

Pattern matching

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

Ex:- Consider

text $[0 \dots n-1]$

Pattern $[0 \dots m-1]$

Check whether 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

① first two characters 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 process completed until if the complete pattern is available ② pattern is not available

Ex: -

test

~~a b b b a b a b a b a b~~

text

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| a | b | b | b | a | b | a | b | a | b |
| a | b | a | a | | | | | | |
| | a | b | a | a | | | | | |
| | | a | b | a | a | | | | |
| | | | a | b | a | a | | | |
| | | | | a | b | a | a | | |
| | | | | | a | b | a | a | |

Wrong

| | | | | | | | | | | | |
|--------|---|---|---|---|---|---|---|---|---|---|---|
| text | a | b | b | b | a | b | a | b | a | a | b |
| bottom | a | b | a | a | | | | | | | |
| | a | b | a | a | | | | | | | |
| | a | b | a | a | | | | | | | |
| | a | b | b | a | a | | | | | | |
| | a | b | b | a | a | | | | | | |
| | a | b | b | a | a | | | | | | |
| | a | b | b | a | a | | | | | | |

- * Pattern will mismatch occurred the concept if shift the pattern one cell to the right &
- * again we back track to index @

~~so~~ it goes on continuously. whenever the pattern is matched it returns the index 6

- ⑥ Algorithm consist of two loops one loop is u used for repetition of the text & one loop repetition for the ~~next~~ pattern.
- ⑦ if pattern is not matching completely then what it happens "

text all Element = n } n-m+1

pattern Elements ~~(if it matches then)~~

loop goes 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 ($t[i+j] \neq p[j]$)

 break

 if ($i+j == m-1$)

