22805904
September 30, 2014

CS176_Fall2014 HW 1

Problem 1

a. If we let $\beta = xyxyxy$ and $S = xyxyxyxy$, then we can see that the two tandem arrays overlap from index 2 onwards, if we let the first tandem repeat $\alpha_1 = xyxyxy$ starting at index 1, and $\alpha_2 = xyxyxy$ starting at index 2.

b. In order to find the maximally tandem arrays, we want to use the Z algorithm to find every occurance of β in S. We'll also want keep track of the the number of repeats that appear in S, let's call that array M, so that $M[i]$ will be the the number of repeats up until i. If the Z-score of i is equal to $|\beta|$, then know that the repeat β and $M[i] = 0$, then the repeat must be beginning at i, otherwise it's in the middle of a full repeat, and $M[i] > 0$. Thus as we go through the array ($\beta\$S$), we set $M[i + |\beta|]$ to $M[i] + 1$ (only if Z-score of i is the cardinality of β). Now that we have this array M, we can go through, and if there are any indices that have values greater than or equal to 2, we know that there was at least two repeats of β, which would constitute as a maximal tandem array. Because we only want to find the odd tandem repeats, we look for all $\forall k \in \mathbb{Z}$ where $M[i] = 2n + k$

   The Z Algorithm runs in $O(|\beta| + |S| + Z_s)$. We know that $|\beta|$ is in the order of $|S|$, and we know that Z_s, the time to compute the Z-score is linear, then we can say the runtime of this solution is $O(|S|) \implies O(n)$.

Problem 2

First we build a suffix tree of T. A unique substring is any string that from starts from the root and continues down to a leaf node. This is the inherent nature of the suffix tree. So let us take a look at one of these paths from the root to an edge node. Now this path needs to have at least two nodes including the root, let us call the branch length Y, and the branch length that precedes it X, so the path would be something like $\{root \to \cdots \to X \to Y\}$ We know that the path that ends at Y is not unique as it will have another child, so to find a minimally unique substring, we take YX_1, where X_1 is the first prefix of the X leaf node (not including the terminating symbol \$.

So to implement this, we want to keep track of the starting and ending indices for each branch length, so for example we will want to keep track of (x_i, x_j) and (y_i, y_j) and so forth. So to return all minimally unique substrings we just traverse the tree and return the string that occurs from x_i to y_i, (if x_i starts from the node.) Because we have the constraint l, if $y_i - x_i > l$, then we don't return that substring. We know that suffix tree construction can be done in $O(m)$ and searching can be done in $O(m + \log(n) + j)$, where j is the number of matches, and $\log(n)$ is the number of steps for binary search.

Problem 3

We use a modified suffix array, where instead of placing actual suffixes in each "index" we put the k-mers. Finding all the k-mers, and the construction of a suffix array is $O(n+n) \implies O(n)$ time, and then searching for frequency will take $O(k\log(n))$ time using binary search.

Say $k = 3$ and $S = ebcdebcd$

1	ebc
2	bcd
3	cde
4	ebc
5	bcd

unsorted k-mer array

$\rightarrow$

2	bcd
5	bcd
3	cde
1	ebc
4	ebc

sorted k-mer array

Problem 4

We can just generate a "super" generalized suffix tree, so that if we had four strings $\alpha, \beta, \gamma, \delta$, then we would concatenate them with non-alphabetical charachters, e.g. our new string would be $''\alpha\$\beta\heartsuit\gamma \blacksquare \delta\spadesuit''$. Now we perform "LCP" for each pair of strings. You find LCP by looking for the deepest node that is a common ancestor of two nodes that contain the "termination characters" of the two strings that you're trying to compare. There exists a O(1) algorithm to perform LCP, but we didn't cover it in class. This turns out to be $O(kn + p)$ if p is the number pairs, and all the strings are of length n.