29/10/2024

Pattern matching

* Pattern matching is the technique that identifies specific patterns or sequences or a combination of characters with a in a larger piece of information.

Ex:- Consider

text [0... n-1]

Pattern [0... m-1]

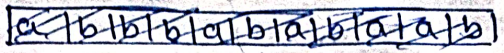
check wheather the pattern is available in the text or not for checking purpose we are using different algorithms basic new algorithm is Brute Force approach.

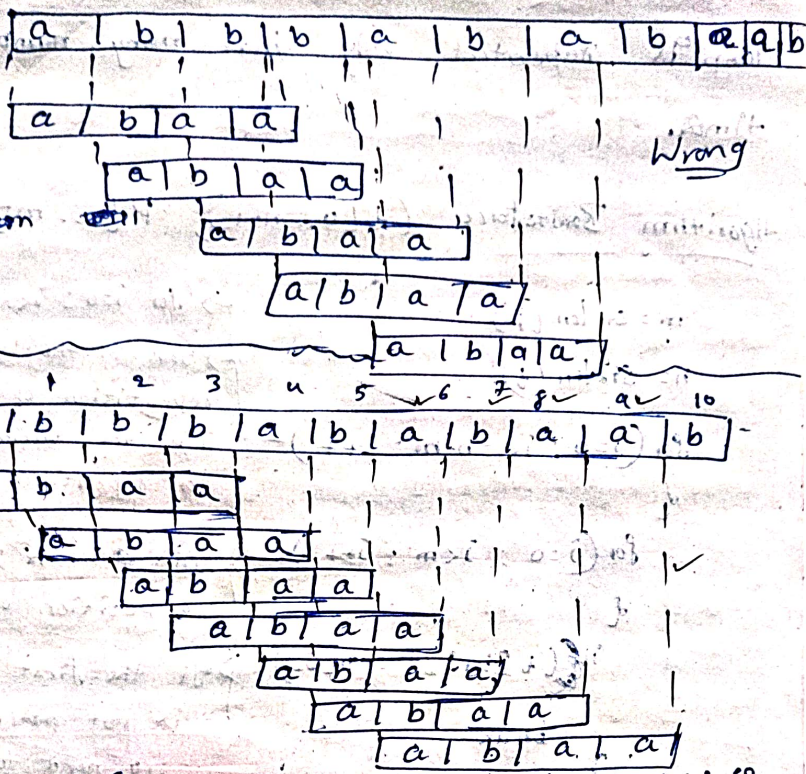
Brute Force Approach :-

Consider two strings one is text another one is Pattern the procedure is compare character by character that means the

* first two character of the text is compared with the first character of the pattern if both are Equal move to second character of the text, compared with the second Character of the Pattern. if both are matched move to third ... so the Procen completed until if the complete Pattern is available or pattern is not available

Ex:-





* Pattern will mismatch occured the concept is shift the pattern one cell to the right &

* again we back track to index (0)

it goes on continuesly. whenever the pattern is matched it returns the index (6)

* Algorithm consist of two loops one loop is u used for repetition of the text, & one loop repetition for the pattern.

* if pattern is not matching completly then what it happens u

text all Element = n
pattern Element = m
} n-m+1

loop is repeated only that many number of times.

Algorithm Bruteforce (t[0..n-1], p[0..m-1])
{
m = strlen(p)
n = strlen(t)
for (i=0 ; i<=n-m ; i++)
{
for (j=0 ; j<m ; j++)
{
if (t[i+j] != p[j])
break
}
if (j==m)
return i;
}
}

for(i=0 ; i<n-m+1; i++) → This is the loop for repeatition of text the I is the variable index for repeating text.

for(j=0 ; j<m ; j++) → This is the loop for repetition of pattern

if(t[i+j] != p[j]) → If the first character is not match means no need to repeat this loop,

Tracing:- a b b b a b a b a a b
0 1 2 3 4 5 6 7 8 9 10
a b a a

i) i=0 j=0 t[0] != p[0] a != a ✓

ii) j=1 i=0 t[0+1] != p[1] b != b ✓

iii) j=2 i=0 t[0+2] != p[2] b != a ← break

for(j==m) (2==4) ✗

for(i=0 ; i<n-m+1; i++) i=1 j=0

```
def find_brute (T, P):
    "Return the lowest index of T at which substring P
    begins (or else -1)"
    n, m = len(T), len(P)
    for i in range (n-m+1):
        k = 0
        while k<m and T[i+k] == P[k]
            k += 1
        if k == m:
            return i
    return -1
```