 return i

}

Tracing:- a b b b a b a b a a b

 i=0 j=0 $t[0] \neq p[0]$ $j=1, i=0, t[0+1] = p[1]$

 a!=a break b!=b

 j=2 i=0 $t[0+2] \neq p[2]$

 a!=a break b!=b

 j=3 i=0 $t[0+3] \neq p[3]$

 a!=a break b!=b

 j=4 i=0 $t[0+4] \neq p[4]$

 a!=a break b!=b

 j=5 i=0 $t[0+5] \neq p[5]$

 a!=a break b!=b

 j=6 i=0 $t[0+6] \neq p[6]$

 a!=a break b!=b

 j=7 i=0 $t[0+7] \neq p[7]$

 a!=a break b!=b

 j=8 i=0 $t[0+8] \neq p[8]$

 a!=a break b!=b

 j=9 i=0 $t[0+9] \neq p[9]$

 a!=a break b!=b

 j=10 i=0 $t[0+10] \neq p[10]$

 a!=a break b!=b

 j=11 i=0 $t[0+11] \neq p[11]$

 a!=a break b!=b

 j=12 i=0 $t[0+12] \neq p[12]$

 a!=a break b!=b

 j=13 i=0 $t[0+13] \neq p[13]$

 a!=a break b!=b

 j=14 i=0 $t[0+14] \neq p[14]$

 a!=a break b!=b

 j=15 i=0 $t[0+15] \neq p[15]$

 a!=a break b!=b

 j=16 i=0 $t[0+16] \neq p[16]$

 a!=a break b!=b

 j=17 i=0 $t[0+17] \neq p[17]$

 a!=a break b!=b

 j=18 i=0 $t[0+18] \neq p[18]$

 a!=a break b!=b

 j=19 i=0 $t[0+19] \neq p[19]$

 a!=a break b!=b

 j=20 i=0 $t[0+20] \neq p[20]$

 a!=a break b!=b

 j=21 i=0 $t[0+21] \neq p[21]$

 a!=a break b!=b

 j=22 i=0 $t[0+22] \neq p[22]$

 a!=a break b!=b

 j=23 i=0 $t[0+23] \neq p[23]$

 a!=a break b!=b

 j=24 i=0 $t[0+24] \neq p[24]$

 a!=a break b!=b

 j=25 i=0 $t[0+25] \neq p[25]$

 a!=a break b!=b

 j=26 i=0 $t[0+26] \neq p[26]$

 a!=a break b!=b

 j=27 i=0 $t[0+27] \neq p[27]$

 a!=a break b!=b

 j=28 i=0 $t[0+28] \neq p[28]$

 a!=a break b!=b

 j=29 i=0 $t[0+29] \neq p[29]$

 a!=a break b!=b

 j=30 i=0 $t[0+30] \neq p[30]$

 a!=a break b!=b

 j=31 i=0 $t[0+31] \neq p[31]$

 a!=a break b!=b

 j=32 i=0 $t[0+32] \neq p[32]$

 a!=a break b!=b

 j=33 i=0 $t[0+33] \neq p[33]$

 a!=a break b!=b

 j=34 i=0 $t[0+34] \neq p[34]$

 a!=a break b!=b

 j=35 i=0 $t[0+35] \neq p[35]$

 a!=a break b!=b

 j=36 i=0 $t[0+36] \neq p[36]$

 a!=a break b!=b

 j=37 i=0 $t[0+37] \neq p[37]$

 a!=a break b!=b

 j=38 i=0 $t[0+38] \neq p[38]$

 a!=a break b!=b

 j=39 i=0 $t[0+39] \neq p[39]$

 a!=a break b!=b

 j=40 i=0 $t[0+40] \neq p[40]$

 a!=a break b!=b

 j=41 i=0 $t[0+41] \neq p[41]$

 a!=a break b!=b

 j=42 i=0 $t[0+42] \neq p[42]$

 a!=a break b!=b

 j=43 i=0 $t[0+43] \neq p[43]$

 a!=a break b!=b

 j=44 i=0 $t[0+44] \neq p[44]$

 a!=a break b!=b

 j=45 i=0 $t[0+45] \neq p[45]$

 a!=a break b!=b

 j=46 i=0 $t[0+46] \neq p[46]$

 a!=a break b!=b

 j=47 i=0 $t[0+47] \neq p[47]$

 a!=a break b!=b

 j=48 i=0 $t[0+48] \neq p[48]$

 a!=a break b!=b

 j=49 i=0 $t[0+49] \neq p[49]$

 a!=a break b!=b

 j=50 i=0 $t[0+50] \neq p[50]$

 a!=a break b!=b

 j=51 i=0 $t[0+51] \neq p[51]$

 a!=a break b!=b

 j=52 i=0 $t[0+52] \neq p[52]$

 a!=a break b!=b

 j=53 i=0 $t[0+53] \neq p[53]$

 a!=a break b!=b

 j=54 i=0 $t[0+54] \neq p[54]$

 a!=a break b!=b

 j=55 i=0 $t[0+55] \neq p[55]$

 a!=a break b!=b

 j=56 i=0 $t[0+56] \neq p[56]$

 a!=a break b!=b

 j=57 i=0 $t[0+57] \neq p[57]$

 a!=a break b!=b

 j=58 i=0 $t[0+58] \neq p[58]$

 a!=a break b!=b

 j=59 i=0 $t[0+59] \neq p[59]$

 a!=a break b!=b

 j=60 i=0 $t[0+60] \neq p[60]$

 a!=a break b!=b

 j=61 i=0 $t[0+61] \neq p[61]$

 a!=a break b!=b

 j=62 i=0 $t[0+62] \neq p[62]$

 a!=a break b!=b

 j=63 i=0 $t[0+63] \neq p[63]$

 a!=a break b!=b

 j=64 i=0 $t[0+64] \neq p[64]$

 a!=a break b!=b

 j=65 i=0 $t[0+65] \neq p[65]$

 a!=a break b!=b

 j=66 i=0 $t[0+66] \neq p[66]$

 a!=a break b!=b

 j=67 i=0 $t[0+67] \neq p[67]$

 a!=a break b!=b

 j=68 i=0 $t[0+68] \neq p[68]$

 a!=a break b!=b

 j=69 i=0 $t[0+69] \neq p[69]$

 a!=a break b!=b

 j=70 i=0 $t[0+70] \neq p[70]$

 a!=a break b!=b

 j=71 i=0 $t[0+71] \neq p[71]$

 a!=a break b!=b

 j=72 i=0 $t[0+72] \neq p[72]$

 a!=a break b!=b

 j=73 i=0 $t[0+73] \neq p[73]$

 a!=a break b!=b

 j=74 i=0 $t[0+74] \neq p[74]$

 a!=a break b!=b

 j=75 i=0 $t[0+75] \neq p[75]$

 a!=a break b!=b

 j=76 i=0 $t[0+76] \neq p[76]$

 a!=a break b!=b

 j=77 i=0 $t[0+77] \neq p[77]$

