new dining format to improve efficiency. Instead of having stations, all the food options will be available in a single line. Further, students must choose a continuous interval of foods to take. That is, if the available options are [french fries, brussel sprouts, chicken sandwiches, tomato soup, fruit salad], then a student could take french fries, brussel sprouts, and a chicken sandwich, but not just the fries and sandwich.
>
> Luckily, you can use your DSA skills to find the optimal food interval for you. Suppose that there are n foods available, and let h_i be the happiness value for item i (possibly negative if you dislike a food). Give a divide-and-conquer or recursive algorithm to find the interval $[i, i+1, ..., j-1, j]$ that maximizes the sum of happiness scores $\sum_{k=i}^{j} = h_i$. In a few sentences, convincingly argue why your algorithm finds an optimal solution, and analyze the runtime of your algorithm (it might help to write pseudocode to do the latter.) The description of your algorithm should be clear and precise enough that I could write code for it just given your solution.

For a divide-and-conquer algorithm, we'll want to recursively subdivide every layer of the problem by splitting the list in half. At each layer until we hit the base case, the solution has three cases:

1. The solution might be in the left half of the list.
2. The solution might be in the right half of the list.
3. The solution might cross the two halves. In this case, it must include the last element of the left half, the first element of the second half, some number of elements to the left of the first "middle element" and some number of elements to the right of the second "middle element."

So, as we descend the tree, we'll need to test all three solutions at each layer except the base case, and always return the winning sum and the bounding indices $[i, j]$. In the base case, the winning subsequence is always the single element we've descended to, since we cannot descend further.

This algorithm must produce the correct result because in the base case, we find the longest subsequence (the element itself, so always just $[0, 0]$). In every layer returning upwards through the merge, we'll maintain a proper solution because, for each of the three possible cases:

1. If, at this layer, the solution is in the left half of the list: we'll sum the left half, find it is higher than the other two options, and return its bounding indices and sum.
2. If, at this layer, the solution is in the right half of the list: we'll sum the right half, find it is higher than the other two options, and return its bounding indices and sum.
3. If, at this layer, the solution crosses the two halves: it must include the last element of the left half, the first element of the second half, and some number of consecutive surrounding arguments. We'll iterate outwards in both directions from the middle, and on both sides we'll store the greatest result we find and its outer index. We can sum the two sides' greatest bounding indices for the greatest overall solution that crosses the middle. When we compare that to the left and right half sums, if the best solution that crosses the middle is greater than the left and right solutions, we can return the greatest overall bounding indices and the sum.

The performance at each layer in searching for the solution that crosses the middle will be:

$$2 * O(\frac{n}{2}) = O(n); \quad n = \text{number of elements at this layer}$$

Because at each layer, we iterate through the two halves of the list. At worst, checking the left and right halves will be $O(\frac{n}{2})$ each.

Since there are a constant number of $O(n)$ operations, we can say that checking each layer takes $O(n)$ time. Since there are $log_2(n)$ total layers, we can say that the runtime is:

$$log_2(n) * O(n) = O(n * log(n))$$

# Exercise 2: Academitis > The students at Olin are suffering a new version of the freshman plaque called academitis. Fortunately, this unique bug can only be passed between students during class time and there's a known cure – leweekend. Suppose that you want to distribute the cure to all potentially affected students and you've identified patient zero that started the outbreak. Design an your algorithm to find this list. Be sure to give a clear description and mention any supporting data structures you use. Then, in a few sentences, argue the correctness of the output. You do not have to analyze the runtime, and you may assume that you can find a list of a student's classmates.

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.

Exercise 3: Implement Happiness Maximization Algorithm

See `./sdaitzman.py` for my implementation.

Exercise 4: Average Results

Exercise 5: Picky Eating