Homework 1

Sam Daitzman // DSA Spring 2020

Exercise 1

> Explain how you would implement list concatenation in python and analyze the runtime using big O analysis. Be sure to define any variable(s) you use in your analysis. For example, you may let a be the resize constant for a list.

For this problem, we want to take some list l, and append all the elements of another list m to the end of it. Because lists use dynamic sizing, we can reason about a resize constant a, which is the coefficient that determines how the list scales/allocates memory. We do not need to modify, address, insert/delete or change the contents of list l to perform this operation, so we are only concerned with big O analysis of iterating through list m and adding each item from it to the end of list l.

Since iterating through list m and appending each of its items to list l will generally scale linearly over large numbers, we can intuit that it will follow a linear O(n) growth pattern. While the list will occasionally resize, this will not increase significantly with the growth of the list.

Exercise 2

> Subsequences of lists.

Exercise 2A

> Write pseudocode for a function that takes in a list of integers and returns the longest strictly increasing subsequence (as a list). For example, the longest increasing subsequence of [1, 2, 0, 4, 8, 9, 3] is [0, 4, 8, 9].

```
m = some list
last = 0
currentNum = 0

currentList = []

longest = []
```

```
iterate through list m
    currentNum = current element in list

    if first element in list
        current[0] = currentNum
        longest[0] = currentNum
    else
        if currentNum > last
            append currentNum to currentList
        if length(currentList) > length(longest)
            longestList = currentList
    last = currentNum

return longest
```

Exercise 2B

> Analyze the runtime of your function. Be sure to define any variable(s) you use in your analysis.

I'll use the variable n to represent the length of list m.

Since the length of list m is n, it will take n steps to iterate through. This alone means that section will be guaranteed to take O(n) steps, so we can discard the O(1) steps that the declarations above like instantiating the list will take.

Within the for loop, we have another operation that could contribute resizes, appending the current number the loop is iterating past to the list currently being analyzed. This step will take at worst O(n) steps, but will usually take O(1) steps because we're appending to the final element and usually don't need to resize the list.

Therefore, this runtime will be at worst quadratic, or $O(n^2)$ in Big-O notation. For most steps, though, the performance should be closer to $O(n)$.

Exercise 3

> Write a function that implements your algorithm from Q2 and a corresponding test function. For this assignment, you can put all your test cases into a single function. Also, be sure to include appropriate documentation in your code (e.g. docstrings and any necessary comments).
>
> Your tests should cover the following input scenarios: an empty list, a decreasing list (i.e. the longest increasing subsequence has length

1), an increasing list (i.e. the longest increasing subsequence is the whole list), and at least one additional test.

See attached file `sdaitzman.py`