COLLATE

Algorithm Design and Analysis.

Notes

Unit 4

Syllabus

UNIT – IV

String matching: The naïve String Matching algorithm, The Rabin-Karp Algorithm, String Matching with finite automata, The Knuth-Morris Pratt algorithm.

NP-Complete Problem: Polynomial-time verification, NP-Completeness and Reducibility, NP-Completeness Proof, NP –hard ,Case study of NP-Complete problems (vertex cover problem, clique problem).

UNIT-IV

* STRING MATCHING

String matching consists of finding one or more generally, all of the occurences of a pattern in a text. finding a certain pattern in a text is a problem arises in text editing programs & web "surfing".

Given a text array $T[1...n]$, of n characters and a pattern array $P[1...m]$ of m characters the problem is to find an integer s, called a valid shift where $0 \leq s < n-m$ and $T[s+1 .... s+m] = P[1..m]$

In other words, to find whether P in T i.e. P is a substring of T. The elements of P and T are characters drawn from some finite alphabet such as $\{0,1\}$ or $\{A, B, ..Z, a, b, ...z\}$.

* NAIVE-STRING MATCHER

The naive approach simply tests all the possible placement of pattern $P[1...m]$ relative to text $T[1....n]$. Specifically, we try shift $s = 0, 1, ... n-m$, successively & for each shift s. Compare $T[s+1..s+m] = P[1....m]$.

The naive algorithm finds all valid shifts using a loop that checks the condition $P[1 \ldots m] = T[s+1 \ldots s+m]$ for each of the $n-m+1$ possible values of s.

```
NAIVE-STRING-MATCHER(T,P)
1  n ← length[T]
2  m ← length[P]
3  for s ← 0 to n-m
4      if P[1...m] = T[s+1...s+m]
5          print "Pattern occurs with shift" s
```

The for loop beginning on line 3 considers each possible shift explicitly. The test on line 4 determines whether the current shift is valid or not; this test involves an implicit loop to check corresponding character positions until all positions match successfully or a mismatch is found. Line 5 prints out each valid shift s.

The procedure can be interpreted graphically as a sliding pattern $P[1 \ldots m]$ over the text $T[1 \ldots n]$ & noting for which shift all of the characters in the pattern match the corresponding characters in the text.

Implementation of Naive String Algo in order to understand line 4:-

```
NAIVE-STRING-MATCHER(T,P)
n ← length[T]
m ← length[P]
for s ← 0 to n-m do
    j ← 1
    while j ≤ m and T[s+j] = P[j]
        j ← j+1
    if j > m then
        return valid shift s
return no valid shift s.
```

Time complexity:-

first for loop will run at most (n-m+1) times
2nd for loop will run at most m times.
Therefore, the running time of the algorithm is $O((n-m+1)m)$, which is clearly $O(mn)$. Hence, in the worst case, when the length of the pattern, m are roughly equal, this algo. runs in the quadratic time.

* RABIN-KARP ALGORITHM

This algorithm creates a hash value for the pattern, and for each M-character subsequence of text to be compared. If the hash values are unequal, the algorithm will calculate the hash value for next M-character sequence. If the hash values are equal, the algo. will compare the pattern and the M-character sequence. There is only one comparison per text subsequence, & character matching is only needed when hash values match.

Thus, this algo. exploits the fact that if two strings are equal, their hash values are also equal. All we have to do is to compute the hash value of the substring we're searching for, & then look for a substring with the same hash value.

Problem:- If the hash values match, the strings might not match. We have to verify that they do, which can take a long time for long substrings. A good hash function promises us that on most reasonable inputs, this won't happen too often, which keeps the average search time good.

Given:-

a pattern $P[1...m]$ whose decimal value is denoted by p.

a text $T[1...n]$ and let t_s denote the decimal value of the length-m substring $T[s+1...s+m]$ for $s = 0, 1, ..., n-m$.

Certainly, $t_s = p$ if & only if $T[s+1...s+m] = P[1...m]$; thus, s is a valid shift if & only if $t_s = p$.

We can compute p in $O(m)$ using Horner's Rule:-

$$p = P[m] + 10(P[m-1] + 10(P[m-2] + \cdots + 10(P[2] + 10P[1])\cdots))$$

The value t_0 can be similarly computed from $T[1...m]$ in time $O(m)$.