 a!=a break b!=b

 j=78 i=0 $t[0+78] \neq p[78]$

 a!=a break b!=b

 j=79 i=0 $t[0+79] \neq p[79]$

 a!=a break b!=b

 j=80 i=0 $t[0+80] \neq p[80]$

 a!=a break b!=b

 j=81 i=0 $t[0+81] \neq p[81]$

 a!=a break b!=b

 j=82 i=0 $t[0+82] \neq p[82]$

 a!=a break b!=b

 j=83 i=0 $t[0+83] \neq p[83]$

 a!=a break b!=b

 j=84 i=0 $t[0+84] \neq p[84]$

 a!=a break b!=b

 j=85 i=0 $t[0+85] \neq p[85]$

 a!=a break b!=b

 j=86 i=0 $t[0+86] \neq p[86]$

 a!=a break b!=b

 j=87 i=0 $t[0+87] \neq p[87]$

 a!=a break b!=b

 j=88 i=0 $t[0+88] \neq p[88]$

 a!=a break b!=b

 j=89 i=0 $t[0+89] \neq p[89]$

 a!=a break b!=b

 j=90 i=0 $t[0+90] \neq p[90]$

 a!=a break b!=b

 j=91 i=0 $t[0+91] \neq p[91]$

 a!=a break b!=b

 j=92 i=0 $t[0+92] \neq p[92]$

 a!=a break b!=b

 j=93 i=0 $t[0+93] \neq p[93]$

 a!=a break b!=b

 j=94 i=0 $t[0+94] \neq p[94]$

 a!=a break b!=b

 j=95 i=0 $t[0+95] \neq p[95]$

 a!=a break b!=b

 j=96 i=0 $t[0+96] \neq p[96]$

 a!=a break b!=b

 j=97 i=0 $t[0+97] \neq p[97]$

 a!=a break b!=b

 j=98 i=0 $t[0+98] \neq p[98]$

 a!=a break b!=b

 j=99 i=0 $t[0+99] \neq p[99]$

 a!=a break b!=b

 j=100 i=0 $t[0+100] \neq p[100]$

 a!=a break b!=b

 j=101 i=0 $t[0+101] \neq p[101]$

 a!=a break b!=b

 j=102 i=0 $t[0+102] \neq p[10$

def find-brute (T, P):

Return the lowest index of T at which Substring P

begins (or else -1)

$n, m = \text{len}(T), \text{len}(P)$

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 rule: we calculate the values for each character in the pattern.

Good shift rule: using the values of their characters in the pattern we keep on shifting the pattern position in the right corresponding to the value of the table.

Ex:
Text = HELLOCOMETOSURANA COLLEGE
 $n=|T|=22$
Pattern = SURANA
 $m=|P|=6$

classify slide as of question type, as asked first few are in Q3, where (*) note is given for a prior part

P = SURANA

0 1 2 3 4 5 6 7 8 9

| | | | | |
|---|---|---|---|---|
| S | U | R | A | N |
| 5 | 4 | 3 | 6 | 7 |

In this table where all the characters from the pattern are being written now they should not be any repeated character 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. $m|p| = 6$

$$\text{value} = \text{length} - \text{index} - 1$$

value = 6 - 1 - 1 = 4

value = 6 - 2 - 1 = 3

$$A = 6$$

$$N = 6 - 4 - 1 = 1$$

if in case last is not A then

S U R A N A 5

6 1 2 3 4 5 6

value = length - index - 2

$$A = 6 - 5 - 1 = 0$$

S U R A N A 5

6 4 3 8 1 .

last character missed here.

T = WELCOME TO SURANA COLLEGE

P = SURANA M = 6

Step 1:- we match the characters always from right to left

now the character will be matched with the character b now there is not much

which character is miss matched then check the table, check the value & ~~the~~ shift that to next.

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

Step 2:
 $T = \text{W E L C O M E T O S U R A N A C O L L E G E}$
 $P = \text{C O L L E G E}$

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

* Comparing it starts from right to left index (position)

Step 3:-

$T = \text{W E L C O M E T O S U R A N A C O L L E G E}$
 $P = \text{C O L L E G E}$

P present in T at 7th position. (index) ✓

Step 4:-

$T = \text{W E L C O M E T O S U R A N A C O L L E G E}$

$P = \text{C O L L E G E}$
 $0 \ 1 \ 2 \ 3 \ 4 \ 5 \ 6$

Table

| C | O | L | E | C | G | E |
|---|---|---|---|---|---|---|
| 6 | 5 | 4 | 3 | 7 | 1 | 7 |

value = length - index - 1

If character is repeated

$$C = 7 - 0 - 1 = 6$$

the second character

$$O = 7 - 1 - 1 = 5$$

value is shifted to

$$L = 7 - 2 - 1 = 4$$

first character.

$$E = 7 - 3 - 1 = 3$$

* when you are doing shifting
then check left side
shift

$$M = 7 - 4 - 1 = 2$$

$$G = 7 - 5 - 1 = 1$$

$$E = 7$$

tracing :- $T = \text{W E L C O M E T O S U R A N A C O L L E G E}$
 $P = \text{C O L L E G E}$

* Matching starts from
right side to left side.

