# ADA ASSIGNMENT

ASHISH KUMAR
2K18/SE/041.

Ques 1) Explain the following in details with the help of example.

a) P problems

b) NP problems

c) NP complete problem

d) NP hard problem

e) Reduction Process in NP completeness Theory.

Ans:- (a) P → polynomial time solving.

Problems which can be solved in polynomial time, which takes time like $O(n)$, $O(n^2)$, $O(n^3)$

eg. finding max^m element in an array, Bubble sort, linear search. There are many problems which can be solved in polynomial time.

(b) NP → Non-deterministic Problem time solving.

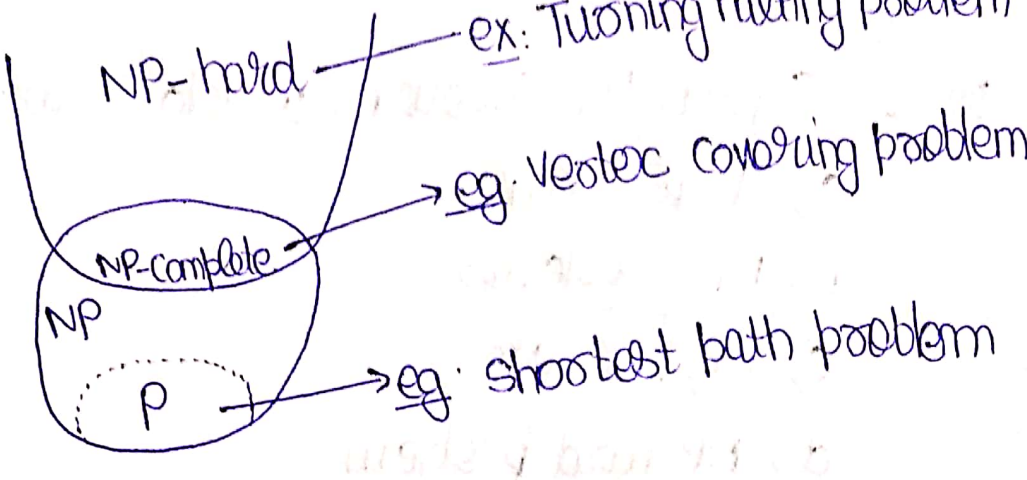
Problems which can't be solved in polynomial time like TSP(travelling salesman problem), subset sum, mergesort.

c) NP-complete problem → are the hardest problems in NP set. A decision problem is NP-complete if:-

(i) if L is NP

(ii) Every problem in NP is reducible to L in polynomial tim

d) NP-hard - A problem is NP-hard if it follows property(ii) mentioned above, doesn't need to follow property(i). Therefore, NP-complete set is also a subset of NP-hard set.




examples of NP complete problems

- Determining whether a graph has a Hamilton cycle.
- Determining whether a boolean formula is satisfiable etc

example of NP-hard problems

The following problems are NP-hard:

- Travelling salesman problem
- Vertex Cover
- The circuit-satisfiability problem.
- K-means clustering

