* Chosen approach

The algorithm I chose to write for this problem first divides the list in two. It then assigns the first half to a variable pattern. It then checks the pattern against the next sequence in the list of the same size, pattern\_length. For this algorithm, if the list contains an even number of elements and the pattern is exactly half of the sequence in the list, this is our best case time-complexity. If the list contains an odd number of elements or if the patterns don’t match, we decrement the pattern\_length and reassign the new pattern with one element sliced off the end. We then compare this element to the sub-sequences in the array using a for loop incremented by the size of the pattern\_length. This process continues until either an subarray of pattern matches all subsequent subarrays, or until our pattern is less than one.

I considered another way to solve the problem which was to start at index 0 of the list and increment the pattern sequence and compare it to the following sequences in a similar fashion within the for loop. This would have been a less optimal design because we would never be able to take advantage of the best case time-complexity when the list is even and the repeating pattern is exactly half the sequence of elements in the list.

* Challenges and Benefits

The benefits of this algorithm is that the code is very simple to understand. It consists of only a couple of if else statements within a for loop. It is very easy to debug when an issue occurs, and the straightforward nature of the loop was simple to write.

The challenges of the algorithm are ensuring that the conditional statements are actually checking what I wanted them to check. The most challenging part was writing the conditional statement that checks that the sequence we are checking the pattern against occupies the final set of indices in the list; when the pattern is equal or not equal to the length of the final sequence of the list. For this I finally settled on checking if the pattern length matched the total length of the list minus the index variable i.

* Modularity

I ended up not writing subroutines for this algorithm. I found that subroutines made the code a bit more complex to read because it is such a straightforward algorithm within a for loop. I considered making a subroutine for slicing off the last element in our base pattern, but it required several arguments to accomplished and made the code look a bit messy and difficult to follow and analyze.

* Big-O

This algorithm runs in O(n^2) time-complexity. This is not an optimal solution to the problem, because the for loop runs inside of a while loop, each of which takes O(n) operations. After writing this algorithm I did some research and found another implementation that runs in O(n) time using a suffix array. I’m doing some research into this data structure, but as of now, it will take some time to understand and implement.