Day 15 Pattern Searching Algos.

(i) Basic problem
(ii) Pattern matching
(iii) problems on (ii)

Q. Given strings s1 and s2, we need to tell whether the two strings are isomorphic.

↓

characters of s2 can be replaced by characters of s1 to get a similar string.

↓

example. egg → add ✓

foo → ban ✗

geg → add ✗

relative position matters.

↓

2 hashmaps to store mapping from s1 → s2 and s2 → s1. If while traversing the strings, any anomaly occurs, we will return true;

i.e. if there is a one to one mapping possible for each character in the string with the other one, and all occurrences of every character in str1 map to some char in str2.

* Pattern matching

text: a b c b c g l x

pattern: b c g l

↓

Brute force: for each character in the text, we start matching the pattern, one by one.

Time complexity for which will be O(m*l)

m → length of text

l → length of pattern

↓

this is inefficient & needs to be optimised.

↓

text: a b c n a b c d a b m a b c d a b a b c d c y

pattern: a b c d a b c y

↓ ↕ x

in brute force the characters in the pattern string are repeatedly matched with chars in the text string. No account of already matched chars in the previous check is maintained, which information could be of use in the next search & can save us a lot of operations.

↓

This is utilised in KMP approach.

(Knuth - Morris - Pratt)

↓

basic premise or hypothesis is that if in the pattern string we have a suffix substring which is a prefix then for the next search, the matching can start from element next to the prefix

O(m+n)

↑

so now the matching can begin from c

↑

for example abcdab matched with text, since ab is a prefix and a suffix in the substring

i) This involves a pre-processing step to build a lookup array which will give us the length of the longest substring which is a suffix and a prefix at all index of pattern string.

lps (longest prefix & suffix)

	0	1	2	3	4	5	6	7
	a	b	c	d	a	b	c	y
lps	0	0	0	0	1	2	3	0

(pattern preprocessing step)

* we take 2 variables i & j and check if the characters at the two indices match, if they do then lps[j] = i+1, if they don't then i is transferred to lps[j-1] if i is already not 0; if it is then lps[j]=0, and j is incremented. This is repeated till j reaches the end of the array

↓

the movement of i to lps[j-1] is done to check the current string with the previous string which is a prefix & suffix, and check for that.

(kind of dp?)

a y a b a y a y
0 0 1 0 1 2 3 2

a y a y a b a b a y a y
0 0 1 2 3 0 1 0 1 2 3 4

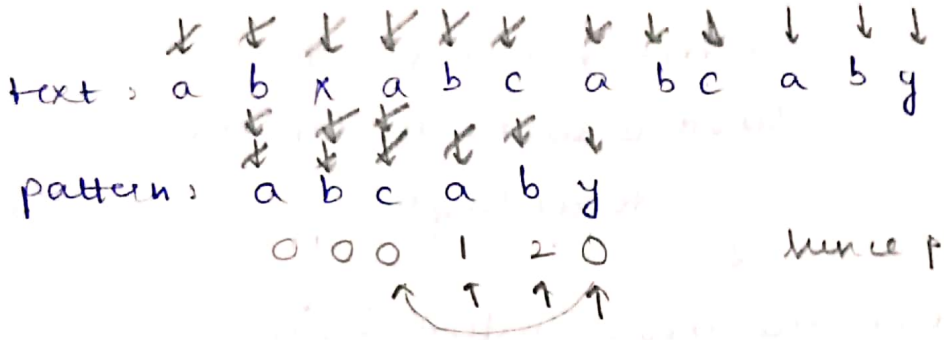
a a a a a a b
0 1 2 3 4 5 0

a b a b a b a b y
0 0 1 2 3 4 5 6 0
0 1 2 3 4 5 6 7 8

suffix which is also a prefix for the substring ending at the ith index.

0	1	2	3	4	5	6	7	8
a	y	c	d	a	y	c	a	y
0	0	0	0	1	2	3	1	2

ii) using the lookup table to find the pattern in the text.



since pattern matches

* on a mismatch the lookup array takes its pointer to the index next to the lps of the previous subarray to check if the match happens for that substring or not & the search continues.

↓

constructing lps → O(m)

search for pattern in text using lps → O(n)

↓

O(m+n)

space : O(m) auxilary space.

Q. Given a string, how many characters need to be added in front to make it a palindrome.

edef ffede

00100012 3

str.length() - lps.back()

for a palindrome, rev(str) = org(str),

↓

problem can be to find how
far from rev, our org. string
is.

