Finding substrings

BY Taariq Mowzer

What are we doing?

How many times does a string *L* appear in a string *S.*

E.G How many times does AABA appear in ABABAABAABA.

In this case twice:

ABAB|AABA|ABA

ABABAAB|AABA|

Notice the overlap ABAB|AAB|A|ABA|

Rabin-Karp and hashing

How to hash a string S?

Let p = 1000000007 and k = 3247683247 (some other big prime)

Let f(char v) = the position v is in the alphabet e.g f(a) = 1, f(e) = 5.

hash (‘adeb’) = f(‘a’)*k^3 + f(‘d’)*k^2 + f(‘e’)*k + f(‘b’) (mod p)

What’s the point of hashing?

If hash(P) ≠ hash(Q) then $P \neq Q$

That means we can check less cases.

Notice that if hash(P) = hash(Q) does not mean $P = Q$, so you still have to check if $P = Q$.

Rolling hash

Suppose we're hashing length $n = 4$.

S = 'abbaaccd'

hash('abba') = $1k^3 + 2k^2 + 2k + 1$ ALL mod p

hash('bbaa') = $2k^3 + 2k^2 + 1k + 1$

hash('baac') = $2k^3 + 1k^2 + 1k + 3$

To go from one hash to another: Remove the first letter
times k
add the next letter

Rabin-Karp

Robin- Karp is using the rolling hash and comparing it to our original string to see if they have the same hash.

Where n is length of *L* and m is length of *S*

Average running time of O(n + m)

Worst case: O(nm)

e.g. *L* = ‘AAA’, *S* = ‘AAAAAAAAAAAAAAAAAAAAAAAAH’

How to deal with AAAAAAAAAAAAAAAAAH?

Use KMP

Knuth-Morris-Pratt

How many times does a string *L* appear in a string *S.*

We use pre-processing.

When a mistake occurs we do not start from over but to the shortest prefix that ‘works’.

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD instead of ABACABABACABAD

ABACABAD ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Example

L = ABACABAD

S = ABACABABACABAD

ABACABABACABAD

ABACABAD

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

L = ABACABAD

ABACABAD M[0] = 0

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

L = ABACABAD

ABACABAD M[0] = 0

ABACABAD M[1] = 0

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

L = ABACABAD

ABACABAD	M[0] = 0
ABACABAD	M[1] = 0
ABACABAD	M[2] = 0

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

L = ABACABAD

ABACABAD	M[0] = 0
ABACABAD	M[1] = 0
ABACABAD	M[2] = 0
ABACABAD	M[3] = 1

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

L = ABACABAD

ABACABAD	M[0] = 0
ABACABAD	M[1] = 0
ABACABAD	M[2] = 0
ABACABAD	M[3] = 1
ABACABAD	M[4] = 0

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

L = ABACABAD

ABACABAD M[4] = 0

ABACABAD M[5] = 1

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

L = ABACABAD

ABACABAD M[4] = 0

ABACABAD M[5] = 1

ABACABAD M[6] = 2

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

L = ABACABAD

ABACABAD	M[4] = 0
ABACABAD	M[5] = 1
ABACABAD	M[6] = 2
ABACABAD	M[7] = 3

Where to fall-back?

M[i] is the length of the longest prefix that is also a suffix of L[:i],

M[i] ≠ i

L = ABACABAD

ABACABAD	M[4] = 0
ABACABAD	M[5] = 1
ABACABAD	M[6] = 2
ABACABAD	M[7] = 3
ABACABAD	M[8] = 0

Where to fall-back?

L = ABABBABABAA

ABABBABABAA

M[0] = 0

Where to fall-back?

L = ABABBABABAA

ABABBABABAA M[0] = 0

ABABBABABAA M[1] = 0

Where to fall-back?

L = ABABBABABAA

ABABBABABAA M[0] = 0

ABABBABABAA M[1] = 0

ABABBABABAA M[2] = 1

Where to fall-back?

L = ABABBABABAA

ABABBABABAA — M[0] = 0

ABABBABABAA — M[1] = 0

ABABBABABAA — M[2] = 1

ABABBABABAA — M[3] = 2

Where to fall-back?

L = ABABBABABAA

ABABBABABAA	M[0] = 0
ABABBABABAA	M[1] = 0
ABABBABABAA	M[2] = 1
ABABBABABAA	M[3] = 2
ABABBABABAA	M[4] = 0

Where to fall-back?

L = ABABBABABAA

ABABBABABAA	M[0] = 0
ABABBABABAA	M[1] = 0
ABABBABABAA	M[2] = 1
ABABBABABAA	M[3] = 2
ABABBABABAA	M[4] = 0
ABABBABABAA	M[5] = 1

Where to fall-back?

L = ABABBABABAA

ABABBABABAA	M[0] = 0
ABABBABABAA	M[1] = 0
ABABBABABAA	M[2] = 1
ABABBABABAA	M[3] = 2
ABABBABABAA	M[4] = 0
ABABBABABAA	M[5] = 1
ABABBABABAA	M[6] = 2

Where to fall-back?

L = ABABBABABAA

ABABBABABAA	M[0] = 0
ABABBABABAA	M[1] = 0
ABABBABABAA	M[2] = 1
ABABBABABAA	M[3] = 2
ABABBABABAA	M[4] = 0
ABABBABABAA	M[5] = 1
ABABBABABAA	M[6] = 2
ABABBABABAA	M[7] = 3

Where to fall-back?

L = ABABBABABAA

ABABBABABAA M[5] = 1

ABABBABABAA M[6] = 2

ABABBABABAA M[7] = 3

ABABBABABAA M[8] = 4

Where to fall-back?

L = ABABBABABAA

ABABBABABAA M[5] = 1

ABABBABABAA M[6] = 2

ABABBABABAA M[7] = 3

ABABBABABAA M[8] = 4

ABABBABABAA M[9] = 3

Where to fall-back?

L = ABABBABABAA

ABABBABABAA	M[5] = 1
ABABBABABAA	M[6] = 2
ABABBABABAA	M[7] = 3
ABABBABABAA	M[8] = 4
ABABBABABAA	M[9] = 3
ABABBABABAA	M[10] = 1

How to implement?

```
lenn = 0
\\len is the longest prefix of L that currently matches up to S[i]
for i in range(len(S)):
        while (L[lenn] != S[i] and lenn > 0):
                \\Change the start until it matches S[i] or is 0
                lenn = M[lenn- 1]
                \\Off by 1 errors will make you suicidal
        if (L[lenn] == S[i]):
                len++
        If (lenn == L.size()):
                \\The entire L has been found in S
                ans++
                lenn = M[lenn - 1]
```

How to find M?

You can do the $O(n^2)$ which isn't too bad.

There is a $O(n)$ which is similar to the previous code .

How to find M?

```
M = [0, 0]
lenn = 0
for i in range(1, len(L)):
    while (L[lenn] != L[i] and lenn > 0):
        lenn = M[lenn- 1]
    if (L[lenn] == L[i]):
        lenn++
    M.append(lenn)
```

Note:

For some reason my code is a lot simpler than other sites.

So maybe my code is slow or doesn’t work.

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

https://en.wikipedia.org/wiki/Knuth%E2%80%93Morris%E2%80%93Pratt_algorithm

If you need code.

Time complexity of KMP

O(n) preprocessing

O(m) matching time

O(n + m) total time