**Algorithm Pseudocode**

The basic idea of this algorithm is to loop through each character of the input string S. For each character we will loop through the X and Y strings until we find a match and once both are matched can verify that each following character in S matches either the next character expected in X or Y.  
  
If successful a proof certificate can be returned in the form of an array of length S. Each item in the array is the character index of X or Y which matches the character in Sk.

sIndex = 0 # The current character in S we are looking to match

xIndex = 0 # The current character of X we are matching

firstXFound = False # If we have started matching characters in X yet

yIndex = 0 # The current character of Y we are matching

firstYFound = False # The current character of Y we are matching

sNonAdvanceCount = 0

# The main loop – we go through every character in ‘S’ and advance to the

# next character once a match with an x or y character is found. If the

# entire length of x and y are traversed without a match then the loop should

# be terminated to avoid infinite repetition.

while sIndex < s.length and sNonAdvanceCount < max(x.length, y.length)

# If we match the current character in x then we add this to the proof

# certificate and repeat this loop, pointing to the next character in s and

# x.

if s[sIndex] = x[xIndex mod x.length]

if !firstXFound then firstXFound = True

add x[xIndex] to Proof Certificate

sIndex++

xIndex++

# If we match the current character in y then we add this to the proof

# certificate and repeat this loop, pointing to the next character in s and

# y.

else if s[sIndex] = y[yIndex mod y.length]

if !firstYFound then firstYFound = True

add y[yIndex] to proof cert

sIndex++

yIndex++

# If the current character in ‘S’ does not match the current character in

# ‘X’ or ‘Y’ then we have a special if/else block below…

Else

# If we have spotted the first instance of the X and Y string then the

# character must have been added in from another source and we should

# return this as a failure (that is, S is not just an interweaving)

if firstXFound and firstYFound

return Proof Certificate

# Otherwise if the first X character has not been found yet we should

# just advance the xIndex and look for the next X character.

elsif !firstXFound xIndex++

# Likewise for Y

elsif !firstYFound yIndex++

# And always increment the counter saying no match was found. If this is

# equal to the length of the larger string (X or Y) then the loop will

# terminate and record a failure

sNonAdvanceCount++  
  
return Proof Certificate

The ‘while’ loop and if block directly under the while loop will be performed for each of the n characters in S, and so will run in 2n time. In addition the else block may run through each element of x and y in the worst case before finding the first matching

x or y value, potentially adding max(x.length, y.length).

In addition to the above loop the final proof certificate will need to be inspected to make sure there are no empty indexes in the array. This check will be bounded by the length of the array n.

So the worst-case run time for the algorithm will be O(3n + max(x.length, y.length) where n is the length of the input string S.

**Results**

For the values I fed into the implemented algorithm I used groups of different-sized S, X, and Y strings. In order to measure the run time I set a global variable to count the number of operations performed by the code, incrementing it for every iteration of the while loop (always run n times), the top if statement (always run n times) and within the top else statement (runs at most max(x.length, y.length) times).  
  
In addition I incremented the counter by len(S) during the final verification of the proof certificate to determine if the signal was interweaved, given that this will involve that many comparisons.

The following table contains the results, reflecting an accurate estimated upper bound:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Input #** | **S Length** | **X Length** | **Y Length** | **Est. Upper Bound** | **# of Operations** |
| 1 | 9 | 3 | 1 | 30 | 27 |
| 2 | 9 | 3 | 1 | 30 | 20 |
| 3 | 9 | 3 | 1 | 30 | 20 |
| 4 | 9 | 3 | 1 | 30 | 26 |
| 5 | 9 | 3 | 1 | 30 | 27 |
| 6 | 18 | 6 | 2 | 60 | 53 |
| 7 | 18 | 6 | 2 | 60 | 28 |
| 8 | 18 | 6 | 2 | 60 | 26 |
| 9 | 18 | 6 | 2 | 60 | 39 |
| 10 | 18 | 6 | 2 | 60 | 44 |
| 11 | 27 | 9 | 3 | 90 | 36 |
| 12 | 27 | 9 | 3 | 90 | 46 |
| 13 | 27 | 9 | 3 | 90 | 41 |
| 14 | 27 | 9 | 3 | 90 | 55 |
| 15 | 27 | 9 | 3 | 90 | 39 |