↓

longest prefix in org. which is
suffix in reverse, and the
length is subtracted from the
total length.

↓

length of the longest palindrome in
the string starting at ~~0~~ 0th index,
the rest letters can be added to
the string accordingly.

fedef

ccaaeb aabeaacc

ccaabaacc

for a palindrome, the
string, all prefix and
suffixes are same.

longest prefix which is
a suffix

aabaa$aaboa ←
01012012345

abcde$edcba
00000000000

dabae$eabad
00000000000

dabaed$deabad
00000100001

dabaedd$daedbad
00000010120001

Another pattern matching algorithm

↓

Rabin Karp algorithm

↓

uses the concept of hashing.

↓

i) compute the hash of the pattern.

ii) calculate hash of all substrings of length of the pattern.

{ If the hashes match, then do character by character matching, else move on → if we calculate hash code for all substrs of length m, then O(m*n) time will be elapsed ↓ efficient method req.

CPP, JAVA, prime for hash function: 31

for example: a b d a b c : text

a b c : pattern

let hash: $\sum_{i=0}^{m} p^i \cdot str[i]$ where p is a prime no

↓

let p: 3 here and let ascii value for alphabets be 1 under. i.e. 1-26 for lowercase alphabets.

hash(abc) = $1 * 3^0 + 2 * 3^1 + 3 * 3^2$

= 1 + 6 + 27

= 34

hash(abd) = $1 * 3^0 + 2 * 3^1 + 4 * 3^2$

= 1 + 6 + 36

hash(abd) ≠ hash(abc) : 43

now hash(bda) is not calculated from scratch rather hash(abd) is recycled for bda

hash(bda) = $2 + 3^0 + 4 * 3 + 1 * 3^2$

$= 2 + 12 + 9$

$= 23$

$$\frac{h(abd) - a}{p} + a * p^2$$

$$\frac{h(abd) - a}{3} + a * q$$ → O(1) operation, rather than looping on pattern for hash.

$\frac{43 - 1}{3} + 1 * 9 = 14 + 9 = 23$ ✓

↓

in worst case, the time complexity

O(m*n)

↓

when all characters of pattern are same & all characters of text are same.

↓

hashes will be same, and character by character check would be required repeatedly, which was avoided in KMP.

If the hash function is not good, and collisions are frequent, then time elapsed will be more

↓

implementation of hash code fn. and the i/p.

* How do we know that prime number hash is a good hash function?

i) distribution should be uniform good. (well spread)

ii) If non prime, then the numbers fall in the same range, which is undesirable

Q. Given a stream of characters, we need to return whether the updated string is a palindrome or not.

↓

i) create partition, calculate hash for the rev of RHS, if the hash for rev(RHS) == hash for LHS, then it becomes a candidate for character by character search, if it doesn't then we return no.

a b c b a k

h(a b c) h(k a b)

ii) calculating the hash each time would be expensive and to answer the query in O(1), we need to use the previous hash values to calculate the current value.

a b c | b a k

$h(abc) = 1 * 3^0 + 2 * 3^1 + 3 * 3^2$

$= 34 \% 3 = 1$

$h(kab) = 11 + 3 + 18 = 32 \% 3 = 2$

↓

a b c | a k d

even → odd

$h(abc) = 34 \% 3 = 1$

$h(dka) = 4 + 11 * 3^1 + 1 * 3^2 = 4 + 33 + 9 = 46$

$h(dka) = (h(kab) - 2 * 3^2) * 3 + d(4)$

$= (32 - 18) * 3 + 4$

$= 14 * 3 + 4 = 42 \% 3 = 0.$

a b c b a k d e odd ↓ even

$h(abcb) = h(abc) + 2 * 3^4$

$= 34 + 2 * 81$

$= 34 + 162 = 196 \% 3 = 0$

$h(edka) = h(dka)^* 3 + 5$

$= 42 * 3 + 5$

$= 126 + 5 = 131 \% 3 = 2$

* prime number for hash: consider key ∈ [0, 100] and a hash table has m = 12. since 3 is a factor of 12, the keys which are different order multiples of 3 will fall in buckets that are multiples of 3.

keys [0, 12, 24, 36) → 0

(3, 15, 27, 39) → 3

(6, 18, 30, 42) → 6

(9, 21, 33, 45) → 9

↓

if the elements are biased to being 3 centric then the chaining in these buckets will be very high, which is unfavourable. ∴ the factors of m, should be minimal, and that is best represented or taken care of by a prime number.