11.2 - Week 11 - Applied - Practical

Introduction

Objectives

- Becoming better acquainted with the implementations of recursive functions.
- Becoming better at spotting recursive solutions to problems.
- Understanding in more detail the implementations of quick and merge sorts, and their complexities.

Recursive functions

As with applied theory questions, you should attempt these problems with a mindset of breaking up an existing problem into smaller, more manageable chunks - This is where recursion is most applicable.

These chunks could actually look almost identical to the original problem, as we saw with merge and quick sort "chunks" - They were just sorting smaller lists than before.

For many of these rudimentary tasks recursion might seem redundant, and that's because in these cases they likely are - iteration would suffice. But, as you can hopefully see with merge and quick sorts, and as you will see as you continue to study next weeks content (and future units content), recursion is a very powerful tool for discussing the solution to complex problems with ease (Especially when that complex problem is on a data structure a bit more complicated than a list).

Digit Sum

You want to compute the **digit sum** of a number.

The digit sum is as it sounds - the sum of digits (base 10). So for the number 979853562951413, the sum of its digits is 9 + 7 + 9 + 8 + 5 + 3 + 5 + 6 + 2 + 9 + 5 + 1 + 4 + 1 + 3 = 77.

Write a function `sum_of_digits(n: int) -> int)`, that computes the sum of digits. You may assume input is always positive.

> ⚠ You cannot use `str` or other string methods. You can solve the problem using only mathematical notation such as `%`, `//`, `+`, etc.

Once you've implemented the function, analyse the Big O complexity, and whether your solution is tail recursive. If your solution is not tail recursive, try making it tail recursive.

Hint:

▶ Expand

Recursion on Linked Structures

Assume we have a Linked List, which implements two new methods:

```
class MysteryList(LinkedList[float]):

    def mystery(self):
        build_properties = [len(self)]

        if self.head is None:
            return build_properties

        while True:
            all_same = True
            saved = None
            current = self.head
            while current is not None and current.link is not None:
                if current.item is None:
                    break
                if saved is None:
                    saved = current.item
                if saved is not None and saved != current.item:
                    all_same = False
                    break
                current = current.link

            build_properties.append(self.head.item)
            if all_same:
                break

            self.mystery_helper_aux(self.head)
        return build_properties

    def mystery_helper_aux(self, current: ListNode):
        if current is None or current.item is None:
            return
        if current.link is None or current.link.item is None:
            current.item = None
            return
        saved = current.link.item
        self.mystery_helper_aux(current.link)
        current.item = saved - current.item
```

Explain what `mystery_helper_aux` and `mystery` are doing, by attempting to execute the function with input list [1, 4, 9, 16, 25]. Analyse their Big O Complexity.

You can assume that for all lists which `mystery` is called on, the list starts off with every `.item` being a number.

Digital Roots

Now that we can compute the digit sum of a number, we want to find the digital root of a number - this is similar to the digit sum, but we repeat the process until the result is a single digit - so for 979853562951413 we get 77, then 14, then 5. So `digital_root(979853562951413) == 5`.

Write a recursive function for `digital_root` which uses your solution to `sum_of_digits`.

Advanced: Analyse the time complexity of `digital_root`.

Palindromic Lists

A list is *Palindromic* if it is the same reversed. For example `[1, 5, "Test", "Test", 5, 1]` is palindromic, while `[1, 4, "Test", "Test", 5, 1]` is not.

Write a *recursive* function `is_palindromic(lst: list) -> bool` that tests if a function is palindromic, in **O(N)** time complexity, where **N** is the length of `lst`.

Hint:

▶ Expand

Advanced: Finite Difference Method

Funnily enough, that `mystery` function from the previous question on Linked Structures can actually be though of as a compression method, in fact:

> For any list of numbers, if these numbers can be written as `n` points `(i, f(i))`, where `i` is the index in the list, on a polynomial of order `k`, then `mystery` will return a list of size `k+1` , and the entire list can be retrieved from these `k+1` numbers.

For more information, see the Finite Difference Method

Your task is to write a recursive method that creates a new LinkedList based on the return value of `mystery`, to retrieve the LinkedList that `mystery` was called on.

For example, since calling `mystery` on [1, 4, 9, 16, 25] returns [5, 3, 2], calling `demystify` on [5, 3, 2] should return [1, 4, 9, 16, 25].