EC31002- Digital Communication

Assignment-1

Name: *Rahul Wadhwa*

Roll No.: *19EC30034*

Compressing of a Text file using Ziv-Lempel

Theory:-

LZ-77 is a lossless data compression Algorithm widely spread in our current systems since, for instance, ZIP and GZIP are based on LZ77.

It is a sliding window algorithm.
LZ77 iterates sequentially through the input string and stores any new match into a search buffer. The process of compression can be divided in 3 steps:

1. Find the longest match of a string that starts at the current position with a pattern available in the search buffer.
2. Output a triple (n, u, c) where,

- *u*: offset, represents the number of positions that we would need to move backwards in order to find the start of the matching string.
- *n*: length, represents the length of the largest match.
- *c*: character, represents the character that is found after the match.

3. Move the cursor n+1 positions to the right.

So this algorithm is mainly based on string matching. If we go by brute force then time complexity is much worse. It is O(n1*n2) where n1=length of pattern which we are searching and n2=length of string in which we are searching string. The fact that makes it more worse is that we don’t we know n1 since we are trying to find the largest match and we have no idea what could be the largest match.

Optimization:-

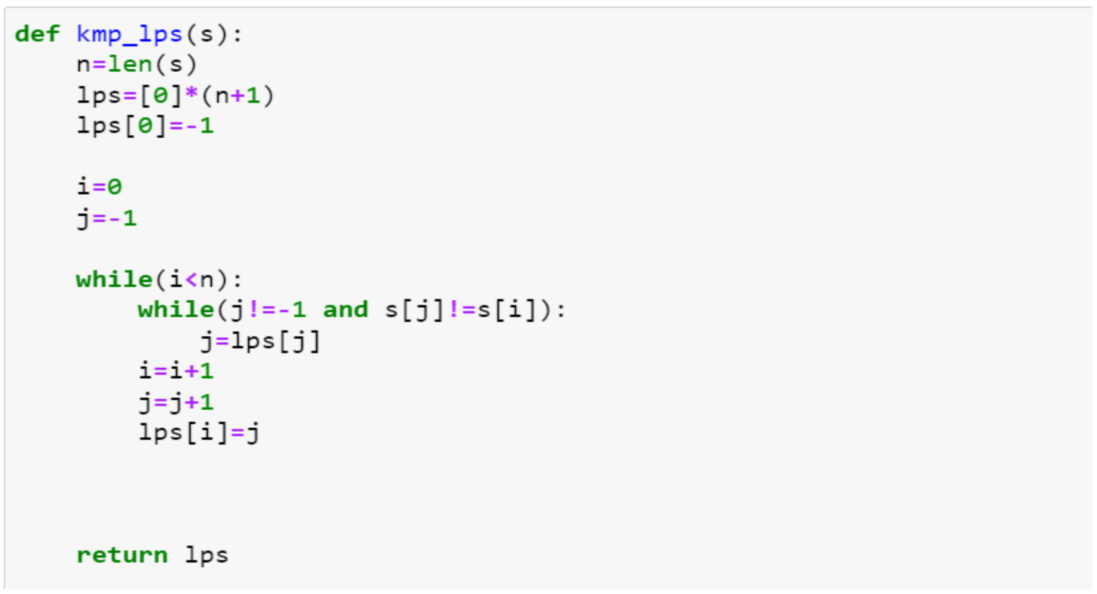
A very efficient way of finding pattern matches is **Knuth morris part Algorithm (KMP)** . The KMP matching algorithm uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern) of the pattern and improves the worst case complexity to O(n). The basic idea behind KMP’s algorithm is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this information to avoid matching the characters that we know will anyway match.

We use an array to store the length of **largest proper prefix** that is also a suffix. Suppose the name of array is lps[ ].

Lps[i] stores the length of largest proper-prefix of str[0,1…..i-1] that is also a suufix in this string.

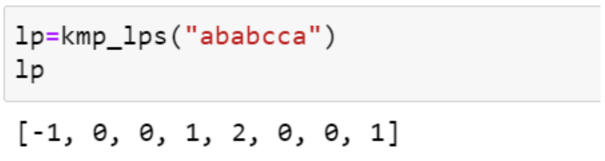
Lps[0]=-1 by default.