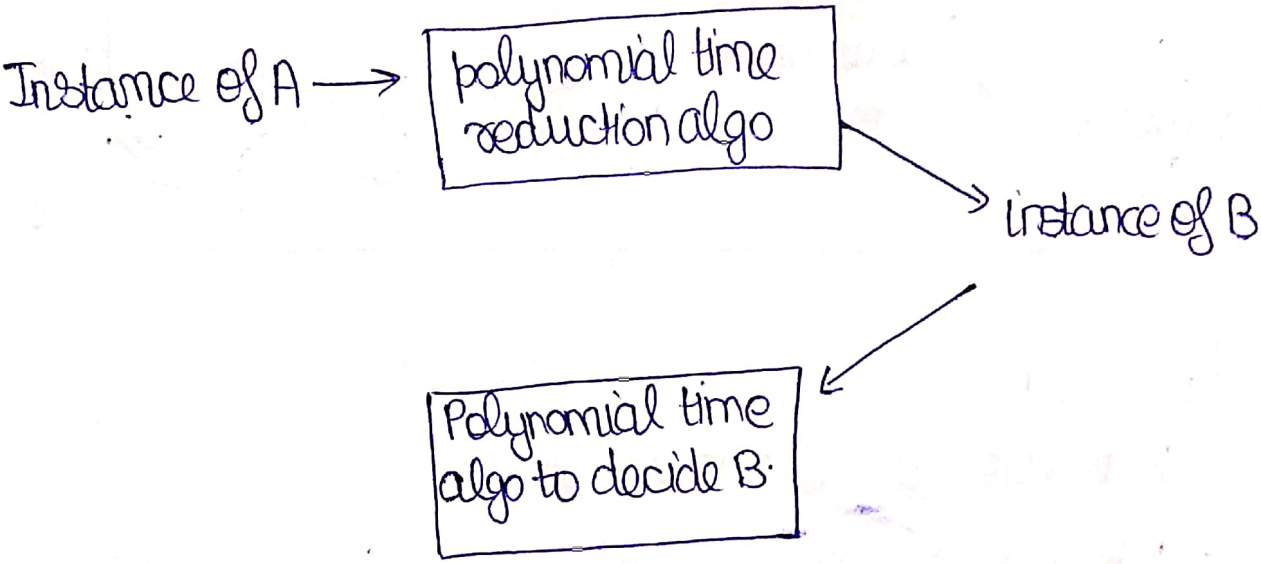
## Conclusion

- P problems are quick to solve
- NP problem are quick to verify but slow to solve.
- NP-complete problem are also quick to verify, slow to solve & can be reduced to any other NP-Complete problem.
- NP-hard problem are slow to verify, slow to solve & can be reduced any other NP-problem.

e) Reduction process :-

Steps :-

1) Given an instance of problem A, use a polynomial time reduction algo to transform it to a problem B.
2) Run the polynomial time decision algo for B as the instance B
3) Use the answer of B as answer of A

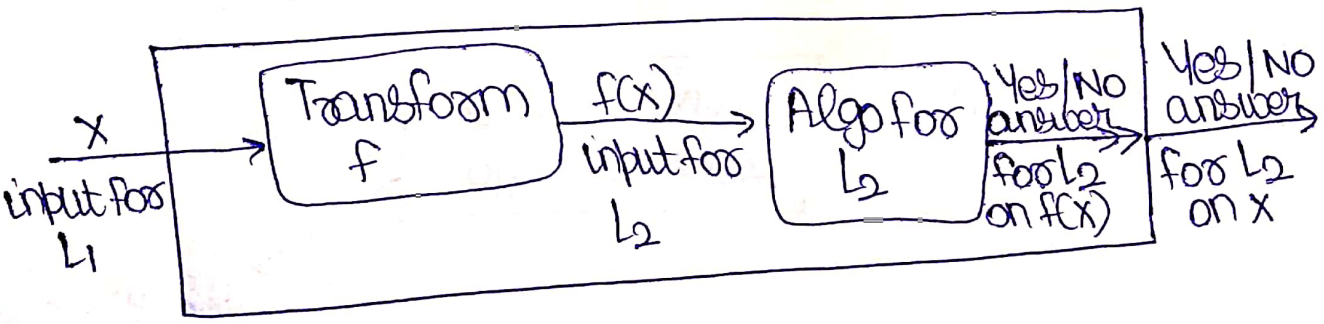



Reduction Process :-

Take two problems A & B, both are NP problems. If we can convert one instance of a problem A into problem B (NP problem), then it means that A is reducible to B.

Let $L_1$ and $L_2$ be two decision problems. Suppose algorithm $A_2$ solves $L_2$. That is, if y is an input for $L_2$ then algo $A_2$ will answer Yes or No depending upon whether y belongs to $L_2$ or not.

The idea is to find a transformation from $L_1$ to $L_2$ so that algo $A_2$ can be reduced to $A_1$ to solve $A_1$

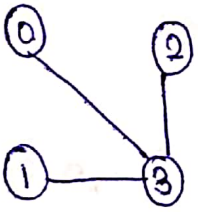


Ques 2) Explain Vertex Cover Problem? Prove it is NP complete problem? Design the approximate soln for the same?

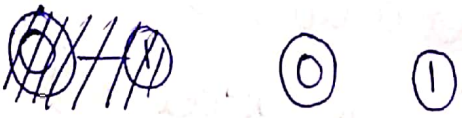
Ans:- Vertex Cover Problem

A vertex cover of an undirected graph is a subset of its vertices such that for every edge (u,v) of the graph, either 'u' or 'v' is in the vertex cover.

Given an undirected graph, the vertex cover problm is to find minm size vertex cover.




Minm vertex cover is {3}




Minm vertex cover is empty.

Vertex Cover problem is a known NP complete problem i.e, there is no polynomial time soln.

Algorithm for vertex cover:-

1. Initialise the result as { } "empty".
2. Consider a set of all edges in given graph. Let the set be E.
3. Do following while E is not empty means untill all edges r covered.
   a) Pick an edge (u,v) from set E & add 'u' and 'v' to result
   b) Remove all edges from E which are either incident on u or v.
4. Return result.

## Proving vertex cover is NP

Given a graph G(V, E) & a +ve integer k, the problem is to find whether there is a subset V' of vertices of size atmost k.

If any problem is in NP, then given a solution to the problem & an instance of the problem. Then from verification algo, we have |V'| = k & then check for each edge (u, v) ∈ E. Thus, it can be verified in polynomial time.

We can check whether the set V' is a vertex cover of size k using the following strategy (for a graph G(V, E)):

1. let count be an integer
2. set count to 0.
3. For each vertex v in V'
   remove all edges adjacents to v from set E
   increment count by 1.
   if count = K & E is empty
   then
      the given solution is coct.
   else
      the given solution is wrong.