Boyer Moore Algorithm

* we have pre-processing step where we construct a ~~bad~~ match tape in this algorithm works with the idea of a bad match rule and a good shift rule

Bad match rules:- we calculate the values for each and every characters using a formula in the pattern. ~~the good shift rule is~~ using value = length - index - 1

Good shift rules:- using the values of their characters of the pattern we keep on shifting the pattern position in the right ~~one~~ corresponding to the value of the table. Ex:-

Text = HE L COMETOSURANA COLLEGE n = |T| = 22

Pattern = S U R A N A m = |P| = 6

P = S U R A N A

0 1 2 3 4 5

Bat match this

S	U	R	A	N	*
5	4	3	6	1	6

In this table where all the characters from the pattern are being written now they should not be any repeated characters in the table.

SURANA → A is repeated so in table we no need to write.

SURAN *

0 1 2 3 4

where the value of this star (*) is always equal to the length of the pattern. m|p| = 6

value = length - index - 1

S = 6 - 0 - 1 = 5

U = 6 - 1 - 1 = 4

R = 6 - 2 - 1 = 3

A = 6

N = 6 - 4 - 1 = 1

If in case last is not A then

SURANA (S)

0 1 2 3 4 5 6

value = length - index - 1

A = 6 - 5 - 1 = 0

S	U	R	A	N	A	S
5	4	3	6	1		

last character 0 skipped here.

T = WELCOME TO SURANA COLLEGE

P = SURANA M = 6

Step 1:- we match the characters always from right to left now the characters will be matched with the characters & now there is no much which characters is mill matched, then check the table, check the value & shift that to next.

(*) If you are text value is not matching in a table character the value of that string is star (*) value.

step 2:-

T = W E L C O M E T O S U R A N A C O L L E G E

P : S U R A N A (R=3) → check in table

i+1 = 6+1 position move the pattern.

* Comparing is starts from right to left

index (position)

step 3:-

T = W E L C O M E T O S U R A N A C O L L E G E

P: o/b S U R A N A

P present in T at 6th position. (index)

Ex:-2:-

T = W E L C O M E T O S U R A N A C O L L E G E

P = C O L L E G E
0 1 2 3 4 5 6

Table					
C	O	L	E	G	*
6	5	4	3	1	7

value = length - index - 1

C = 7 - 0 - 1 = 6

O = 7 - 1 - 1 = 5

L = 7 - 2 - 1 = 4

L = 7 - 3 - 1 = 3

E = 7 - 4 - 1 = 2

G = 7 - 5 - 1 = 1

E = 7

If character is repeated the second character value is shifted to first character.

* when you are doing shifting then check left side & shift

tracing:- T = W E L C O M E T O S U R A N A C O L L E G E

P = C O L L E G E

M = ? so take E = 7

N = ? so take * = 7.

G = 1

* Matching starts from right side to left side.

shift position

Test:- T = W E L C O M E T O S U R A N A C O L L E G E

P =

P present in T at (index) position 15

~~Knuth - Morris - Pratt~~

```
def find_boyer_moore(T, P):
    """Return the lowest index of T at which substring P begin Grew."""
    n, m = len(T), len(P)
    if m == 0: return 0
    last = {}
    for k in range(m):
        last[P[k]] = k

    i = m-1
    k = m-1
    while i < n:
        if T[i] == P[k]:
            if k == 0:
                return i
            else:
                i -= 1
                k -= 1
        else:
            j = last.get(T[i], -1)
            i += m - min(k, j+1)
            k = m-1
    return -1
```

Knuth - Morris - Pratt (KMP)

```
F[0] = 0
i = 1
j = 0
while (i < m) {
    if (P[i] == P[j]) {
        F[i] = j+1
        i = i+1
        j = j+1
    }
    else if (j > 0)
        j = F[j-1]
    else
        F[i] = 0
        j = i+1
```

KMP algorithm do simply considers an Element of the text you need avoid comparison of an Element of the text that how previously involved in comparison with the same Element of the pattern, if in case comparison is already now you need to avoid the comparison one more time that means backtracking is not achieved.