Test:- $T = \text{W E L C O M E T O S U R A N A C O L L E G E}$
 $P = \text{C O L L E G E}$

P present in T at (index) position is ✓

~~Knuth - Morris - Pratt~~

def find = boses_moores(T, P):

"Return the lowest index of T at which substring P begin from"

n, m = len(T), len(P)

if m == 0: return 0

last = -1

for k in range(m):

last [P[k]] = k

i = m - 1

l = m - 1

while i < n:

if T[i] == P[k]:

if k == 0:

return i

i = i + 1

else:

Knuth - Morris - Pratt (KMP)

while (i < m) &

(P[i] == P[j]) &

f[i] == j

j = j + 1

j = j + 1

else if (j > 0)

j = f[j - 1]

else

f[i] = 0

j = i + 1

KMP algorithm do simply consider 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 back tracking 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 occurred 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 -1)

n, m = len(T), len(P) # introduce convenient notations

if m == 0: return 0 # trivial search for empty 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:k] matched thus far

 if k == m-1: # match is complete

 return j-m+1

Introducing history to extend match

$k+1 = \text{length}(p)$

elif $k > 0$: will start from $s[i+k]$ to search for $p[0:k]$

if $k == \text{fail}[k-1]$: # return suffix of $p[0:k]$

else:

$j += 1$ # move to next character

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 & returns KMP 'fail' array

$m = \text{len}(p)$

fail = [0] * m # by default, presume overlap of 0 elements

$j = 1$

$k = 0$

while $j < m$: # compute $f(j)$ during this pass, & non zero

 if $p[j] == p[k]$: # $k+1$ characters match thus far:

$\text{fail}[j] = k+1$

$j += 1$

$k += 1$

 elif $k > 0$: # k follows a matching prefix

$k = \text{fail}[k-1]$

 else: # no match found starting at j

$j += 1$

return fail

Text Compression

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

① 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 communication.

* 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 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 $\text{Huffman}(x)$

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 &

for each character c in X do

↳ yield (create a "single-node" binary tree T & string)

↳ insert T into $\&$ with key $f(c)$

↳ while $\text{len}(\&) > 1$ do

$(f_1, T_1) = \&, \text{remove_min}()$

$(f_2, T_2) = \&, \text{remove_min}()$

↳ create a new binary tree T , with left subtree

$f_1 T_1$ (and) right subtree T_2

↳ insert T into $\&$ with key $f_1 + f_2$

$(f, T) = \&, \text{remove_min}()$

return tree T

Greedy Method: Huffman's algorithm for building an optimally uncoding with examples 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 posess the greedy-choice property which mean that a global optimal condition which can be reached by a series of locally optimal choices starting from a well defined starting condition.

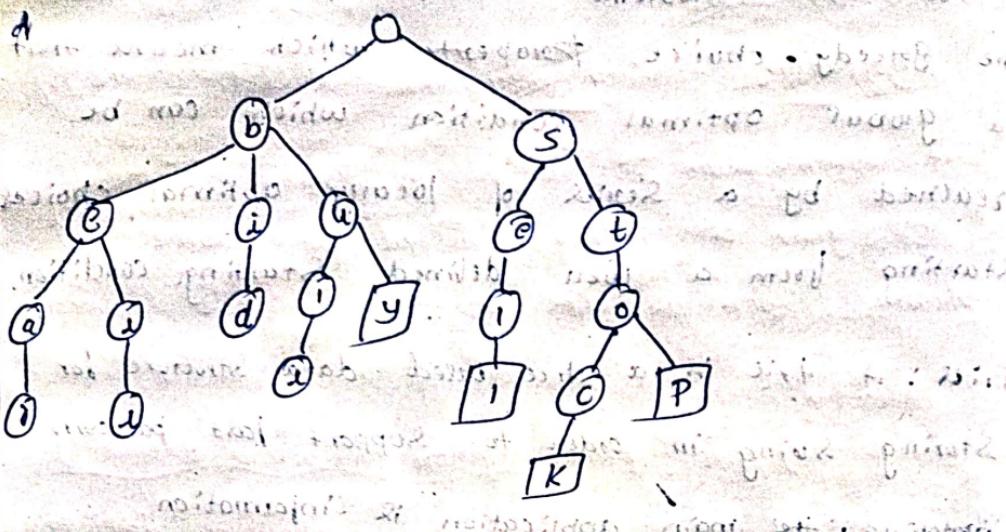
Trie: A trie is a tree based data structure for storing string in order to support fast pattern matching. The main application is information retrieval. The primary query operations that tries support are pattern matching & prefix matching.

