COT 6417 - Algorithms on Strings and Sequences
Fall 2020

Name: Nabila Shahnaz Khan

PID: 5067496

COT 6417
Algorithms on Strings and Sequences
Fall 2020
Midterm

### Question 1:
Solution:
Here, for strings **S_1** and **S_2** (length of **S_1** and **S_2** are **m** and **n** respectively), we have to build a generalized suffix tree (GST) if their lengths are equal, meaning if **m** = **n**. Otherwise, **S1** and **S2** won't be cycling strings.

If their lengths are equal then while building the tree, for each node we have to keep an array **TN** of length 2 to store some additional information. Basically, for a node **'v'**, initially array **TN** will be set to 0. If there's a terminal edge from node **'v'** for string **S_1**, then set **TN**[0] = 1, otherwise it will remain 0. Similarly, if node **'v'** has a terminal edge for string **S_2**, then set **TN**[1] = 1 .

Check 1:
In the GST, first we have to look at the leaf labeled (2, 1) as the path label of this leaf is **S_2[1,n]** (it represents the whole **S_2** string). From this leaf, we have to visit upwards through its internal nodes up to the root (jumping from node to node). While visiting upward, we have to keep track of internal nodes which have outgoing terminal edges for string **S_1**, means their **TN**[0] = 1. For such an internal node **'u'**, store its string depth [length of the substring represented by path between root and **'u'**] into an array called **String_Depth** (use an array cause there might be multiple such internal nodes). Suppose the value of string depth is **x**.

Check 2:
Now, from root, we have to visit the path between root and node labeled (1,1) [this path represents the whole string **S_1**]. We have to jump the first (**m – x)** number of characters (**x** stored in **String_Depth**[0]) and see if there's an internal node **v'** after that with terminal edge for **S_2** (**TN**[1] = 1 for **v'**) and the label of the leaf node from **v'** should be (2, x+1). If these conditions are true, then we can say that **S_1** is cyclic to **S_2**.

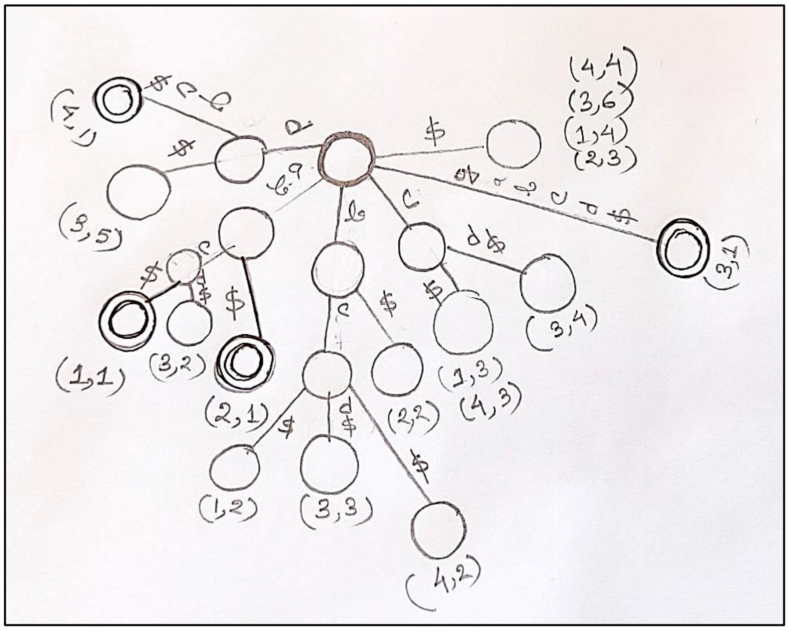
If there are multiple values stored in array **String_Depth**, then we have to perform 'Check 2' for all the string depth values stored in the array until 'Check 2' condition becomes true or all the stored values have been checked. This is because, between two cycling strings, there can be multiple possible values of α and β (Ex: when two strings are exactly same).

Runtime:
Building GST will require **O(n+m)** time. And time required for 'Check 1' and 'Check 2' is bounded by the number of nodes. So total runtime required is **O(n+m)**

### Question 2:
Solution:
A set **S = *{s_1, s_2, .., s_Z}*** of **Z** distinct strings is given. We can assume that none of the strings are completely similar to each other. Means, $s_i \neq s_j$ for any i,j $\in$ **Z**.

For the given set of strings **S**, first we have to build a generalized suffix tree (GST). Now, for each string **s$_i$**, we have to visit the leaf node for its suffix **s$_i$** [1,m] where **m** is the length of string **s$_i$**. The label of the leaf node should be (i, 1). If the edge between this leaf node and its parent node **'u'** is a terminal edge, then we can say that the string **s$_i$** is a substring to another string in set **S**.

Example:
Suppose, S = {abc, ab, gabcd, dbc}. The GST is given below:



As leaf (1, 1) and (2, 1) has terminal edges, strings ***s$_1$* and *s$_2$*** are substrings of other strings. Whereas, leaf (3, 1) and (4, 1) don't have terminal edges, so strings ***s$_3$* and *s$_4$*** are not substrings to any other string.