* Compare the first two character of the text with the first character of the pattern if both are Equal moved to second one if both are Equal move to next one whenever there is a failure that means, mismatch occured the KMP algorithm uses LPS table, Prefix table, & failure function.

```
def find_kmp(T,P):
 Return the lowest index of T at which substring P begins (or else -1)
 n, m = len(T), len(P)          # introduce convenient notations
 if m==0: return 0              # trivial search for every string
 fail = compute_kmp_fail(P)     # rely on utility to precompute
 j=0                            # index into text
 k=0                            # index into pattern
 while j<n:
   if T[j]==P[k]:               # P[0:1+k] matched thus far
     if k==m-1                  # match is complete
       return j-m+1
```

```
        j += 1                  # try to extend match
        k = 1
    elif k > 0:
        k = fail[k-1]           # reuse suffix of P[0:k]
    else:
        j += 1
return -1                       # reached end without match
```

* Each time we have two characters that match, we set $f(j) = k+1$. Since we have $j > k$ throughout the execution of the algorithm $f(k-1)$ is always well defined when we need to use it.

```
def compute_kmp_fail(P):
    """Utility that computes and returns KMP 'fail' list."""
    m = len(P)
    fail = [0] * m              # by default, presume overlap of 0 everywhere
    j = 1
    k = 0
    while j < m:                # compute fail[j] during this pass, if nonzero
        if P[j] == P[k]:        # k+1 characters match thus far
            fail[j] = k + 1
            j += 1
            k += 1
        elif k > 0:             # k follows a matching prefix
            k = fail[k-1]
        else:                   # no match found starting at j
            j += 1
    return fail
```

Text Compression

The Huffman coding Algorithm $(O(n + d \log d))$

* Text Compression is useful in problems where we are given a string x defined over some alphabet, such as the ASCII or unicode character sets, & we want to Efficiently Encode x into a small binary string y (only the characters 0 and 1). It is useful in any situation where we wish to reduce bandwidth for digital communications.

* The Huffman coding saves space over a fixed-length Encoding by using short code-word strings to Encode high-frequency characters & long code-word strings to Encode low frequency characters.

* The Huffman coding Algorithm begins with Each of the d distinct characters of the string X to Encode being the root node of a Single-node binary tree. The Algorithm proceeds in a series of rounds. The algorithm takes the two binary trees,

Algorithm Huffman(X)

Input : String x of length n with d distinct characters

Output : Coding tree of X

Compute the frequency f(c) of Each character c of X

Initialize a priority queue Q

for Each character c in X do

Create a single-node binary tree T storing

Insert T into Q with key f(c)

while len(Q) > 1 do

$(f_1, T_1) = Q.remove\_min()$

$(f_2, T_2) = Q.remove\_min()$

Create a new binary tree T_1 with left subtree T_1 and right subtree T_2

Insert T into Q with key $f_1 + f_2$

$(f_1, T) = Q.remove\_min()$

return tree T.

Greedy Method :- Huffman's algorithm for building an optimical uncoding is Example of greedy method. this design pattern is applied to optimization problems, where we are trying to construct some structure while minimizing or maximizing some property of that structure while minimizing or maximizing some property of that structure.

The general formula for the greedy method Pattern is almost as simple as the brute force method. this approach does not always lead to an optimal solution.

For some problems it works for which posses the greedy-choice property which means that a global optimal condition which can be reached by a series of locally optimal choices starting from a well defined starting condition.

Tries: A trie is a tree based data structure for storing string in order to support fast pattern matching. the main application is information retrival. the primary query operations that tries support are pattern matching & prefix matching

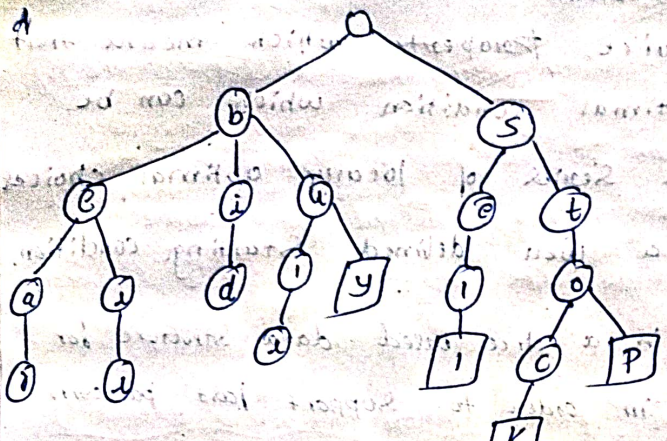
Standard Tries:- Let S be a set of s strings from alphabet Σ such that no string in S is a prefix of another string A standard trie for S is an order tree T with the following properties