To compute the remaining values $t_1, t_2, ..., t_{n-m}$ in $O(n-m)$, we can compute t_{s+1} from t_s in constant time;

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

Subtracting $10^{m-1}T[s+1]$ removes the high order digit from t_s, multiplying the result by 10 shifts the number left one position, & adding $T[s+m+1]$ brings in the appropriate low order digit.

Solutio

In general, with a d-ary alphabet $\{0, 1, \ldots, d-1\}$, we choose a prime no. q so that dq fits within a computer word.

The only difficulty with this procedure is that p & t_s may be too large to work with conveniently. One solution is to compute p & t_s's modulo a suitable modulus q.

The recurrence equation becomes:-

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \bmod q.$$

where $h \equiv d^{m-1} \pmod q$ is the value of the digit "1" in the high order position of an m-digit text window.

However, the sol^n. of working modulo q is not perfect. Since $t_s \equiv p \pmod q$ does not imply that $t_s = p$. On the other hand, if $t_s \not\equiv p \pmod q$, then we definitely have that $t_s \neq p$, so that shift s is invalid. We use the test $t_s \equiv p \pmod q$ to rule out invalid shifts s. Any shift s for which $t_s \equiv p \pmod q$ must be tested further to see if s is really valid or we just have a spurious hit.

Solution Evaluate

$p = P \bmod q$

$= 26 \bmod 11$

$= 4$

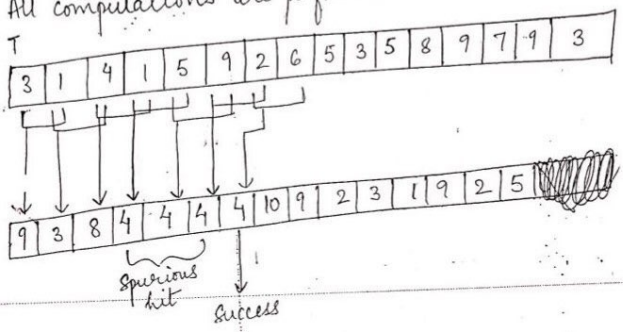
$n = 16$ 79

$m = 2$

s vary from 0 to $n-m$

0 to 14

All computations are performed modulo 11.



3 spurious hits

```
RABIN_KARP_MATCHER (T, P, d, q)
n ← length[T]
m ← length[P]
h ← d^(m-1) mod q
p ← 0
t0 ← 0
for i ← 1 to m
    p ← (dp + P[i]) mod q
    t0 ← (dt0 + T[i]) mod q
for s ← 0 to n-m
    if p = ts
        if P[1...m] = T[s+1...s+m]
            print "Pattern present at shift" s
    if s < n-m
        ts+1 ← (d(ts - T[s+1]h) + T[s+m+1]) mod q
```

Running time = $\theta((n-m+1)m)$.

Example – $q = 11$

$P = 26$

T = 31415926535 89793.

find the no. of spurious hits. = ?

String Matching with finite Automata

String matching automata are very efficient because they examine each text character exactly once, taking constant time per text character.

A finite automaton M is a 5-tuple $(Q, q_0, A, \Sigma, \delta)$, where

(a) Q is a finite set of states
(b) q_0 is the initial state & $q_0 \in Q$
(c) A is the set of accepting states
(d) Σ is a finite input alphabet
(e) δ is a function from $Q \times \Sigma$ into Q called the transition function of M.

The finite automaton begins in state q_0 and reads the characters of its input string one at a time. If the automaton is in state q & reads input character a, then it moves from state q to state $\delta(q, a)$. Whenever its current state q is a member of A, the machine M is said to have accepted the string read so far. An input that is not accepted is said to be rejected.

String matching automata

There is a string matching automaton for every pattern P; this automaton must be constructed from the pattern in a preprocessing step before it can be used to search the text string.

We first define an auxiliary function σ, called the suffix function, corresponding to P. The function σ is a mapping such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x:

$$\sigma(x) = \max\{k : P_k \sqsupset x\}$$

The suffix function σ is well defined since the empty string $P_0 = \varepsilon$ is a suffix of every string.

