01/5/20
Saturday

Unit-6

String matching:

We have to find whether pattern is there in text or not.

Navie String Matching:

Here text (string) will be given and pattern will be given we have to find all the occurences of the pattern in the given text.

Ex: txt[] = "AABAACAADAABAABA"
pat[] = "AABA".

O/P: pattern found at index 0
pattern found at index 9
pattern found at index 12

⇒ Say text size "n", pattern size m, n>m.
⇒ Take window size equal to the pattern size.
⇒ Start running the window over the text series starting from index 0 then compare.
⇒ Compare window strings with pattern strings, if both equal then print window index.
⇒ Move window forward again compare window string with pattern string, similarly we keep moving the window until we reach the last window.

Ex:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	A	B	A	A	C	A	A	D	A	A	B	A	A	B	A

pattern → 4

A	A	B	A

→ window size: 4

The index of the last window n-m = 16-4 = 12

Algorithm:-

```
{
  m = strlen(pat);
  n = strlen(txt);
  for(i=0; i<=n-m; i++)
  {
    int j;
    for(j=0; j<m; j++)
      if(txt[i+j] != pat[j])
        break;
    if (j==m)
      print "pattern found at index";
  }
}
```

Time Complexity:-

Best case: 1st character of the pattern is not present in the text at all.

Ex: txt[] = "AABCCAADDEE"

pat[] = "FAA" ⇒ Time complexity: O(n).

⇒ Here we compare only 1st character of windows with pattern

⇒ Time complexity is no. of windows i.e n-m ≃ O(n).

Worst case:-

(1) All characters of text & pattern are same

ex: "AAAAAAAAAA" → txt[] ; "AAAA" → pat[]

for each window we have to compare all the characters of the window with pattern.

(2) Only the last character is different. Ex: txt[] = "AAAAAAAAAB"

pat[] = "AAAB".

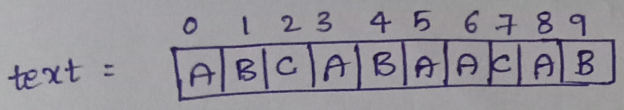
⇒ The no. of comparisions in worst case is O(m * (n-m+1))

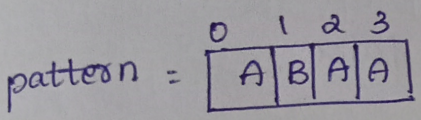
Navie string matches

Given

text = ABCABAACAB

pattern = ABAA





Algorithm

```
{
  m=10
  n=4
  for(i=0; i<=m-n; i++)  i=0; i<=6; i++
  {
    for(j=0; j<n; j++)  j=0; j<4; j++
    {
      if(text[i+j] != pat[j])
          text[0] != pat[0] false

      j=1
          text[0+1] ! pat[1]  false

      j=2
          text[0+2] ! pat[2]  true
            Break
```

* for i=1
  for j=0
    if (text[1+0] != pat[0]; true
      Break;

* for i=2
  for j=0
    if (text[2+0] != pat[0]) true
      Break;

* for i=3

for j=0

if (text [3+0]! pat[0]) false

if (j==4) false

for j=1

if (text [3+1]! pat[1]) false

if (j==n) false

for j=2

if (text [3+2]! pat[2]) false

if (i==4) false

for j=3

if (text [3+3]!= pat[3]) false

if (j==n) true

print (string matches);

* for i=4

for j=0

if (text [4+0]! pat[i]) true

Break;

* for i=5

for j=0

if (text[5+0]! pat[0]) false.

for j=1 if (text[5+1]! pat[1]) true

Break;

* for i=6

j=0 if (text [6+0]! pat [0]) false

j=1 if (text [6+1]! pat [1]) true

Break;

05/05/21
Wednesday

Rabin karp string matching

Text - n

pattern - m

n > m.

ex: "abcdba" → text.

pattern: "cd"

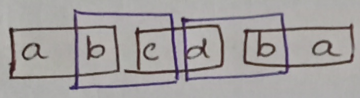
o/p: Pattern found at 2.

⇒ Window size m=2

→ Hash value

pattern : c d

$3\times10^1 + 4\times10^0 = 34$ (Hash value).

text : 

1st coindow ⇒ a b

$34 \neq 12, 34 \neq 23, 34 = 34.$

c = c
d = d } matched.

ab = $1\times10^1 + 2\times10^0 = 12$
bc - 23
cd - 34
db - 42
ba - 21

a b — 1st w
b c - 2nd
c d - 3rd
d b - 4th
b a - 5th

If Hash value is matched, then do character comparision.

⇒ If hash value is matched, then character not matche
It is called as Spurious hit.

Algorithm :-

```
Rabinkarp(T, P)
{
  n = T.length
  m = P.Length
  h^P = hash(P[---]) mod q
  h^T = hash(T[---]) mod q
  for s = 0 to n-m
  if (h^P = -h^T)
   if (P[0--m-1]) = T([s+0---s+m-1])
      print "pattern found with shift s"
      if (s<n-m)
       h^T = hash(T(s+1--- s+m])
}
```

Time Complexity

Best case - O(n-m)

Worst case - O(nm).

06/04/21 Thursday

String matching with finite Automata

text : abcaba ⟹ n

pattern : aba ⟹ m

n>m

States are indicated with circles in finite Automata.

No. of states = No. of symbols in pattern + 1

No. of states = m+1 = 3+1 = 4.

