**Lab 8: String Matching**

**Please finish Lab 7 practice if you haven’t done it because of the in-class test.**

**Task 1 Naïve shifting algorithm**

The idea is very simple. First, we have the pattern to search in a text, and have two pointers i for the text and j for the pattern. We increase i one per round, and each round we compare each element in the pattern to its counterpart text[i+j]. This process can be executed by the following pseudocode. Note that we’ll count all the occurrences of the pattern.

def naiveSeaching(pattern, text):

lp = length of pattern

lt = length of text

set i, j and count 0

for i in range(lt-lp+1) # the last few elements of the text cannot be matched anyway

found = True # indicate the match

foreach j in lp:

if text[i+j] ! = pattern[j]: # mismatch

found = False

break

if found:

print ("Found pattern at index " + str(i))

count+=1

return count

Of course, you can use WHILE loops to check the match as well, but they are similar anyway. Then you can use the following code to test your function

text = "ABABDABACDABABCABAB"

pattern = "ABA"

print(naiveSearching(pattern, text))

The successful program should return 4 occurrences and their positions.

Now, let’s try a bigger example. Please download the binary.txt file from BB and use the following code to try again.

f = open('binary.txt', 'r')

text = f.read()

pattern = '10110101'

start = time.time()

print(naiveSearching(pattern, text))

end = time.time()

print("The elapsed time is " + str(end-start))

**Task 2 KMP pattern search**

KMP is one of the more efficient algorithms than the naïve searching by avoiding overlapping comparisons. However, it needs to pre-process the pattern to create a LPS array, and then use it to do the pattern search.

First, you should be able to work out the LPS array for a short pattern manually. For example, for the pattern “ABA”, you need to find the LPS of ‘A’, ‘AB’ and ‘ABA’, and use the length of the LPS to fill in the LPS array as ‘001’.

If you want to challenge your programming skill, please go to the lecture slides for the pseudocode for LPS. It’s straightforward to translate it into a Python code. Or simple compare the pseudocode with the following code.

def calculateLPSArray(pattern):

lp = len(pattern)

lps = [None]\*lp

i = 0

j = 1

lps[0] = 0

while lps[lp-1] == None:

if pattern[i] == pattern[j]:

lps[j] = i+1

i += 1

j += 1

else:

if i==0:

lps[j] = 0

j += 1

else:

i = lps[i-1]

return lps

Please use ‘ABA’ or other patterns to test this function before you start to carry on. For the pattern search of KMP, there are many different implementations in fact. If you google the Internet, there are quite a few different versions. The following one is an example.

def KMPSearch(pattern, text):

lp = len(pattern)

lt = len(text)

count = 0

lps = calculateLPSArray(pattern)

i=0

j=0

while i < lt: # iterate the pointer for the text

if pattern[j] == text[i]: # if single elements from text and pattern are matched

i += 1

j += 1

if j == lp: # if the pointer is the length of pattern, the pattern is fully matched

# note that j is out of the range of pattern because the pointer starts from 0

print ("Found pattern at index " + str(i-j))

count += 1

j = lps[j-1] # j is reset by the last value in the LPS array

else: #if mismatch

if j == 0: # first element from pattern doesn’t match, i is increased by one

i += 1

else:

j=lps[j-1] # or j is reset by the value of j-1 in the LPS array

return count

Now, finish task 2, and use the simple case in Task 1 to test your program. If it returns the sample result with the naïve search, you can try the ‘binary.txt’. Is it quicker than the naïve search?