## Approximate solution

~~1 c / e~~

① C = ϕ
② E' = G.E
③ while E' ≠ ϕ
④ Let (u, v) be non-arbitary edge of E'
⑤ C U [u, v]
⑥ Remove from E' every edge on either u or v
⑦ Return C.

Ques 3> For the following string-matching Algorithms. Analyze their complexities & explain the working behind them with the help of example?

a) Naive Base Algo

b) Rabin Karp Algo

c) KMP Algo for pattern searching.

Ans:- We formalise the string-matching problem as follows: We assume that text is an array $T[1 \ldots n]$ of length $n$ & that the pattern is an array $P[1 \ldots m]$ of length $m$. The string matching problem is the problem of finding all valid shifts with which a given pattern P occurs in given text T.

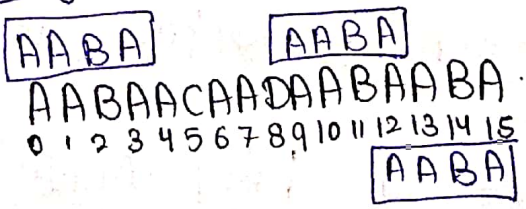
(a) Naive Base Algorithm

I/P :- txt[] = "AABAACAADAABAABA"

pat[] = "AABA"

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

Working




The Naive string Matching Algo slides the pattern one by one. After each slide, it one by one checks characters at the current shift & if all characters match then points the match.

Algorithm

```
n ← length of txt
m ← length of pat

for i ← 0 to n-m
    ~~do if pat[]~~
        for j ← 0 to m
            do if ( txt [i+j] != pat[j])
                break.
            else
                if( j==m)
                then point "Pattern found at" i index;
```

Complexity

Naive string matching takes time $O((n-m+1)m)$ in the worst case. Ex. When all characters of the text & pattern are same.

txt[] = "AAAAAAAA"

pat[] = "AAA.

(b) Rabin Karp Algorithm

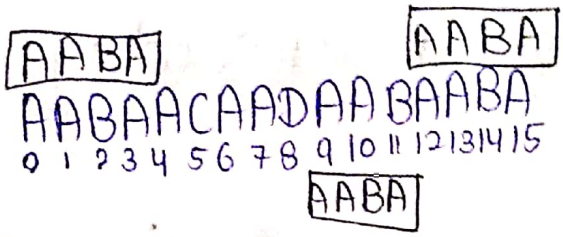
Like Naive Algo, Rabin-Karp Algo also slides the pattern one by one. But unlike Naive Algo, Rabin-karp matches the hash value of the pattern with hash value of current substring of text, & if the hash value matches, then only it starts matching individual characters. So, Rabin-Karp algo needs to calc hash values for pattern itself, All the substrings of text of length m.

I/P txt[] = "AABAACAADAABAABA" pat[] = "AABA"

O/P pattern found at index 0

" " " " 9

" " " " 12




## Algorithm

m ← length of pat
n ← length of txt
p = 0, hash value for pattern
t = 0, " " " txt
d ← 256
q ← 101, A prime no.

First, calc hash value of pattern & first window of text.

```
for i←0 to m
    p = (d*p + pat[i])%q;
    t = (d*t + txt[i])%q;
```

Then slide the pattern over text one by one.

```
for i←0 to n-m
    if(p==t) //check hash values
    { for j←0 to M
        if(j==M)
        then Print "Pattern found at" i index;
    }
then Rehash the function.
```

## Complexity

The avg. & best-case running time of Rabin-Karp algo. is O(n+m), but its worst-case time is O(nm), it occurs when all char. of pattern & text are same as hash values of all substring of txt[] matches with hash value of pat[].

Ex. pat = "AAA"
txt = "AAAAAA"

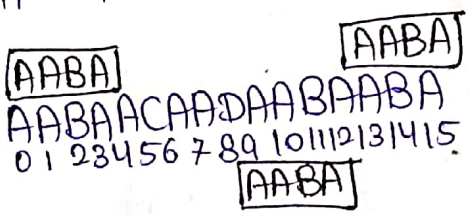
# (C) KMP (Knuth Morris Pratt) Pattern searching

## Working

The KMP algo uses degenerating property (pattern having same sub-patterns appearing more than once in the pattern).
The basic idea behind KMP's algo is: whenever we detect a mismatch (after some matches), we already know some of the characters in the text of the next window. We take advantage of this info" to avoid matching the char that we know.

I/P txt[] = "AABAACAADAABAABA" pat[] = "AABA"

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




## Searching Algorithm

- We start comparison of pat[j] with j=0 with char. of current window of text.
- We keep matching characters txt[i] & pat[j] & keep incrementing i and j while pat[j] & txt[i] keep.
- When we see a mismatch
  - We know that characters pat[0...j-1] match with tx[i-j...i-1].
  - We also know that lps[j-1] is count of characters of pat[0...j-1] that r both prefix & suffix.
  - From above two points, we don't need to match that characters, where we found matched, point that index.

**Complexity:** The complexity of KMP algo in worst case is O(n).