Austin Youngren

CS-320-001

Lab 1 Reflection Essay

Feb. 2, 2024

Algorithm

For my implementation, I chose to use the divisors of the list length as a method of sectioning the list for comparison. For example, if the list length was 12, the divisors list would be [1, 2, 3, 4, 6, 12], for list lengths 9 the divisors would be [1, 3, 9], and for list lengths that are prime, such as 7, the divisor list would be [1, 7].

I took many approaches to get this right, such as recursion, or taking care of all special cases as separate functions within repeat(). Though, it ended up being how I wrote these different approaches that caused me issues, rather than the ideas themselves.

Before the divisors are collected into a separate list, special cases are checked, such as the list being None or having a length less than 2. If these evaluate to True, then None is returned. After these cases, the divisors of the list length are collected in an ordered list, as shown above. The divisor list is first used to check if the list length is prime by evaluating the divisors list against [1, list_length]. If it is prime, the algorithm runs through the list to see if all elements of the list are the same. If they are, this check will return a list with one of the repeating elements (e.g., a prime list length consisting of only ones will return [1]); if not, None is returned.

After these checks, the first and last elements are removed from the divisors list. This removes elements 1 and list length, which are only needed to evaluate lists of a prime length. Then, the divisor list is reversed, making the order largest to smallest. The first (or outer) for-loop, loops through the divisors list, for each divisor, allocating list[0: divisor] to the variable ‘section1’. The inner for-loop increments from the value of the divisor to the value of the list length, where the value of list length is not inclusive. This allows section1 to be compared to the remaining sections of the list (e.g., with a list of length 9, section1 = list[0: 3], the remaining sections would be list[3: 6] and list[6: 9]). If all remaining sections are equal to section1, section1 is returned. Otherwise, the outer for-loop will continue to the next divisor and repeat the process. Finally, if no matches are found for any divisor, the algorithm returns None.

Challenges

I faced many challenges in accounting all the special cases that come along with the repeating pattern problem. This challenge ultimately stemmed from my issue in finding a way to evaluate the first section of the list to all other equivalent sections. Due to this, many versions of my algorithm, some including recursion, only compared the first section to the last section, or the first section to the second section. This would result in non-patterned lists, returning a pattern (e.g., [1, 2, a, a, 1, 2] or [1, 2, 1, 2, a, a] returning [1, 2]). Additionally, for many versions of the

algorithm, the coverage of one special case broke a previously covered case. Due to this, I eventually decided to start over and create the current divisor implementation.

Complexity

Overall, the time complexity of the algorithm is O(n^2) in the worst case.

To start, obtaining the divisors list, or checking for same-element prime lists, makes this implementation's big-Oh, O(n), before evaluating the core of the algorithm. This is due to each of these checks making n comparisons in the worst case.

The core of the algorithm runs at a quadratic complexity, so O(n^2). This is due to the outer loop iterating for the number of divisors, and the inner loop iterating over each section, of length divisor, to check if all remaining sections' element are equivalent to the first section.