This is the algorithm for calculating lps array



```
def kmp_lps(s):
    n=len(s)
    lps=[0]*(n+1)
    lps[0]=-1

    i=0
    j=-1

    while(i<n):
        while(j!=-1 and s[j]!=s[i]):
            j=lps[j]
        i=i+1
        j=j+1
        lps[i]=j



    return lps
```

Eg. For lps array



```
lp=kmp_lps("ababcca")
lp
```

```
[-1, 0, 0, 1, 2, 0, 0, 1]
```

Now we use this lps array to find the largest match through this way.

We first find a character not present in our pattern and string in which we are searching. (For our assignment one of such characters is ‘#’).

T=pattern

S=string in which we are finding pattern

We generate another string

S1=T+’#’+S

And generate its lps array since ‘#’ is present nowhere the maximum prefix that is also a suffix could be T only. So we start iterating through lps values from indices starting from S and find the largest lps[i] and that will give us the length of largest match of T in S. After that we encode (lps[i], offset, charcter found after match).

In LZ-77 since we are finding lowest offset we can start iterating lps array from end till string S is there.

Since in LZ-77 algorithm there can be an overlap between pattern and window.

Suppose window length=w and our pointer is currently at p. Since pattern length can be variable and we can start from any value of n but for practical purposes pattern length is of order of w so for optimization i am fixing maximum n=5*w;

So, T=str[p,p+1,….p+n-1]

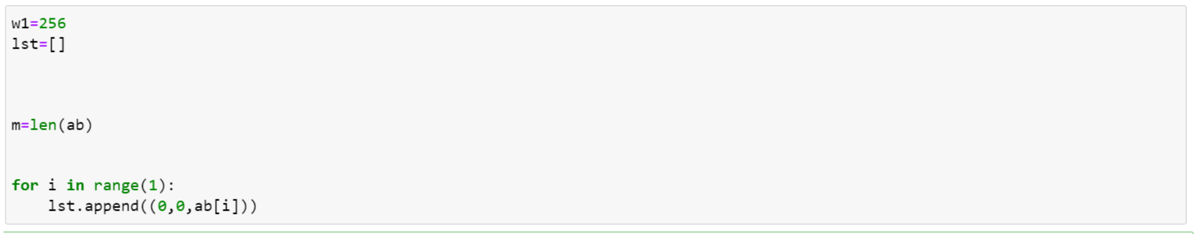
S=str[p-w,p-w+1,……p+n-2] (since we can find matching of pattern with some substring of n also but starting point of match has to lie in window).

Another optimization:-

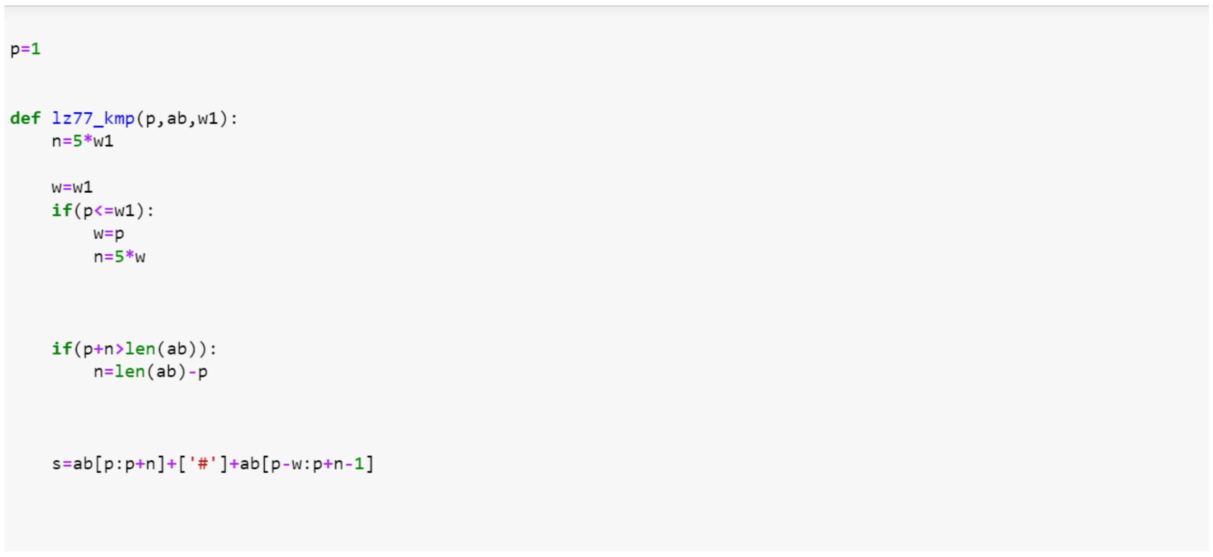
In LZ-77 window length is fixed. So suppose if w=512 then first w characters will be encoded using fixed length encoding. We can improve it by starting from w=1 and we will increase our window length till the point we reach w=512. Through this way we will get even more compression.

