string

Thursday, 3 August 2023 6:32 AM

① Naive Pattern Searching

Naive : $O((n-m+1)*m)$	No preprocessing	$m \to$ pattern length, $n \to$ text length, $m \le n$
Naive when all characters of pattern are distinct : $O(n)$		
Rabin Karp : $O((n-m+1)*m)$ (But better than naive on average)	Preprocessing pattern	
*KMP : $O(n)$		
Suffix Tree : $O(n)$	Preprocess text	

Pattern Searching in Python

```
text = input("Enter text")
pat = input("Enter pattern")
pos = text.find(pat)
while pos>=0:
    print(pos)
    pos = text.find(pat, pos+1)
```

Naive Pattern Searching

```
def naive_fnt(text,pat):
    n = len(text)
    m = len(pat)
    for i in range(n-m+1):
        j=0
        while j<m:
            if pat[j] != text[i+j]:
                break
            j=j+1
        if j==m:
            print(i,end=" ")
```

$O((n-m+1)*m)$

Improved Naive for Distinct in Pattern

→ If we see a mismatch after j matches, then we increment i by j

```
def ImpNaiveSearch(text,pat):
    m,n= len(pat), len(text)
    i=0
    while i<=(n-m):
        for j in range(m):
            if (pat[j] != text[i+j]):
                break
        if j==m:
            print(i,end=" ")
        if j==0:
            i+=1
        else:
            i+=j
```

Time : $O(n-m+1)$

② Rabin Karp Algorithm

① Like Naive algorithm, slide the pattern one by one.

② Compare hash values of pattern and current text window. If hash values match, then only compare individual characters.

Rolling Hash :

$$t_{i+1} = t_i + text[i+m] - text[i]$$

M : length of pattern

Improved Hash :

$d=5$

Rolling : $t_{i+1} = d(t_i - text[i]*d^{M-1}) + text[i+m]$

$m \to$ length of pattern

Example

a : 1
b : 2
c : 3
d : 4
e : 5

text = " a b d a b c b a b c "

$t_0 = 1*5^2 + 2*5 + 4*1 = 39$

$t_1 = 5*(t_0 - 1*5^2) + 1 = 71$

$t_2 = 5*(t_1 - 2*5^2) + 2 = 107$

$t_3 = 5*(t_2 - 4*5^2 + 3 = 88$

. . . . .

pattern = "abc"

$P = 1*d^2 + 2*d^1 + 3*d^0 = 38$

```
d=256
def RKSearch(pat,text,q):
    M,N= len(pat),len(text)
    h=1
    for i in range(1,M):
        h=(h*d)%q
    P,t =0,0
    for i in range(M):
        P=(P*d+ord(pat[i]))%q
        t=(t*d+ord(text[i]))%q
    for i in range(N-M+1):
        if P==t:
            for j in range(M):
                if text[i+j] != pat[j]:
                    break
            else:
                print(i,end=" ")
        if i<(N-M):
            t=((d*(t-ord(text[i])*h))+ord(text[i+M]))%q
        if t<0:
            t=t+q
```

compute $(d^{M-1})\%q$

compute P & t

check for spurious hit

compute t_{i+1} using t_i

Time complexity

Worst case
text = "AAAAAA"
Pat = "AAA"
$O((n-m+1)*m)$

③ KMP Algorithm

Constructing Longest Proper Prefix Suffix Array

proper prefix of 'abcd'	lps = ch	lps : lps[]
"", "a", "ab", "abc"	"abcbc"	[0,0,1,2,0]
suffix of 'abcd'	"aaaa"	[0,1,2,3]
"", "d", "cd", "bcd", "abcd"	"abcd"	[0,0,0,0]
	"ababab"	[0,0,1,2,3,4]

str = " aba c ab ad "
lps[] = {0, 0, 0, 1, 0, 1, 2, 3, 0}

str = " abbabb "
lps[] = {0, 0, 0, 1, 2, 3}

Naive $O(n^3)$

```
def longestProperPrefixSuffix(str,n):
    for len in range(n-1,-1,-1):
        for j in range(len):
            if str[j] !=str[n-len+j]:
                break
            else:
                return len
    return 0
def fillLPS(str,lps):
    lps[0] =0
    for i in range(1,len(str)):
        lps[i]=longestProperPrefixSuffix(str,i+1)
```

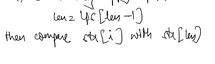
Efficient

Basic Idea

① If $len = lps[i-1]$ and $str[len]$ and $str[i]$ are same, then $lps[i] = len+1$

② If $str[i]$ and $str[len]$ are not same:

ⓐ if $len == 0$



$lps[i] = 0$

ⓑ Else, we recursively apply $lps[]$
$len = lps[len-1]$
then compare $str[i]$ with $str[len]$

```
def fillLPS(str,lps):
    lps[0]=0
    i=1
    len=0
    while i<n:
        if str[i]==str[len]:
            len+=1
            lps[i]=len
            i+=1
        else:
            if len==0:
                lps[i]=0
                i+=1
            else:
                len=lps[len-1]
```

$O(n)$

KMP String Matching

Ip : text = "abcdefg" → n = 7
pat = "ncd" → m = 3

Naive	Rabin Karp
$O((n-m)*m)$ Time	$O((n-m)*m)$ time
$O(1)$ space	$O(1)$ space

Naive for distinct characters	KMP
$O(n)$ time	$O(n)$ time
$O(1)$ space	$O(m)$ space

```
void KMP(pat,text):
    n= len(text)
    m= len(pat)
    lps = [0]*M
    fillLPS(pat,lps)
    i=j=0
    while i<N:
        if pat[j]==text[i]:
            i+=1
            j+=1
        if j==m:
            print(i-j)
            j= lps[j-1]
        elif i<N and pat[j] !=text[i]:
            if j==0:
                i+=1
            else:
                j= lps[j-1]
```

④ Longest Substring With Distinct Characters

Naive $O(n^3)$

```
def areDistinct(str,i,j):
    visited = [0]*256
    for k in range(i,j+1):
        if visited[ord(str[k])]==True:
            return False
        visited[ord(str[k])] = True
    return True

def longestDistinct(str):
    n= len(str)
    res=0
    for i in range(n):
        for j in range(i,n):
            if areDistinct(str,i,j):
                res= max(res,j-i+1)
    return res
```

Better $O(n^2)$

```
def longestDistinct(str):
    n=len(str)
    res=0
    for i in range(n):
        visited = [0]*256
        for j in range(i,n):
            if visited[ord(str[j])]==True:
                break
            else:
                res = max(res,j-i+1)
                visited[ord(str[j])]= True
    return res
```

Efficient $O(n)$

maxEnd(j) : length of longest substring that has distinct characters and ends with j

$$res = \max_{j=0}^{n-1} (maxEnd(j))$$

$$maxEnd(j) = \begin{cases} maxEnd(j-1)+1 & \text{; if str[j] is not present in maxEnd(j-1)} \\ j - prev(str[j]) + 1 & \text{; where prev(str[j]) is previous index of character str[j]} \end{cases}$$

```
def longestDistinct(str):
    n= len(str)
    res=0
    prev= [-1]*256
    i=0
    for j in range(n):
        i = max(i, prev[ord(str[j])]+1)
        maxEnd = j-i+1
        res = max(res, maxEnd)
        prev[ord(str[j])] = j
    return res
```