CAP 6515 ASSIGNMENT #1

1 Solution to Question No: 1

If P is a given pattern with length m and S is a given text with length n, KMP algorithm can find out all the occurrences of pattern P in string S in just linear amount of time.

1.1 Proof:

1.1.1 Best Case, O(n):

Case (i): Pattern P doesn't exist in String S. Example:
S: A T C U X D D R Y W
P: P Q O M N
In this case for loop will run n times. The value of k in the pseudocode will always remain 0 so it won't enter the while loop. Complexity O(n).

Case(ii): Pattern P and String S are exactly same. Example:
S: A B A B C A
P: A B A B C A
In this case outer for loop will run n times and while loop will not be executed as p[q+1]!=S[i] condition won't be true. Complexity O(n)

1.1.2 Worst Case, (2n):

Worst case can occur when there is mismatch after matches. Example:
S: AAAABAAABAAB
P: AAAA
This situation will trigger the while loop. KMP wont need to start comparing characters from the beginning of pattern P or postion i+1 of S as some of the characters of the next considered portion of S is already known as matched with prefix (due to prefix function). So, while loop won't run m times. In worst case, inner loop can in total decrease up to n times (at most). Outer loop runs n times. Complexity O(2n).

2 Solution to Question No: 2

2.1 Z-value Algorithm:

For a given text 'S' and a pattern 'P', Z-value algorithm finds out all the occurrences of pattern 'P'in text 'S'. To do this in linear time, it maintains an [L,R] interval which represents prefix substring. Here, Z[i] of Z array holds the length of longest substring that matches with the prefix starting from S[i] position. To find the pattern 'P' in 'S', we form a string PS (Merging P with S) and then calculate Z aaray for PS. If Z[i]¿=—P—, then a match of 'P'is found starting from S[i].

Problem: finding all occurrences of pattern 'P' in text 'S'
Input: String 'PS'
Output: Z-array
The pseudocode for the linear time Z-value algorithm is given below:

2.2 Complexity Analysis:

Here, the complexity is O($|PS|$). Complexity is linear because for each i, whenever there is a match, the value of R increases. We never compare any character that's position is less than R (within the frame). Again, whenever there is a mismatch, the iteration ends and value of R doesn't increase. Here, R $\leqslant |PS|$
For Case 1: O(Z[i]+1)   [No of Matches starting from position i + 1 mismatch]
For Case 2: O(1)
For Case 3: O(Z[i]-Z[j]+1)   [Z[j] holds length of already matched prefix. So, no of new matches with prefix+ 1 mismatch]

Algorithm 1 Z-Value Algorithm

```
p ← length[P]                                   ▷ Length of Pattern P
ps ← length[PS]                                 ▷ Length of Concatenated string PS (Pattern P + Text S)
L ← 0                                           ▷ Initializing left boundary
R ← 0                                           ▷ Initializing right boundary
Z[1] ← 0                                        ▷ First value of Z-array doesn't hold any significance
for i ← 2 to ps do                              ▷ Iterating through the whole ps array (except 1st position)
    if i > R then                               ▷ Case 1: the Left-Right Frame doesn't cover i
        L = R = i                               ▷ Forming new Left-Right frame for i which starts from position i
        while (R ⩽ ps) and (S[R] = S[R-L+1]) do
            R ++                                ▷ Increase the right boundary if matches the prefix
        Z[i] = R-L                              ▷ Z[i] is assigned the size of the frame
        R −−
    else if i ⩽ R then                          ▷ Case 2 or 3: the Left-Right Frame covers i
        j = i − L+1                             ▷ Equivalent position of the prefix for current position i
        if Z[j]+i ⩽ R then                      ▷ Case 2: length of matched sub-string with the prefix
                                                  starting from position i is within the current frame
            Z[i] = Z[j]
        else        ▷ Case 3: length of matched sub-string with the prefix starting from position i crosses the current frame
            L = i                               ▷ Forming new frame for i which starts from position i
            R++                                 ▷ Increase the right boundary if matches the prefix
            while (R ⩽ ps) and (S[R] = S[R-L+1]) do
                R ++
            Z[i] = R-L                          ▷ Z[i] is assigned the size of the frame
            R −−
return Z                                        ▷ Returns the Z-value array
```

2.3 Simulation for string S="aabcaabxaaz"

Initially,
Left, L =0
Right, R=0
Z[1] = 0

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0										

For i = 2 :
As i $>$ R, so Case 1.
Now, L = 2, R = 2
  S[1] = S[2] ; R=3
  S[2] != S[3]; break
Finally, Z[2] = 1, R = 2

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1									

For i = 3 :
As i $>$ R, so Case 1.
Now, L = 3, R = 3
  S[1] != S[3] ; break
Finally, Z[3] = 0, R = 2

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1	0								

For i = 4 :

As i > R, so Case 1.
Now, L = 4, R = 4
    S[1] != S[4] ; break
Finally, Z[4] = 0, R = 3

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1	0	0							

For i = 5 :
As i > R, so Case 1.
Now, L = 5, R = 5
    S[1] = S[5] ; R = 6
    S[2] = S[6] ; R = 7
    S[3] = S[7] ; R = 8
    S[4] != S[8] ; break
Finally, Z[5] = 3, R = 7

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1	0	0	3						

For i = 6 :
As i ⩽ R, so either Case 2 or Case 3.
Now, j = i - L + 1 = 6 - 5 + 1 = 2
    Z[j] + i = 1 + 6 = 7
As Z[j] + i ⩽ R, So Case 2.
    Z[i] = Z[j] = Z[2] = 1
Finally, Z[6] = 1, L = 5, R = 7

