COT 6417

Algorithms on Strings and Sequences

Fall 2020

Homework Assignment 1

1. **Given two strings A and B, of lengths n and m respectively, describe an O(n + m) time algorithm that finds the longest suffix of A that exactly matches a prefix of B.**

Solution:

This problem can be solved using Z-algorithm by calculating the Z-values beforehand. First we have to form text string T by concatenating the strings A and B so that, T = B$A [here $ character doesn't belong to A and B]. Length of text T will be (m+1+n). Next, we have to calculate the Z-values of text T using the Z-algorithm. While calculating the Z-values for any position i [where $m + 1 < i \leq m + n + 1$], keep track of position i with max Z-value Z[i] which is also a suffix of A. If Z[i] = (m+n+1)-i+1 then Z[i] at position i represents length of suffix of A at i position. Thus, this position will represent the longest suffix of A that exactly matches the longest prefix of B.

Example:

A = abdgacktacm, B = acktacmgabd, T = acktacmgabd$ abdgacktacm

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
T	a	c	k	t	a	c	m	g	a	b	d	$	a	b	d	g	a	c	k	t	a	c	m
Z-value	-	0	0	0	2	0	0	0	1	0	0	0	1	0	0	0	**7**	0	0	0	2	0	0

B, m **A, n**

Consider the Z-value for the range T[13,23]. At position i = 17,
(m+n+1)-i+1 = 23-17+1 = 7 = Z[17], so T[17,23] is suffix of A and Z[17] is the suffix with maximum length within the range T[13,23].
So, at position i = 17, longest suffix of A matches the longest prefix of B

Runtime: As we know, the runtime for calculating Z-value is O(|T|). So, runtime required to find the longest suffix of A that exactly matches a prefix of B using this algorithm is O(m+1+n) ≈ O(m+n).

2. **Let T be a text string of length m and let S be a multiset of n characters. The problem is to find all substrings of T of length n that are formed by the characters of S. Note that, for this problem, the order of the characters from S that appear in T does not matter. So, for instance, if T = aabxyaba and S = {a,a,b}, then both substrings aab and aba fit the solution. Provide an algorithm for this problem that runs in O(m) time. Assume that the alphabet is of constant size.**

Solution:

Here, alphabet size is constant. So, by counting a frequency table for the characters in set S and modifying the Z-algorithm, we can find all substrings of T of length n that are formed by the characters of S in linear time. In this modified version, the Z-value array will store the length of substring of T at position i that are formed by characters of S. Suppose, alphabet size is C.

Algorithm:

```
S ← set of n characters
T ← text string
L ← 0
R ← 0
Z[m]  ← 0   // used for storing modified z-value
freq[C] ← 0     // initialize ‘freq’ array of length C with 0. It will store the no. of occurrences of distinct characters in S
count[C] ← 0  // initialize ‘count’ array of length C with 0. It will store frequency of distinct characters of S within current
                 [L,R] range

// stores the number of occurrences of each distinct character in S in the array ‘freq’
for i = 1 to n:
        x = S[i]   //stores the character at i position of S in x
        freq[x] ++  //increases frequency for the character stored in x, here x can be mapped to freq array index using
                       ASCII value or a dictionary, a simplified version has been shown in the algorithm

for i = 1 to m - 1:

          if i > R:          //Similar to Case 1 of Z-algorithim
                L = R = i;
                // freq[T[R]] > 0 checks if character T[R] belongs to S and count[T[R]] < freq[T[R]] checks if the
                frequency of T[R] within [L,R] range is less than its frequency in S
                while (R < m and freq[T[R]] > 0 and count[T[R]] < freq[T[R]])  do:
                        R++
                        count[T[R]] ++
                end while
                Z[i] = R-L
                R--
          else:   //Similar to Case 2 of Z-algorithim
                L = i
                R++
                while (R < m and freq[T[R]] > 0  and count[T[R]] < freq[T[R]]) do:
                        R++
                        count[T[R]] ++
                end while
                Z[i] = R-L
                R--
          end if
          if count[T[i]] > 0 then count[T[i]]--   //Update count array for current range[L,R]
          if z[i] == n return T[i, i+n-1]   //returns substring T[i, i+n-1] of length n which is formed by the characters of S
end for
```

Runtime:

Runtime of Z-algorithm is $O(|T|)$. So, runtime of this modified version of Z-algorithm is $O(n+|T|) = O(n+m)$. As $n \leq m$ ($|substring| \leq |string|$), we can say, $O(n+m) \approx O(2m) \approx O(m)$.