Standard Trie: 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 tree for S is an ordered tree T with the following properties:

- Each node of T , except the root is labeled with a character of Σ .

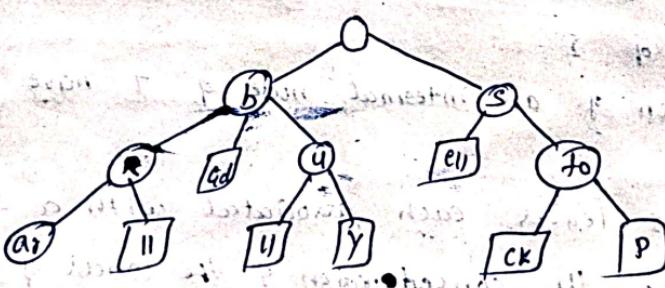
- The children of an internal node of T have distinct labels.
- T has s leaves, each associated with a string of S , such that the concatenation of the labels of the nodes on the path from root to a leaf, v of T yields the string of s associated with that atrie T represents the strings of S with paths from the root to the leaves of T .

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



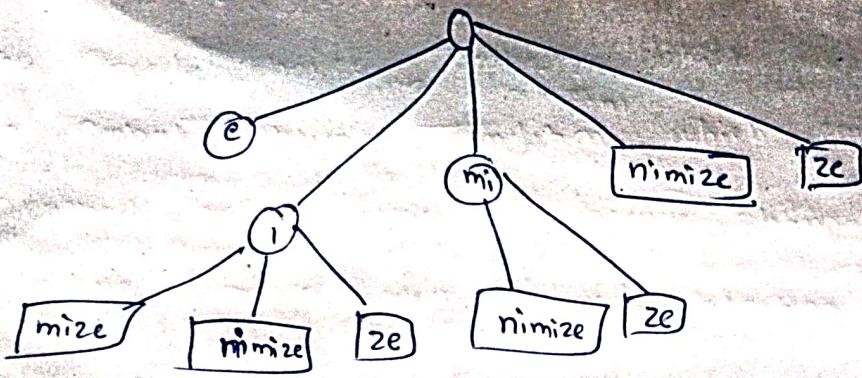
Compressed trie :- It is similar to standard trie but

Ensures that each internal node in the tree has at least two children. It enforces this rule by compressing chains of single child nodes into individual edges. Let T be a standard tree we say that an internal node y of T is redundant if it has one child and is not the root.

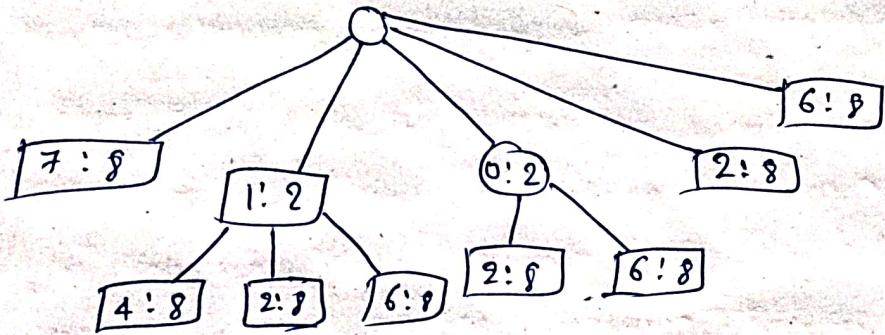


{bear, bell, bid, bull, buy, bark, stock, stop}

Suffix Trie :- 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 tree is called suffix tree, of string X . E.g. {minimize}



| | | | | | | |
|---|---|---|---|---|---|---|
| m | i | n | i | m | z | e |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 |



- ④ Using a Suffix Trie allows us to save space over a standard trie by using several space compression techniques
- ⑤ The Suffix Trie T for a string X can be used to efficiently perform pattern matching queries on text X .