Example:-

$T = a\ b\ a\ b\ a\ b\ a\ c\ a\ b\ a$

$P = a\ b\ a\ b\ a\ c\ a$

$\sigma(abab) = \sigma(x)$

→ suffix of x in T = Suffix of abab in T

= aba c aba

Prefix of abacaba in P

→ Not present

decrement from right

Prefix of abacab in P

Not present

Prefix of abaca in P (present in P)

= ab

length (ab) = 2

Hence, σ(ababa) = 2

We define the string matching automaton that corresponds to a given pattern P[1...m] as follows:-

(a) The state set Q is {0, 1, ..., m}. Start state q0 is 0 & state m is the only accepting state.

(b) Transition function δ is defined by the following equation, for any state q & character 'a':

$$\delta(q, a) = \sigma(P_q a)$$

Continued...

Question

T = a b a b a b a c a b a
P = a b a b a c a

n = 11
m = 7

Start state = 0
Accepting state = 7
Total No. of states = 8 (0 to 7)
Input alphabets, $\Sigma = \{a, b, c\}$

~~No. of edges per state~~

$\delta(0,a) = \sigma(P_0 a) = \sigma(a)$
$= 1$

$\delta(1,a) = \sigma(P_1 a) = \sigma(aa)$
$= 1$

$\delta(1,b) = \sigma(P_1 b) = \sigma(ab) = 2$

$\delta(2,a) = \sigma(P_2 a) = \sigma(aba) = 3$

$\delta(3,a) = \sigma(P_3 a) = \sigma(abaa) = 1$

$\delta(3,b) = \sigma(P_3 b) = \sigma(abab) = 4$

$\delta(4,a) = \sigma(P_4 a) = \sigma(ababa) = 5$

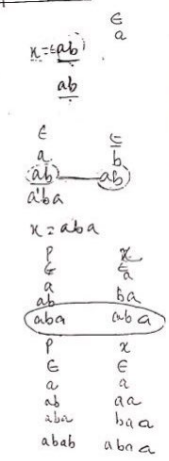
$\delta(5,a) = \sigma(P_5 a) = \sigma(ababaa) = 1$

$\delta(5,b) = \sigma(P_5 b) = \sigma(ababab) = 4$

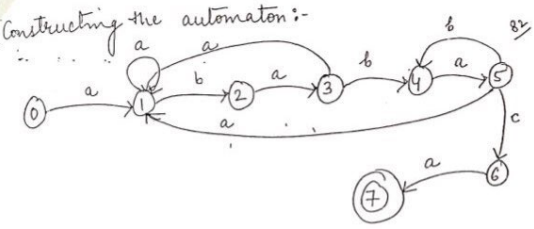
$\delta(5,c) = \sigma(P_5 c) = \sigma(ababac) = 6$

$\delta(6,a) = \sigma(ababaca) = 7.$

	a	b	c
0			
1	1	2	
2	3		
3	1		
4	5		
5	1	4	
6			
7	–	–	–



Constructing the automaton:-



Now,

i →	1	2	3	4	5	6	7	8	9	10	11
T =	a	b	a	b	a	b	a	c	a	b	a
q → 0	1	2	3	4	5	4	5	6	7		

n = 11

accepting state reached.

String match and string found.

after (i-m) = (9-7) = 2 shifts s = 2

FINITE_AUTOMATON_MATCHER(T, δ, S, m) = O(n).

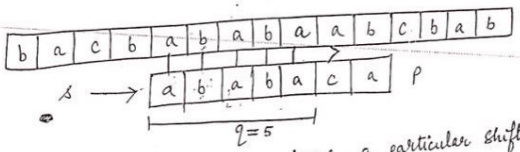
```
n ← length[T]
q ← 0
for i ← 1 to n
    q ← δ(q, T[i])
        if q = m
        then s ← (i-m)
        print "pattern occurs with shift" s.
```

(1) Knuth-Morris-Pratt (KMP) Algorithm.

