

"""

Given a text and a pattern, want a function search that prints all occurrences of pattern

in text

The naive string matching algorithm slides the pattern one by one. After each slide, it one

by one checks the characters at the current shift and if all chars match them print the match

"""

def naiveSearch(haystack, needle):

"""Brute force Algorithm to search strings

    Parameters:

    haystack (str): string to be searched

    needle (str): string to be found

    Returns:

    int: index where matched pattern begins in haystack or -1

    """

    h,n = len(haystack), len(needle)

    #only consider possible starting points

    for i in range(h - n + 1):

        k = 0

        while k < n and haystack[i + k] == needle[k]:

            k += 1

        if k == n:

            #end of needle was reached

            return i

    #no match was found

    return -1

"""

The Robin Karp algorithm looks for a substring by computing a rolling hash function,

whereby q is a prime. We would like that q^n-1 is not too large, n is limiting here, so we want

a small prime: one option is to use the largest prime q such that w is the word size and A

=> The largest character value (UTF-8, 256=^8)

=> The value of q is important to ensure that the hash value is still a full word

=> Contraint: want the 2^w to at least be larger than a multiple of the prime^n-1\*A

=> Time complexity is O(m+n), but in the worst case, it is O(mn)

=> collisions are known to be spurious hits

"""

#================================= Example 1 ===================================#

#Souce: https://www.geeksforgeeks.org/python-program-for-rabin-karp-algorithm-for-pattern-searching/

# Following program is the python implementation of

# Rabin Karp Algorithm given in CLRS book

# d is the number of characters in the input alphabet

d = 256

# pat -> pattern

# txt -> text

# q -> A prime number

def search(pat, txt, q):

M = len(pat)

N = len(txt)

i = 0

j = 0

p = 0 # hash value for pattern

t = 0 # hash value for txt

h = 1

# The value of h would be "pow(d, M-1)%q"

for i in range(M-1):

h = (h\*d)%q # want to implement a unique hashvalue of the first window of text

# Calculate the hash value of pattern and first window

# of text

for i in range(M):

p = (d\*p + ord(pat[i]))%q

t = (d\*t + ord(txt[i]))%q #(no of characters in the alphabet \* len of text + ord (txt[i]))/ q (which is the prime number)

# Slide the pattern over text one by one

for i in range(N-M+1): #There are N-M+1 positions at which the string can occur

# Check the hash values of current window of text and pattern if the hash values match then only check for characters one by one

if p==t:

# Check for characters one by one

for j in range(M):

if txt[i+j] != pat[j]:

break

j+=1

# if p == t and pat[0...M-1] = txt[i, i+1, ...i+M-1]

if j==M:

print "Pattern found at index " + str(i)

# Calculate hash value for next window of text: Remove

# leading digit, add trailing digit

if i < N-M:

t = (d\*(t-ord(txt[i])\*h) + ord(txt[i+M]))%q

# We might get negative values of t, converting it to

# positive

if t < 0:

t = t+q

# Driver program to test the above function, pls specify

txt = ""

pat = ""

q = 101 # A prime number

search(pat,txt,q)

#================================= Example 2 ===================================#

"""

Haven't read through yet

"""

def rk\_search(string,pat,lconst): #lconst is the large constant used to limit the maximum hash value

string = string.upper()

pat = pat.upper()

#ASSUMING ALL CHARACTERS ARE UPPPER\_CASE,

#Can be extended for lower case if necessary

l = len(string)

l\_p = len(pat)

con = 26 #The constant for base system 26

hashval = 0 #For the pattern

currhash = 0 #For each substring

for i in range(l\_p):

hashval += ((ord(pat[i])-ord('A')+1)\*(con\*\*(l\_p-i-1)))%lconst

currhash += ((ord(string[i])-ord('A')+1)\*(con\*\*(l\_p-i-1)))%lconst

for ind in range(l-l\_p+1):

if ind!=0:

currhash = (con\*(currhash-((ord(string[ind-1])-ord('A')+1)\*(con\*\*(l\_p-1))))+((ord(string[ind+l\_p-1])-ord('A')+1))%lconst)

if(currhash==hashval):

i,j = 1,ind+1

while(i<l\_p):

if string[j]!=pat[i]:

break

i += 1

j += 1

else:

print "Found at index",ind

#=====================================================================#

"""

Boyer-Moore

"""

def Boyer\_Moore():

m=len(pattern)

n=len(string)

i=0

j=0

while i<n-m:

j=m-1

while string[i+j]==pattern[j]:

j-=1

if j<0:

return i

break

i+=1

return -1

"""

The Knuth-Morris-Pratt string search algorithm

TheKMP mathcing algorithm uses degenerating priperty (pattern having same sub-patterns appearing

more than once in the pattern) of the pattern that improves the worst case scenario

complexity of O(n)

Irrelevant in our case

"""

#https://www.geeksforgeeks.org/kmp-algorithm-for-pattern-searching/

# Python program for KMP Algorithm

def KMPSearch(pat, txt):

M = len(pat)

N = len(txt)

# create lps[] that will hold the longest prefix suffix

# values for pattern

lps = [0]\*M

j = 0 # index for pat[]

# Preprocess the pattern (calculate lps[] array)

computeLPSArray(pat, M, lps)

i = 0 # index for txt[]

while i < N:

if pat[j] == txt[i]:

i += 1

j += 1

if j == M:

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

j = lps[j-1]

# mismatch after j matches

elif i < N and pat[j] != txt[i]:

# Do not match lps[0..lps[j-1]] characters,

# they will match anyway

if j != 0:

j = lps[j-1]

else:

i += 1

def computeLPSArray(pat, M, lps):

len = 0 # length of the previous longest prefix suffix

lps[0] # lps[0] is always 0

i = 1

# the loop calculates lps[i] for i = 1 to M-1

while i < M:

if pat[i]== pat[len]:

len += 1

lps[i] = len

i += 1

else:

# This is tricky. Consider the example.

# AAACAAAA and i = 7. The idea is similar

# to search step.

if len != 0:

len = lps[len-1]

# Also, note that we do not increment i here

else:

lps[i] = 0

i += 1

txt = "ABABDABACDABABCABAB"

pat = "ABABCABAB"

KMPSearch(pat, txt)