Repeating Patterns Code Literacy + Reflection
Ryan Blocker
CS320
January 31st, 2024

Code: Here is my repeat() routine I submitted to Zybooks:

```
def repeat(lst):
    if lst is None or not isinstance(lst, list):
        return None

    if len(lst) < 2:
        return None

    lst_length = len(lst)
    for pattern_length in range(lst_length // 2, 0, -1):
        if lst_length % pattern_length == 0:
            pattern = lst[:pattern_length]
            sublists_match = all(
                lst[i:i + pattern_length] == pattern
                for i in range(0, lst_length,pattern_length)
            )
            if sublists_match:
                return pattern
    return None
```

Reflection: This problem initially stumped me. I was overcomplicating it in my mind and used multiple pieces of paper to track my loop iterations, which became overwhelming. One of my attempted approaches was to iterate through the given list and identify if characters occurred more than once, then check if the next character also appeared elsewhere in the list after the first occurrence. I planned to add these characters to a separate list called "pattern." However, this approach proved to be overly complicated and time-consuming for both the computer and my brain.

The solution I settled on first checks if the given list is `None` or not a list. Then it checks if the list size is greater than one element; if not, it returns `None`. Before iterating through the list, I create a variable called `lst_length` to keep track of the list size. I then start iterating. For each element in the list from half the list length (`lst_length`

`// 2`) down to element 0, I iterate backwards. I chose to check the back half of the list first because iterating backward from half the length of the list towards the smaller indexes allows the function to check longer potential repeating patterns first.

On each iteration of the loop, I check if `lst_length` is divisible by the `pattern_length` to see if the pattern could make up the entire list without any outstanding elements. If this condition is met, I set the pattern equal to the remainder of the list.

To validate the pattern, I have a boolean variable named `sublists_match` which I set equal to a check that searches the entire list again using the found pattern. It checks if the given list is made up of repetitions of the pattern.

If `sublists_match` returns true, I return the pattern. If not, I return `None`, indicating that the list has no pattern.

The benefit of this solution is its readability and ease of understanding. However, as I will explain later, it may not be the most efficient in terms of time. During debugging and testing, the only issue I encountered was that I initially named my routine "`repeats()`" instead of "`repeat()`", leading to a failure in Zybooks. I was perplexed until I discussed it with Craig earlier this week and he spotted the typo.

Programming Literacy: The time complexity of my "`repeat()`" algorithm is not the most efficient. Here is a breakdown of its time complexity:

The initial checks to ensure that the list is not `None`, has a length greater than zero, and is a list itself, have constant time complexity.

The main part of the program iterates over possible pattern lengths in a loop. Since I cut the list in half, it iterates approximately n/2 times, where n is the length of the list.

Inside the loop, there is a nested loop that checks if the list can be divided into sublists of the current pattern length. This nested loop iterates over the entire list, but the step size is determined by the current pattern length. In the worst case, this loop contributes to O(n) complexity.

The worst-case scenario occurs when the list contains the same element repeated throughout, causing the inner loop to iterate over the entire list for each possible pattern length. In this case, the time complexity becomes $O(n^2)$. Overall, the time complexity of the program is $O(n^2)$. This is because the nested loop, with O(n) complexity, is nested within another loop that iterates approximately n/2 times. So, O(n) * O(n/2) simplifies to $O(n^2)$.