CSC 505 – HW 1
Greg Timmons

In summary my algorithm works by building a tree of all possible substrings. First a left hand tree is built by adding each possible substring. The second string is added to the same tree. Node which share common children from both string A and B are marked. Finally, all nodes that are marked as having been produced by both trees are counted as each represents a unique substring.

STEP1:

This step constitutes construction of the first tree. We need to include all possible substrings, which means we use the following pseudocode.

```
String a;
for( i from 1 to strlen(a) )
    # add letter a[i] to the head and return the created node.
    node = head.add_left( a[i] );
    for( j from i to strlen(a) )
        # add letter a[i+j] as a child to the preceding node.
        node = node.add_left( a[i+j] )
```

Since we are able to find each nodes position as immediate next to the prior nodes positions I am able to place each letter in the tree in O(1) time. This mean the construction of the tree will take $\Theta(n^2)$ time.

$$\sum_{i=1}^{n-1}\sum_{k=i+1}^{n} 1 = \sum_{i=1}^{n-1} n - i = \left(\sum_{i=1}^{n-1} n\right) - \left(\sum_{i=1}^{n-1} i\right)$$
$$= n(n-1) - \frac{n(n-1)}{2} = \frac{n^2}{2} - \frac{n}{2}$$
$$\equiv \Theta(n^2)$$

STEP2: For step 2 we will repeat the above process with the second string. Nodes that were added in both the first and second string will be marked for later counting. We could easily optimize this second loop to not add nodes if there is not an already existing node from the first string, since we know this will never increase the final count. We could in addition ensure that the shorter string tree is also always built first. This is a good tool for bounding the runtime of the algorithm.

```
String b;
for( i from 1 to strlen(b) ) :
    # mark letter b[i] as seen in both if a node exists.
    if( head.has(b[i]) ) :
        node = head.markSeen( b[i] );
    for( j from i to strlen(b) ) :
        # mark letter b[i+j] seen if a node exists.
        if( node.has(b[i+j]) ) :
            node = node.markSeen ( b[i+j] )
```

I will say there is probably a better way to bound the worst case, but as a simple analysis we can say that the inner loop can never run more than n steps before it runs out of seen nodes. Therefore, the runtime of this loop should be $O(nm)$ as it must run through m letters in the second word.

STEP 3:
We then do a depth first search to count all of the substrings. As an easy shortcut, we can guarantee there are less than both nm or n^2 substrings total. There for the runtime for the third step will be dominated by the first two steps and will be no greater than $\min(O(n^2), O(nm))$, and I will simply ignore this term as it will not change the asymptotic runtime of the algorithm.

STEP4:
Putting all together since we know that m is larger the n, and therefore $O(n^2) \in O(nm)$ the runtime of this algorithm should be at most $O(nm)$.