Runtime:
To build GST, required runtime is **O(n)**. And to check for each string **s$_i$**, required runtime is constant. So, to check for **Z** number of strings, required runtime will be **O(Z)**. AS, Z << n, we can say total required runtime is **O(n)**.

### Question 3:

Solution:
First, we have to build suffix tree ST for the given text **T** of length m. Now, the value of **k** is known. Starting from root node **'u'**, we have to look into all immediate descendant node of **'u'**. For such an internal node **'v'** (immediate descendent to 'u'), we have to see if string depth of node **'v'**, known as **depth(v),** is equal to **k**. Here, string depth is defined as the number of characters on the path between root node **'u'** and it's descendant **'v'**.

Case 1: If **depth(v)** $\neq$ **k**, we can discard that node **'v'** from our consideration.

Case 2: If **depth(v) = k**, consider these **k** characters (between root **'u'** and node **'v'**) as string **α** and check the next **k** characters after node **'v'** (for all the branches coming out of **'v'**). If starting **k** characters of branch **b$_i$** from **'v'** are a match to **α**, that means the pattern **αα** is a substring of the text **T**. By looking into the label of the leaf node for that particular branch **b$_i$**, we can figure out the position of the occurrence of pattern **αα** in text **T**. By checking all the descendant nodes of root and all the possible branches coming out of descendant **'v'**, we are considering all the possible patterns **αα** that might occur in **T** where length of **α** is **k**.

Runtime:
Building ST for text **T** will require **O(m)** time. Considering all the descendants of root will require **O(|∑|)** time. As alphabet size is constant, this time can be considered as constant. And for each descendant, number of comparisons will be bounded by ($\sum * k$) which is also constant time [as **k** is constant]. So total runtime is **O(m).**

### Question 4:
Solution:
First, we have to build a suffix tree ST for the given string **S** of length **m**. After that, we have to apply depth-first-search (**DFS**) on the suffix tree to count the number of descendant nodes of each node. Here, initially, number of descendant nodes for all the leaf nodes will be set to 0. And for an internal node **'v',** its number of descendant nodes will be counted by adding the number of descendants of all of its child nodes.

While apply the DFS, at any stage if we see that for a node **'v'**, its number of descendants is equal to the given number **k**, then we have to store the path label between root and node **'v'** (suppose substring **x**) to an array named **String_track**. The reason behind this action is the path label (substring) between root and node **'v'** represents the substring that occurs **k** times in the string **S**. If any substring occurs **k** times in **S**, that substring will be present at the beginning of **k** suffixes of S; hence node **'v'** will have **k** number of descendant nodes in total.

After we have traversed the whole tree ST using **DFS**, we will look into the array **String_track.** The longest substring **x** within this array will be the longest substring of **S** that occurs exactly **k** times in **S**.

Runtime:
Here, building the suffix tree requires **O(m)** time. Time required to traverse the whole tree using **DFS** is bounded by number of nodes, so requires **O(m)** time. And finally figuring out the longest substring from the array **String_track** doesn't require more than **O(m)** as number of stored substrings in this array is also bounded by the number of nodes. So, overall runtime for this algorithm will be **O(m).**

### Question 5:
Solution:
As we know, using the concept of bit vector along with **"Boundary rank"** and **"Small rank"** table, we can calculate **Occ(L,i,c)** in constant time. Here, **Occ(L,i,c)** returns how many times character **'c'** occurs within substring **L[1, i].** This can be done in constant time cause it only calculates block number (length n/block size t) and offset position of index i in the block (i % t) and accesses **"Boundary rank"** table once and accesses **"Small rank"** table once. So runtime is **O(1)** for **Occ(L,i,c)** once preprocessing is done.

Now, for each **"Small rank"** table, space required is $\sqrt{\boldsymbol{n}} * \boldsymbol{logn} * \boldsymbol{log}(\boldsymbol{logn})$ which is **O(n)**. **"Boundary rank"** table requires space less than **n**, so we can say space requirement is **O(n).** For alphabet size of **|∑|**, we will need **|∑|** bit vectors. So, total space required for **|∑|** number of small rank tables will be **O(|∑| * n)** which is constant as alphabet size is constant. Similarly, **"Boundary rank"** tables can be store using **O(n)** space.

So it is evident that, using the function **Occ(L,i,c)** we will be able to count the number of occurrences of **‘c’** within range **L[1,i]** in **O(1)** time and using only **O(n)** bits to store any auxiliary information.

For a given string **S**, **Q(i,j,c)** should return the number of times character **‘c’** occurs in the substring **S[i,j].** So if for a call of **Q(i,j,c),** if we call **Occ(S,i-1,c)** [this will return how many times **‘c’** occurs in range **S[1, i-1]]** and **Occ(S,j,c)** [this will return how many times **‘c’** occurs in range **S[1, j**]] and finally subtract **Occ(S,i-1,c)** from **Occ(S,j,c)**, that will return how many times **‘c’** occurs within the range **S[i, j].** So, **Q(i,j,c) = Occ(S,j,c) - Occ(S,i-1,c).** As mentioned before, **Occ(L,i,c)** function takes **O(1)** time, so **Q(i,j,c)** will also takes **O(1)** time and it’s space requirement will **O(n)** bits.