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1	0	0	3	1					

For i = 7 :
As i ⩽ R, so either Case 2 or Case 3.
Now, j = i - L + 1 = 7 - 5 + 1 = 3
    Z[j] + i = 0 + 7 = 7
As Z[j] + i ⩽ R, So Case 2.
    Z[i] = Z[j] = Z[3] = 0
Finally, Z[7] = 0, L = 5, R = 7

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1	0	0	3	1	0				

For i = 8 :
As i > R, so Case 1.
Now, L =8, R = 8
    S[1] != S[8] ; break
Finally, Z[8] = 0, R = 7

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1	0	0	3	1	0	0			

For i = 9 :
As i > R, so Case 1.
Now, L = 9, R = 9
    S[1] = S[9] ; R = 10

S[2] = S[10] ; R = 11
S[3] != S[11] ; break
Finally, Z[9] = 2, R = 10

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1	0	0	3	1	0	0	2		

For i = 10 :
As i $\leqslant$ R, so either Case 2 or Case 3.
Now, j = i - L + 1 = 10 - 9 + 1 = 2
Z[j] + i = 1 + 10 = 11
As Z[j] + i $\geq$ R, So Case 3.
L = i = 10, R = R + 1 = 11
S[2] != S[11] ; break
Finally, Z[10] = R - L = 1, R = 10

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1	0	0	3	1	0	0	2	1	

For i = 11 :
As i $>$ R, so Case 1.
Now, L = 11, R = 11
S[1] != S[11] ; break
Finally, Z[11] = 0, R = 10

Index :	1	2	3	4	5	6	7	8	9	10	11
S :	a	a	b	c	a	a	b	x	a	a	z
Z :	0	1	0	0	3	1	0	0	2	1	0

3 Solution to Question No: 3

3.1 Problem Statement:

Given a string P[1.....n], for each prefix P[1...i], we have to find out if it is a periodic string or not in linear time. A string is called periodic if it is a repetition of its sub-string.
Here, for each i(prefix P[1..i] of P) we want to know the largest k so that P[1...i] can be expressed as α^k for some sub-string α (period for each prefix). If the string is not periodic then k=1.

3.2 Algorithm Description:

The prefix-function of a string P[1...n] is an array Π of size n where Π[i] holds the length of the longest sub-string which matches the prefix of P[1...i]. IF a string is periodic, then a prefix of certain length must be repeated k times (k¿1) and no other characters can be present in that string at any position. Which means if the length of that periodic string is m and the length of prefix (α) that is repeated k times is l then value of Π[m] must be m-|α|.Example:

Index	1	2	3	4	5	6
Periodic String	A	B	A	B	A	B
$\pi[i]$	0	0	1	2	3	4

$\pi[6]$ = l - |AB| = 6 − 2 = 4

To check if a string of length n is periodic or not, if we subtract Π[n] from n then that will give us the possible length of α (period). If the length of the whole string (n) is divisible by |α| then it means the string is only composed of repetitions of α. Here if k represents the maximum number of times α is repeated in the string for the minimum possible value of α then, k = n / (n - Π[n]).
So, Fist we need to calculate the prefix-function for the string P. Then we have to execute a loop starting from 1 up-to its

length to consider each possible prefix P[1...i]. If we do the above mentioned calculation for each possible prefix inside the loop then we can find out which prefixes of are Periodic along with α and k.
Here first pseudo-code [Algorithm 1] computes the prefix function (Π) and second pseudo-code [Algorithm 2] determines periodic string for all n prefixes of string P:

Algorithm 1 Compute Prefix-Function(P)

```
n ← length[P]
Π[1] ← 0
k ← 0
for q ← 2 to n do
    while k > 0 and P[k+1] ! = P[q] do
        k ← Π[k]
    if P[k+1] = P[q] then
        k ← k+1
    Π[q] ← k
return Π
```

Algorithm 2 Periodic Strings Algorithm

```
n ← length[P]
Π ← Compute-Prefix-Function(P)
for i ← 1 to n do
    PeriodSize = i −Π[i]
    k ← 1
    if (i % PeriodSize) = 0 then
        k = i / PeriodSize
        print k                          ▷ Maximum possible value of k
        print P[1...PeriodSize]          ▷ Minimum possible α
    else
        print "Not Periodic String"
```

3.3 Correctness Proof:

3.3.1 Case 1

Trivial Case, Prefix P[1...1]:
$\Pi[1] = 0$, PeriodSize = 1; k = 1, which is correct.

3.3.2 Case 2:

Prefix P[1...i] where i != 1 and P[1...i] not a Period String:
Example i) ABCABCD, Π[i] = 0, PeriodSize = 1; k = 1, which is correct.
Example ii) ABCDDABC, Π[i] = 3, PeriodSize = 5;
(i % PeriodSize) != 0; So, k = 1, which is correct.

3.3.3 Case 3:

Prefix P[1...i] where i != 1 and P[1...i] a Period String:
i) Only 1 possible value of k ($k > 1$):
Example: ABCABC , Π[i] = 3, PeriodSize = 3; k = 2, which is correct.
ii) More than 1 possible value of k ($k > 1$):
ABABABAB, Π[i] = 6, PeriodSize = 2; k = 4, which is correct. Prefix P[1...i] where i != 1 and P[1...i] a Period String:

3.4 Time Analysis:

For a string P, the time complexity for the calculation of prefix-function is $O(|P|)$. Then to consider all the possible prefixes of string P, a loop need to run $|P|$ times. So, total run-time for the proposed algorithm is $O(|P|+|P|)$ or $O(2|P|)$ which is linear. to the length of the string.