-> Each node of T. Except the root is labeled with a character of Σ

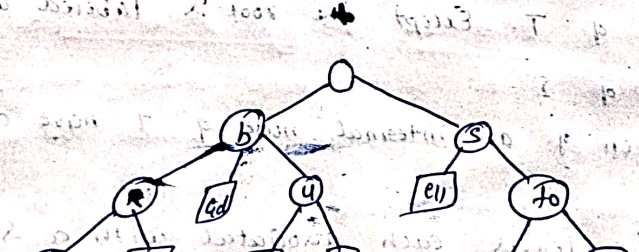
-> The children of an internal node of T have distinct labels,

-> T has s leaves, each associated with a string of S, such that the concatenation of the labels of the nodes on the path from root to a leaf v of T yeilds the string of S associated with that a trie T represents the strings of S with paths from the root to the leaves of T.

Ex:- { bear, bell, bid, bull, buy, sell, stock, stop }

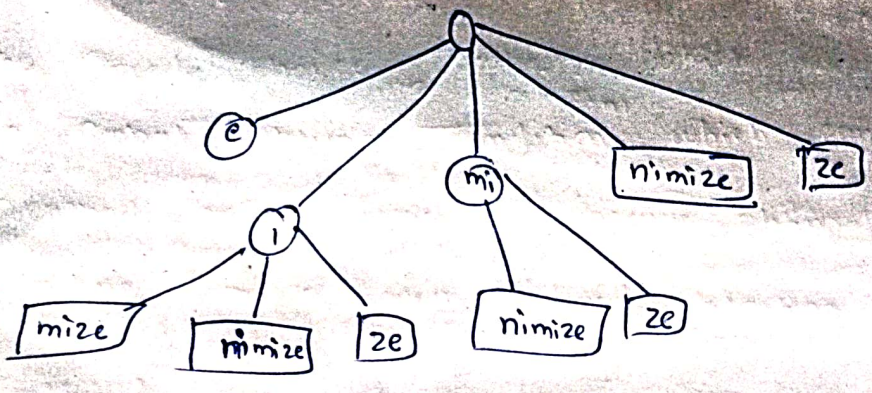


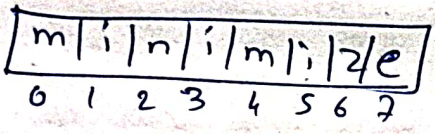
Compressed trie :- It is similar to standard trie but it ensures that each internal node in the trie has at least two children. It enforces this rule by compressing chains of single child nodes into individual edges. Let T be a standard trie we say that an internal node v of T is redundant if it has one child and is not the root.

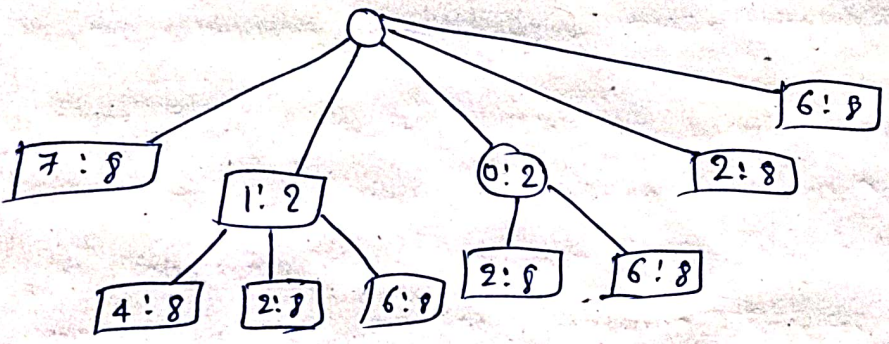


{bear, bell, bid, bull, buy, sell, stock, stop}

Suffix Tries :- One of the primary applications for tries is for the case when the strings in the collection S are all the suffixes of a string x, such a trie is called suffix trie of string x. Ex :- {minimize}







⊗ using a Suffix trie allows us to save space over a standard trie by using several space compression technique

⊗ The Suffix Trie T for a string X can be used to Efficiently perform pattern matching queries on text X.