Example:

T = aabxyaba and S = {a,a,b}, L = R = 0, |T| = m = 8, n = 3

alphabet	a	b	c	d	…	y	z
freq	2	1	0	0	…	0	0

alphabet	a	b	c	d	…	y	z
count	0	0	0	0	…	0	0

For i = 1:

i > R: set L = R = 1

T	**a**	a	b	x	y	a	b	a
Z-val	**3**							

alphabet	a	b	c	d	…	y	z
count	1	1	0	0	…	0	0

L = 1, R = 3

return **T[i, i+n-1] = T[1, 3]**

For i = 2:

i ≤ R: L = 2, R = 4

T	a	**a**	b	x	y	a	b	a
Z-val	3	**2**						

alphabet	a	b	c	d	…	y	z
count	0	1	0	0	…	0	0

L = 2, R = 3

For i = 3:

i ≤ R: L = 3, R = 4

T	a	a	**b**	x	y	a	b	a
Z-val	3	2	**1**					

alphabet	a	b	c	d	…	y	z
count	0	0	0	0	…	0	0

L = 3, R = 3

For i = 4:

i > R: L = R = 4

T	a	a	b	**x**	y	a	b	a
Z-val	3	2	1	**0**				

alphabet	a	b	c	d	…	y	z
count	0	0	0	0	…	0	0

L = 4, R = 3

For i = 5:

i > R: L = R = 5

T	a	a	b	x	**y**	a	b	a
Z-val	3	2	1	0	**0**			

alphabet	a	b	c	d	…	y	z
count	0	0	0	0	…	0	0

L = 5, R = 4

For i = 6:

i > R: L = R = 6

T	a	a	b	x	y	**a**	b	a
Z-val	3	2	1	0	0	**3**		

alphabet	a	b	c	d	…	y	z
count	1	1	0	0	…	0	0

L = 6, R = 8

return **T[i, i+n-1] = T[6, 8]**

For i = 7:

i <= R: L = 7

T	a	a	b	x	y	a	**b**	a
Z-val	3	2	1	0	0	3	**2**	

alphabet	a	b	c	d	…	y	z
count	1	0	0	0	…	0	0

L = 7, R = 8

For i = 8:

i <= R: L = 8

T	a	a	b	x	y	a	b	**a**
Z-val	3	2	1	0	0	3	2	**1**

alphabet	a	b	c	d	…	y	z
count	0	0	0	0	…	0	0

L = 8, R = 8

3. **Let T be a string whose characters are drawn from the alphabet S. We are given three strings α_1, α_2 and α_3. all of whose characters are also drawn from the same alphabet. Let P be the pattern obtained by concatenating the three strings in order, but with two '*' characters inserted between each α_i and α_{i+1} ($1 \leq i \leq 2$). The '*' character is called a wild card character and can match any character in the alphabet. Thus, pattern P is of the form α_1** α_2** α_3. The problem is to determine if P occurs in the text T. Provide a linear time algorithm for this problem.**

Solution:

This problem can be solved using the modified version of Z-algorithm. Suppose we have computed Z_i for all $1 < i \leq (k-1)$. Now we have to compute Z-value for the position k. Here, l = left boundary of Z_i and r = right boundary of Z_i. The changes required in the modified version are given below:

- For Case k>r, in Z-algorithm, we compare the characters starting at position k to the 1st character of the text and we keep comparing until a mismatch occurs. In this modified version, we will keep comparing the characters if the current characters match or one of the comparing characters is '*'.
- For Case $k \leq r$ and $Z_{k'} \leq \beta$, in Z-algorithm, we compare the characters starting at position r+1 of T to characters starting at position $|\beta|+1$ and keep comparing until a mismatch occurs. Here, instead of that, we will keep comparing the characters if the current characters match or one of the comparing characters is '*'. If none of the comparing characters is '*' or a mismatch is found, then we will end the comparison.