→ Prefix function for a pattern

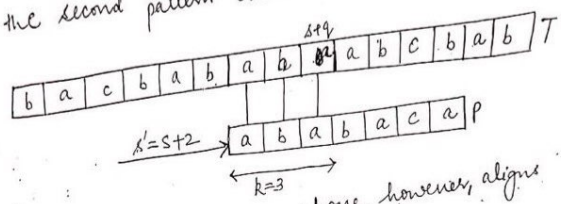
The prefix function π for a pattern encapsulates knowledge about how the pattern matches against ~~itself~~ shifts of it itself. This information can be used to avoid testing useless shifts in the naive pattern matching algorithm.

Consider the operation of the naive string matcher.



As in the example shown above, a particular shift s of a template containing the pattern $P = abababaca$ matches against a text T. For this example, $q = 5$ of the characters have matched successfully, but the 6th pattern character fails to match the corresponding text character. The information that q characters have matched successfully determines the corresponding text characters. Knowing ~~these~~ these q characters allows us to determine immediately

that certain shifts are invalid. In the example of the figure, shift s+1 is invalid since the first pattern character (a) would be aligned with a text character that is known to match with the second pattern character (b).



The shift $s' = s+2$ shown above, however, aligns the first three pattern characters with three text characters that must necessarily match.

In general, it is required to know the answer to the following question:-

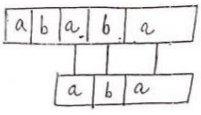
Given that pattern characters $P[1 \ldots q]$ match text characters $T[s+1 \ldots s+q]$, what is the least shift $s' > s$ such that

$$P[1 \ldots k] = T[s'+1 \ldots s'+k],$$

where

$$s' - s = q - k$$

or $s' + k = q + s$



The useful information can be precomputed using the pattern itself. We see that the longest prefix of P that is also a proper ~~prefix~~ suffix of P_7 is P_3. This info. is precomputed & represented in the array π, so that $\pi[5] = 3$.

Given that q characters have matched successfully at shift s, the next potentially valid shift is at

$s' = s + (q - \pi[q])$.

The precomputation is formalized as follows:- Given a pattern $P[1..m]$, the prefix function for the pattern P is the function $\pi : \{1, 2, \ldots m\} \to \{0, 1, \ldots m-1\}$ such that

$$\pi[q] = \max\{k : k < q \text{ and } P_k \sqsupset P_q\}$$

That is, $\pi[q]$ is the length of the longest prefix of P that is a proper suffix of P_q.

KMP_

```
COMPUTE_PREFIX_FUNCTION(P)
m ← length[P]
π[1] ← 0
k ← 0
for q ← 2 to m
    while k > 0 and P[k+1] ≠ P[q]
        k ← π[k]
    if P[k+1] = P[q]
        k ← k+1
    π[q] ← k
return π.
```