Let our text file is stored in string ab.

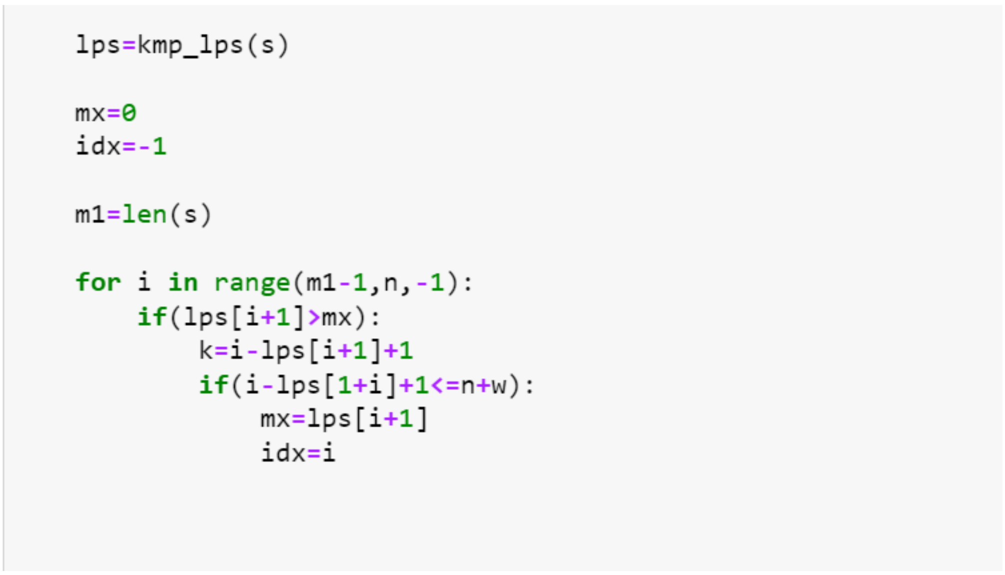
Here is the code of LZ-77



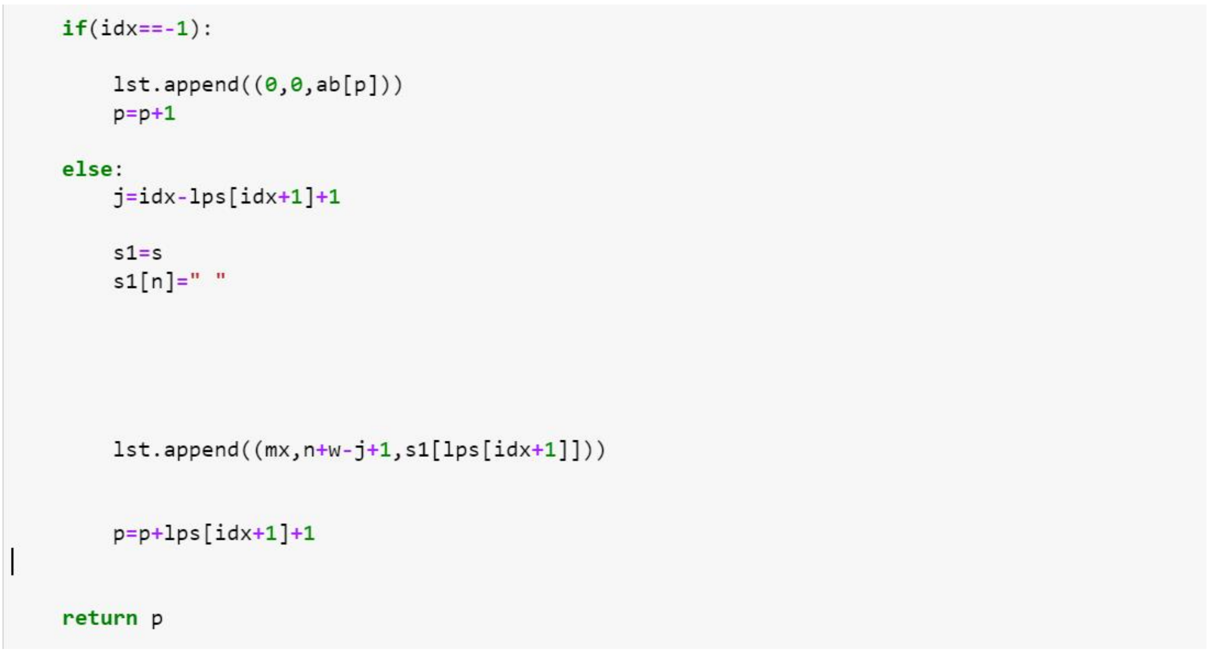
```
w1=256
lst=[]


m=len(ab)


for i in range(1):
    lst.append((0,0,ab[i]))
```



```
p=1


def lz77_kmp(p,ab,w1):
    n=5*w1

    w=w1
    if(p<=w1):
        w=p
        n=5*w



    if(p+n>len(ab)):
        n=len(ab)-p



    s=ab[p:p+n]+['#']+ab[p-w:p+n-1]
```

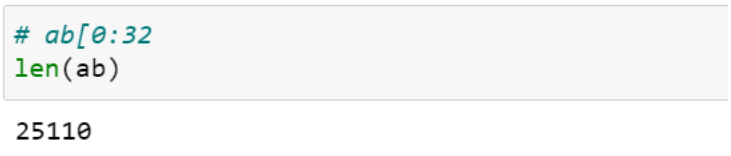


```
    lps=kmp_lps(s)

    mx=0
    idx=-1

    m1=len(s)

    for i in range(m1-1,n,-1):
        if(lps[i+1]>mx):
            k=i-lps[i+1]+1
            if(i-lps[1+i]+1<=n+w):
                mx=lps[i+1]
                idx=i
```



```
    if(idx==-1):

        lst.append((0,0,ab[p]))
        p=p+1

    else:
        j=idx-lps[idx+1]+1

        s1=s
        s1[n]=" "



        lst.append((mx,n+w-j+1,s1[lps[idx+1]]))


        p=p+lps[idx+1]+1

    return p
```

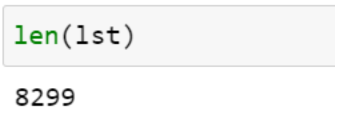
Our text file contains total **25110** characters



```
# ab[0:32
len(ab)
```

```
25110
```

For w=256 we get dictionary size=8299



```
len(lst)
```

```
8299
```

So compression ratio for w=256 is 0.33

Now we need to design a decoder that if given this dictionary as input we must get back our original text file.

Here is the algorithm of decoder

```
string=""

for i in lst:

    if(i[0]==0):
        string+=i[2]
    else:
        m=len(string)

        if(i[0]<i[1]):
            s1=string[m-i[1]:m-i[1]+i[0]]

            string+=s1
            string+=i[2]

        else:
            cnt=0
            while(cnt<i[0]):
                m=len(string)



                string+=string[m-i[1]]

                cnt+=1
            string+=i[2]
```

And here is the link of decoded string and dictionary

LZ-77

And here is the link of python notebook containing code and results

Code