Suppose, $|T| = n$, $|\alpha_1| = m_1$, $|\alpha_2| = m_2$, $|\alpha_3| = m_3$

For position i [where $m_1 + m_2 + m_3 + 5 < i \leq m_1 + m_2 + m_3 + n + 5$], check Z-value Z_i. If Z_i is equal to $m_1 + m_2 + m_3 + 4$, then the pattern P exists. So, runtime of this algorithm is O(|P|+|T|), which is linear.

Example:

Suppose, α_1 = ab, α_2 = cd, α_3 = ef; pattern P = ab**cd**ef, new text T' = P$T = ab**cd**ef$mtabkkcdlgefabt

index	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
T'	a	b	*	*	c	d	*	*	e	f	$	m	t	a	b	k	k	c	d	l	g	e	f	a	b	t
Z-value	0	0	8	1	0	0	4	1	0	0	0	0	0	10	0	0	0	0	0	0	0	0	0	3	0	0

4. **For a string S of length n, recall that $sp_i(S)$ is defined to be the length of the longest proper suffix of S[1..i] that matches a prefix of S. Recall also that $sp_i'(S)$ is defined to be the length of the longest proper suffix of S[1..i] that matches a prefix of S and with the added condition that $S(i+1) \neq S(sp_i' + 1)$. For a string S, (a) given its sp_i values for all $1 \leq i \leq n$, provide an O(n) algorithm to compute its sp_i' values for all $1 \leq i \leq n$; (b) given its sp_i' values for all $1 \leq i \leq n$, provide an O(n) algorithm to compute its spi values for all $1 \leq i \leq n$**

Solution 4(a):

For a string S of length n, $sp_i(S)$ is defined to be the length of the longest proper suffix of S[1..i] that matches a prefix of S. Again, $sp_i'(S)$ is defined to be the length of the longest proper suffix of S[1..i] that matches a prefix of S when $S(i+1) \neq S(sp_i + 1)$. So, when $S(i+1) \neq S(sp_i + 1)$, $sp'_i = sp_i$. Otherwise, we have to look at the suffix-prefix (sp) value of substring $S[1,sp_i]$ as the sp value of that substring will represent the longest suffix value of S[1,i] such that $S(i+1) \neq S(sp_i' + 1)$.

Algorithm:

```
sp'_1 = 0
for k = 2 to n:
        v = sp_k
        if S(k+1) ≠ S(v+1) or v = 0 or k = n:  //if S(i+1) ≠ S(sp_i' + 1) or sp_i = 0 then sp'_i = sp_i
                sp'_k = sp_k
        else:
                sp'_k = sp_v   //otherwise assign the suffix-prefix (sp) value of substring S[1,sp_i] to sp'_i
        end if
end for
```

Runtime: As we can see from the above algorithm, if sp_i value is given then we can calculate sp'_l value using a single for loop across the string S, so required runtime is O(n).

Solution 4(b):

Here, all the sp'_i values are given. We have to figure out the sp_i values. Suppose we know all the sp values upto position i (sp_i). Then to calculate sp_{i+1}, we will check if S(i+1) matches S(sp_i + 1). If match is found then $sp_{i+1} = sp_i + 1$. Otherwise, we have to jump to the index position sp_i and consider the sp value of that position (assume v) as the new index for comparing.

Now, if S(v+2) ≠ S(i+2) then $sp_{i+1} = sp'_{i+1}$, [when S(i+1) ≠ S(sp_i + 1), $sp'_i = sp_i$]. Otherwise, check if character S(i+1) matches S(v+1), means suffix matches prefix. So, $sp_{i+1} = sp_{v+1} + 1$. Else, $sp_{i+1} = 0$ as there is no match between suffix and prefix.

