Code Structure:

The code is divided into two folders, the main code and test. This is a good way to package python packages. This way, in the top level folder, one can keep all the interfaces to the main program while maintaining a separate testing environment that can be used in a CI environment. The main parts of the code are:

Stringshifter:

The *utils* file contains the generic utility functions that are used in the package. Apart from the constrain addressing functions like *get_len,* map_case, join_string and precondition checkers, the important functions are *shift_string* which generates *CShift from C* using a Caesar's shift like algorithm and two variants of string replacement algorithms. We have two because in the contiguous case, the replacement of the pattern can be done in a more efficient way.

The *patterns* file contains the four parts of the problem, with each function representing one way of finding and replacing the pattern in a certain way.

Driver:

The main entry point into the program. Contains logic on input and output of problem

Tests:

All the tests, self-contained.

Algorithm:

Contiguous Pattern Case:

This algorithm works in the following way:

```
1: While True:
    • Try finding the pattern (C and CShift) in left and right strings (1/3S and 2/3S) by iteration
    • If both patterns exist, add pointers for this position to a list
2: Use the string joining algorithm to substitute letters in the array
```

Notes:

This algorithm runs in *O (n*m)* [*O (n)* for best case i.e no pattern] where n=*len of S* and m=*len of C*. The string joining/pattern replacement part takes advantage of the fact that the pattern is contiguous and can be run in *O (n)* instead of *O (n*m)*

Non Contiguous Pattern Case:

The challenge here is the fact that all valid subsequences of the right string need to be searched for the pattern. A valid subsequence is one which:

1. Is the pattern
2. Has indices that have not been seen in another pattern (sorted by index). (Case like BCCDD for pattern BCD)

We use the same algorithm as above but instead of comparing both pointers, for the right string, we compare it to valid subsequences remaining. These valid subsequences are generated by filtering the power set for the string using the above criteria. For replacement of the pattern, we have to iterate over the right string for each replacement of the pattern because the pattern is not contiguous. This can be optimized to *O (n+m)* by iterating once over the indices of patterns in the right string and maintaining a dict of index-> alphabet, but we gain nothing in terms of the overall algorithm.

Notes:

The slowest part of the algorithm is the power set generation and overshadows everything else. This runs in *O (2^N)* time. The string replacement runs in *O (n*m)* time.

Usage:

To Run Tests: *python run_all_tests.py*

To Run Program interactively: *python driver.py*

To Run Tests: *python driver.py S="S" C="C" N="1" R="y"*