State names are indicated with (q0), (q1), (q2) ....etc

Initial state: from where started

final state: from where stopped

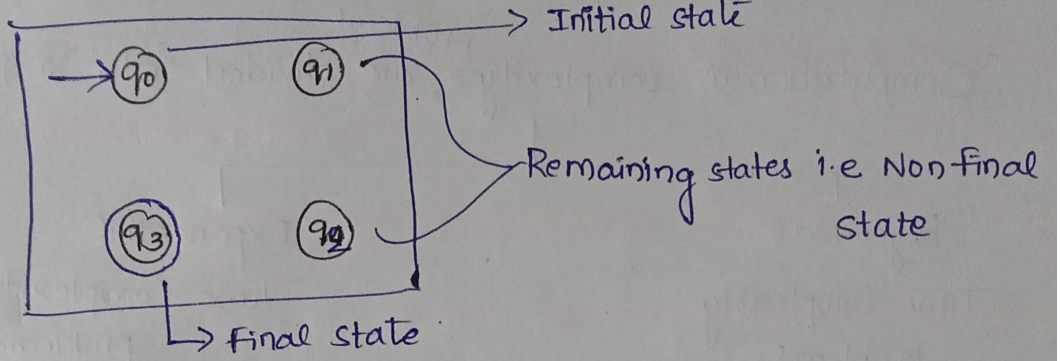
Non final state: Remaining states.

→ Indicated with ⓪

" " →○

⇒ Input set indicated with Σ

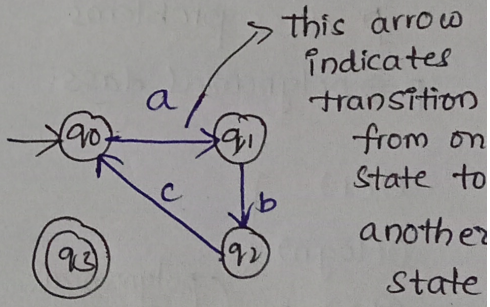
$\Sigma = \{a, b, c\}$



① check whether text char & pat char are matched or not.

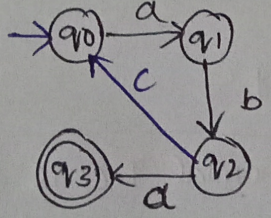
txt: a b c a b a

pat: a b a



⇒ if applied automata is moved to final state ∴ because string is matched.

⇒ here c and a is not matched so, it goes to initial state.

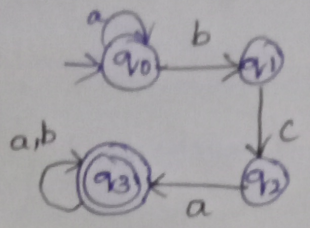


If we are reached to final state, then it is said to be string is matched.

ex: txt: abcabba

pat: bca

Sol:-



=o=

P and NP classes

Computational complexities problems are divided into 2 types

Polynomial Time Complexity problems (Easy problems)

are called as

P-class problems

P → polynomial class.

O(1)
O(log n)
O(n)
O(n²)
O(n³)
⋮
O(nᵏ) k-Constant

} polynomial time

Exponential Time Complexity problems (moderate problems)

are called as

NP-class.

⇒ NP ⇒ Non deterministic polynomial.

O(kⁿ) ⇒ k: constant

O(2ⁿ)
⋮
O(nⁿ)

} Exponential.

Polynomial T.C is smaller than Exponential T.C

O(1) < O(log n) < O(n) < O(n log n) < O(n²) ....

⇒ Exponential problems takes more time.

Eg: All Sorting Algorithms & Dynamic PAll searching Algorithms

(1) Insertion sort
(2) merge sort
(3) Heap sort
(4) Binary Search
(5) Strassen's Matrix multiplication
(6) OBST
(7) Job sequencing

Eg: 1. Travelling Sales man problem.

(2) All the backtracking problems
⇒ n-queens
⇒ Graph Coloring
⇒ Sum of Subsets
⇒ Hamiltonian cycle

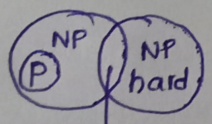
np classes

NP hard → Hardest problems

NP complete → Hardest problems

⇒ If it is both NP and NP hard known as NP complete.

⇒ Relation of



NP complete.

P ⊂ NP

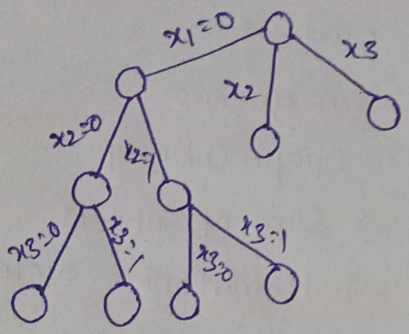
If there exist a.

Initially all deterministic (NP) problems later deterministic algorithm found then it became 'p' problem.

NP hard:-

Inorder to explain NP hard we have to take base problem i.e CNF Satisfying problem.

⇒ Conjunctive Normal form $(A \vee B \vee C) \wedge (A \vee \bar{B} \vee C) \ldots$



x_1	x_2	x_3	x_1 $\bar{x}_2$ x_3
0	0	0	=8 possibilities
0	0	1	
0	1	0	$=2^3$
0	1	1	↓
1	0	0	2^n.
1	0	1	
1	1	0	
1	1	1	

⇓

Similar for all dynamic problems.

⇒ If Satisfiability is reduced to 0/1 Knapsack problem.

Satisfiability $\propto$ 0/1 Knapsack

reduced (NP hard problems)