Algorithm:

```
$sp_1$ = 0
for k = 1 to n-1:
        v = $sp_k$
        if S(k+1) = S(v+1):  //continue matching suffix to prefix
                $sp_{k+1}$ = $sp_k$ + 1
        else:
                v = $sp_v$
                if S(k+2) ≠ S(v+2) and k+2 < n:   // when S(i+1) ≠ S(spi + 1), sp'i = spi
                        $sp_{k+1}$ = $sp'_{k+1}$
                else:
                        if S(k+1) != S(v+1):   //no match between suffix and prefix
                                $sp_{k+1}$ = 0
                        else:
                                $sp_{k+1}$ = $sp_{v+1}$ + 1 //prefix of S(1,v+1) matches suffix of s(1,k+1)
                        end if
                end if
        end if
end for
```

Runtime: As we can see from the above algorithm, if sp'_i value is given then we can calculate sp_i value using a single for loop across the string S, so required runtime is O(n).

5. **(See spi(S) definition from above problem). You had written down a 11-bit password S on a piece of paper but have now lost the paper. However, you recall a few facts about the password S: that the first bit was a 1, and that sp_{11}(S)= 6, sp_6(S) = 3, and sp_2(S) = 0. Can you reconstruct the password S? Explain your reasoning.**

Solution:

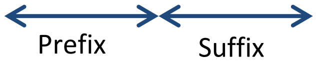
Here the 11-bit long password contains only 2 types of bits, 0 and 1. The very first bit is 1. The given sp_i values are, sp_{11}(S) = 6, sp_6(S) = 3, and sp_2(S) = 0. From the given information, it is possible to reconstruct the password. The step by step solution is given below.

Step1: 1^{st} bit is 1

i	1	2	3	4	5	6	7	8	9	10	11
S	1	*	*	*	*	*	*	*	*	*	*

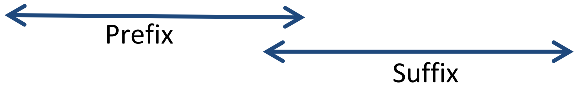
Step2: $sp_6(S) = 3$

i	1	2	3	4	5	6	7	8	9	10	11
S	1	*	*	1	*	*	*	*	*	*	*



Step3: $sp_{11}(S) = 6$

i	1	2	3	4	5	6	7	8	9	10	11
S	1	*	*	1	*	1	*	*	1	*	1

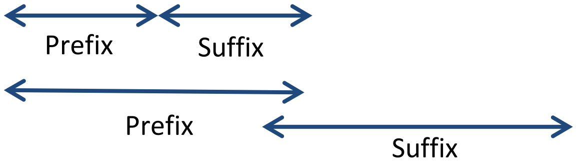


Step4: $sp_2(S) = 0$, means $S(1) \neq S(2)$. So, $S(2) = 0$

i	1	2	3	4	5	6	7	8	9	10	11
S	1	0	*	1	*	1	*	*	1	*	1

Step5: Merging all the sp_i values we get:

i	1	2	3	4	5	6	7	8	9	10	11
S	1	0	1	1	0	1	0	1	1	0	1



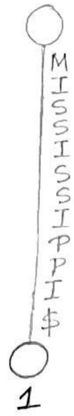
So, the password is, S = 10110101101

6. **(McCreight's suffix tree construction method). Let S be the string MISSISSIPPI$. Recall McCreight's suffix tree construction method; let STi denote the suffix tree containing the first i suffixes of S (i.e. all strings S[j,m], for 1 <= j <= i <= m, and where m is the length of S). Show the suffix tree STi at the end of each iteration i, for 1 <= i <= 12. Please be sure to include the suffix links as well.**

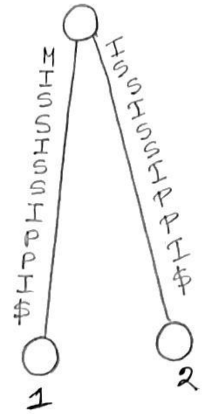
Solution:

Here, S = MISSISSIPPI$. The step by step construction of suffix tree STi (for each iteration i) along with suffix links is shown below:

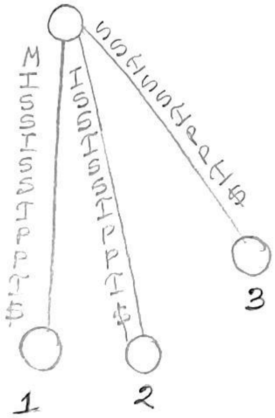
i=1:



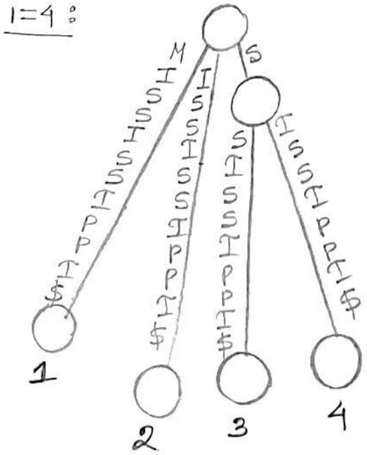
i=2:



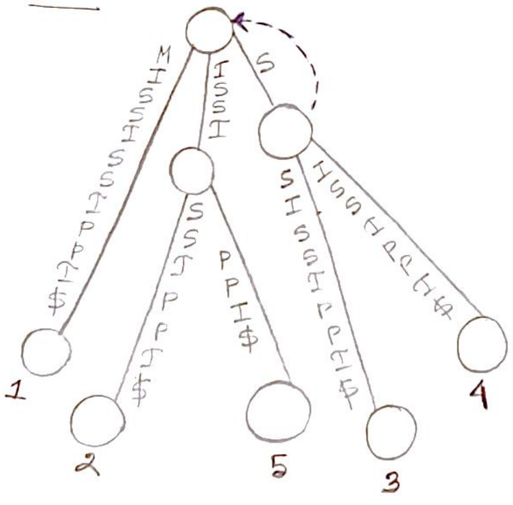
i=3:



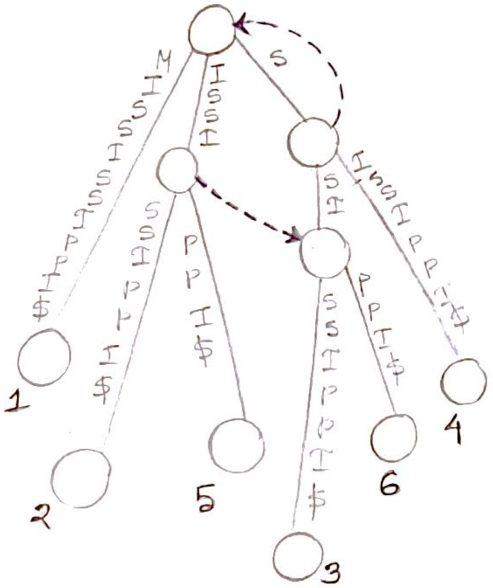
i=4:



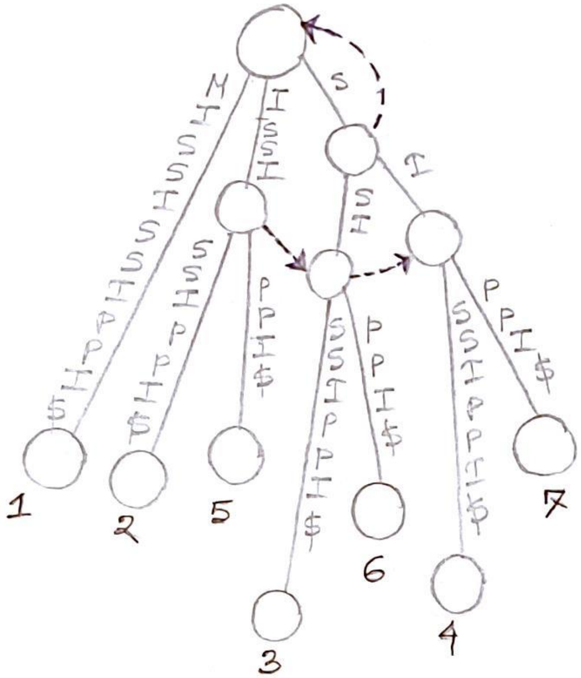
i=5:



i=6:

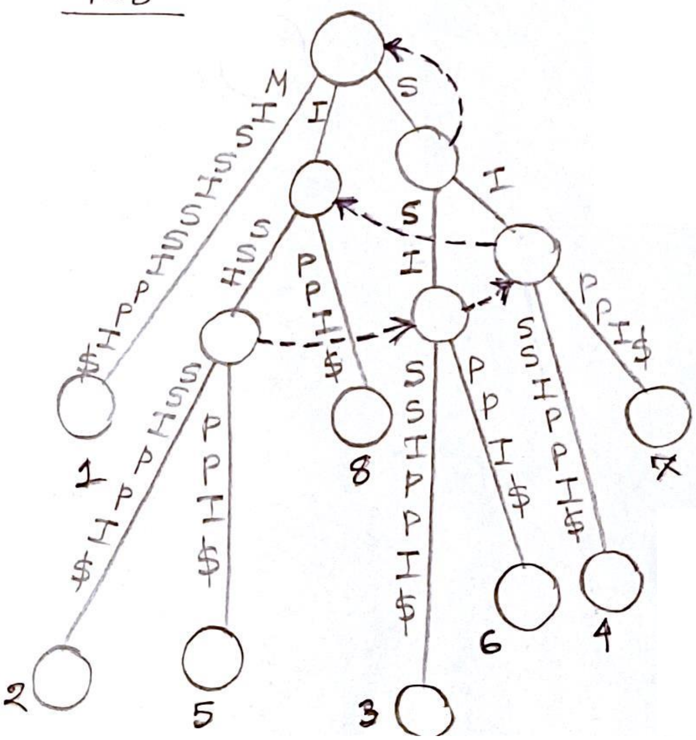


i=7:



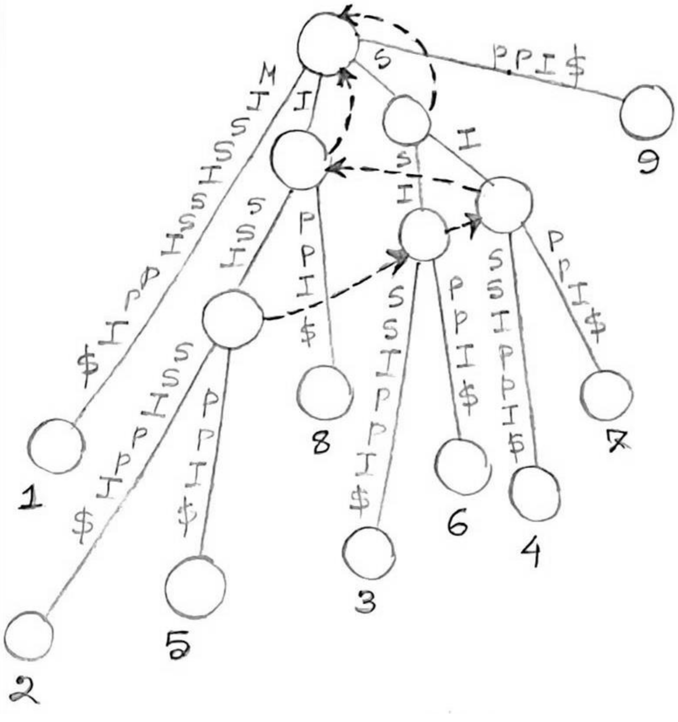
MISSISSIPPI$
ISSI
S
I
SSIPPI$
PPI$
SI
SSIPPI$
PPI$
SSIPPI$
PPI$
1
2
5
3
6
4
7

i=8



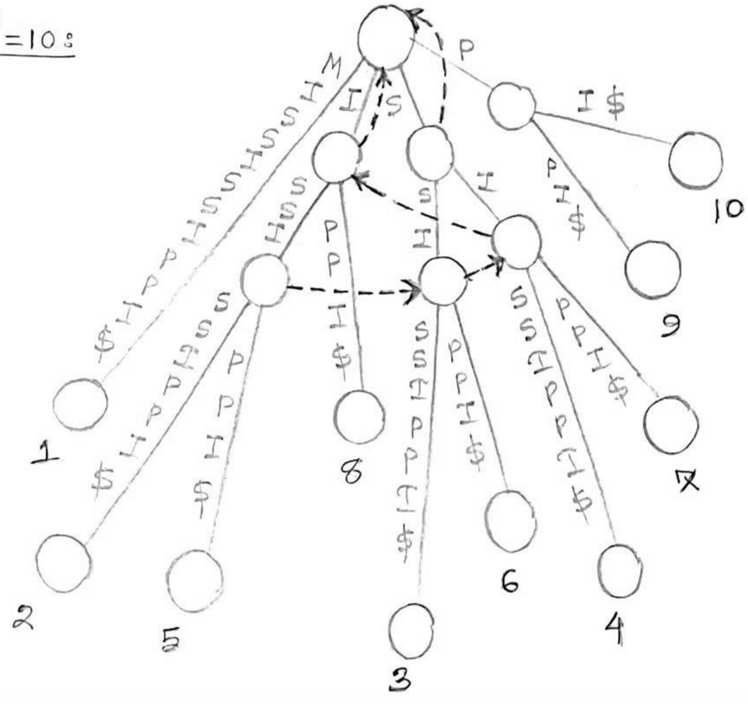
MISSISSIPPI$
I
S
SSI
PPI$
S
I
PPI$
SSIPPI$
PPI$
SSIPPI$
PPI$
SSIPPI$
PPI$
1
8
7
2
5
3
6
4

i=9:



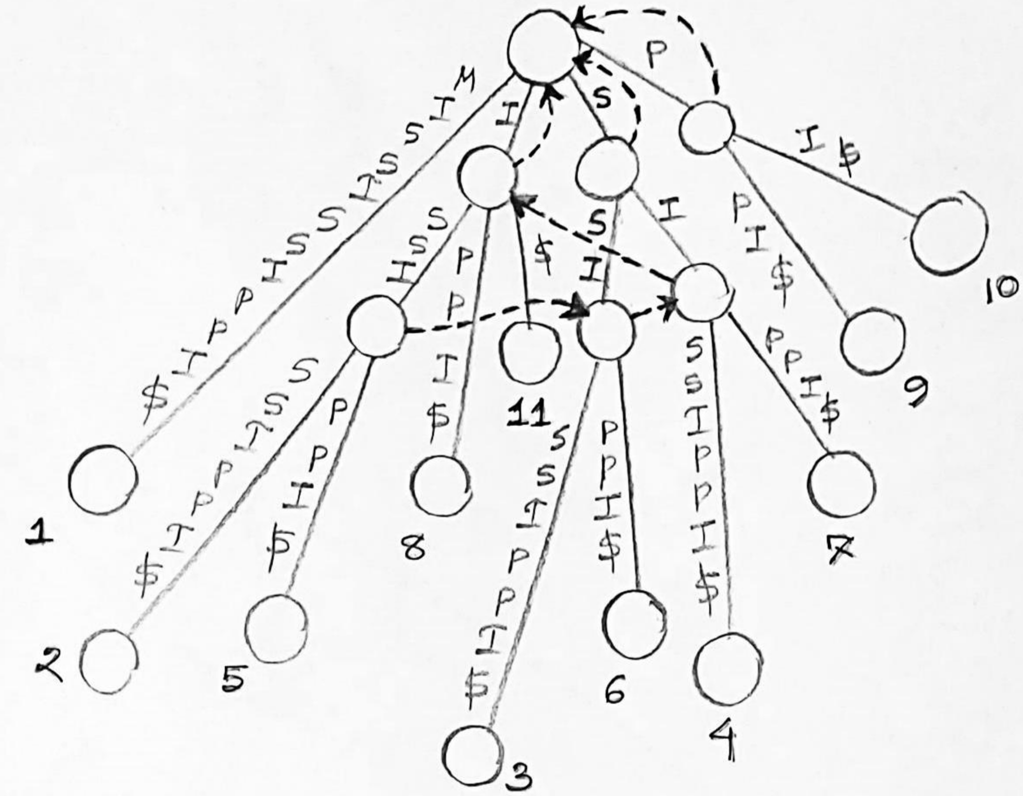
MISSISSIPPI$
I
S
PPI$
I
SSI
PPI$
S
I
SSIPPI$
PPI$
SSIPPI$
PPI$
SSIPPI$
PPI$
SSIPPI$
PPI$
9
1
8
7
2
5
3
6
4

i=10:

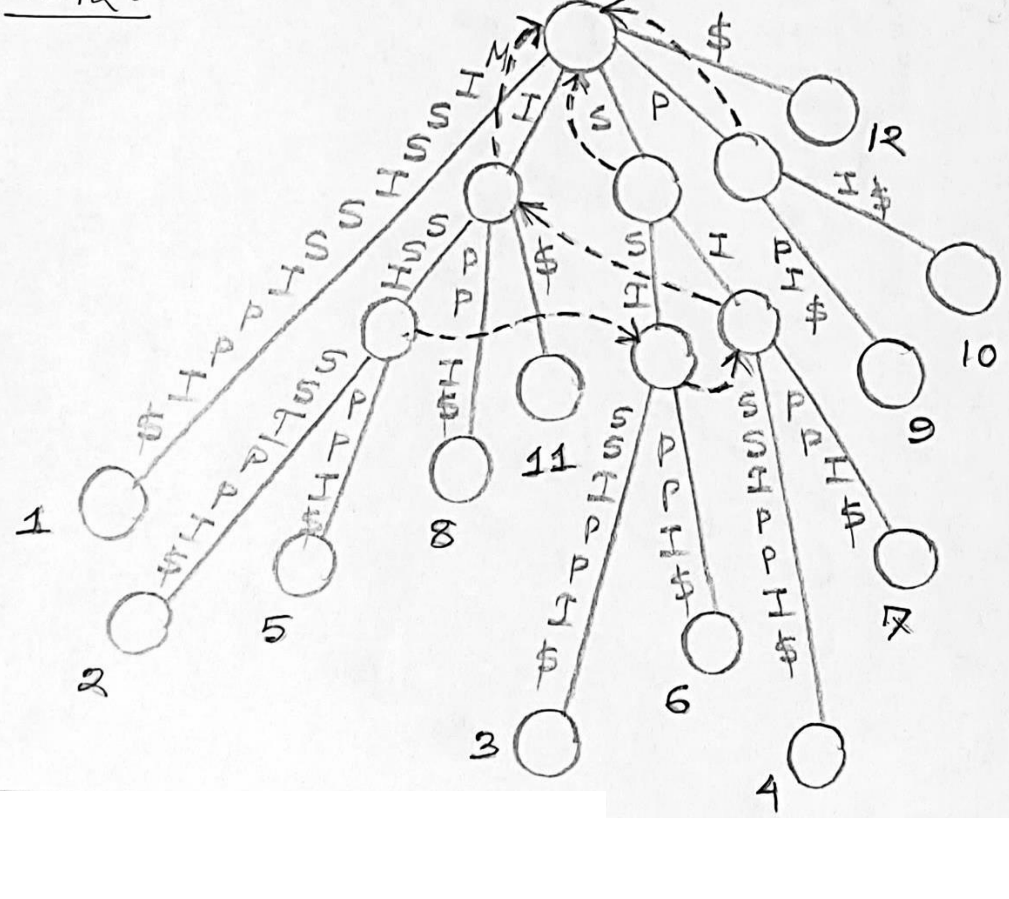


MISSISSIPPI$
I
S
P
I$
PI$
SSI
PPI$
S
I
I
SSIPPI$
PPI$
SSIPPI$
PPI$
SSIPPI$
PPI$
10
9
1
8
7
2
5
3
6
4

i=11:


MISSISSIPPI$
I
S
P
SSI
PPI$
$
I
SI
I$
PI$
SSIPPI$
PPI$
SSIPPI$
PPI$
SSIPPI$
PPI$
1
2
5
8
11
3
6
4
7
9
10

i=12:


MISSISSIPPI$
I
S
P
$
SSI
PPI$
$
SI
I
I$
PI$
SSIPPI$
PPI$
SSIPPI$
PPI$
SSIPPI$
PPI$
1
2
5
8
11
3
6
4
7
9
10
12