Example:-

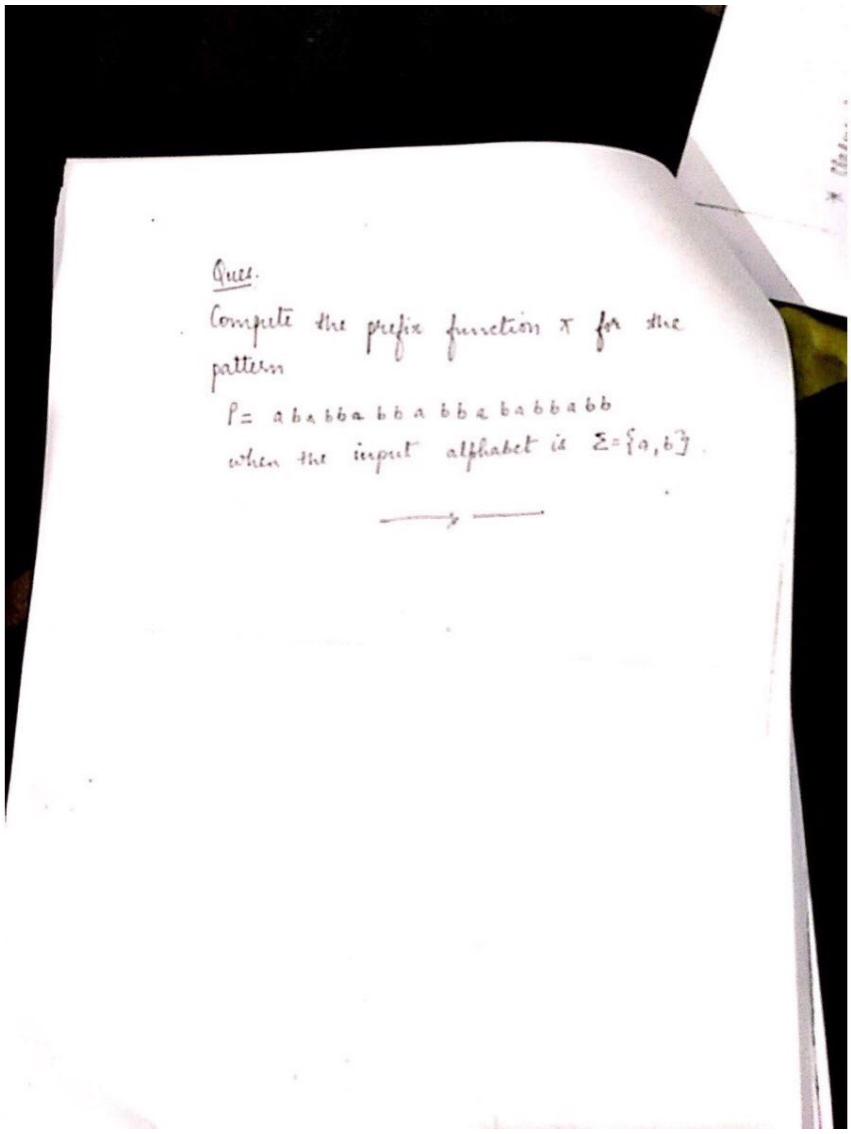
```
KMP_MATCHER(T, P)
n ← length(T)
m ← length(P)
π ← COMPUTE_PREFIX_FUNCTION(P)
q ← 0
for i ← 1 to n do
    while q > 0 and P[q+1] ≠ T[i]
        q ← π[q]
    if P[q+1] = T[i]
        q ← q+1
    if q = m
        Print "Pattern occurs at shift" i-m
        q ← π[q]
```

Running time analysis

The running time of COMPUTE_PREFIX_FUNCTION is $\Theta(m)$.

Running time of KMP_MATCHER is $\Theta(n)$.

Hence, total running time is $\Theta(m+n)$



Ques.

Compute the prefix function π for the pattern

$P = a\,b\,a\,b\,b\,a\,b\,b\,a\,b\,b\,a\,b\,a\,b\,b\,a\,b\,b$

when the input alphabet is $\Sigma = \{a, b\}$.

* Classes of Problems :-

(a) Polynomial Time Algorithms (P Class) - Almost all the algorithms we have studied so far have been polynomial time algorithms on input of size n, their worst case running time is $O(n^k)$ for some constant k (easy/tractable)

(b) NP Class - There are problems which cannot be solved in polynomial time i.e. not solvable in $O(n^k)$ for some constant k (hard/intractable)

(c) NP-Complete - These are the set of problems whose status is unknown. No polynomial time algorithm has yet been discovered for an NP-complete problem, nor has anyone yet been able to prove that no polynomial time algorithm can exist for any one of them. This so called P≠NP question has been one of the deepest research questions.

An interesting fact about the NP-complete problems is that several of them seem on the surface to be similar to problems that have polynomial time algorithms.

Example - Shortest vs longest simple paths - Even with negative weights, we can find shortest paths from a

single source in a directed graph $G=(V,$ in $O(VE)$ time finding the longest simple path between two vertices is NP-Complete, however.

NP Completeness & the classes P and NP.

The class P consists of those problems that are solvable in polynomial time. They are problems that can be solved in $O(n^k)$ for some constant k, where n is the size of the input to the problem. Most of the problems examined in previous chapters are in P.

The class NP consists of those problems that are "verifiable" in polynomial time. Verifiable means that if we were somehow given a "certificate" of a solution, then we could verify that the certificate is correct in time polynomial in the size of the input to the problem. Example- in the hamiltonian cycle problem, given a directed graph $G=(V,E)$, a certificate would be a sequence $\langle v_1, v_2, \ldots v_{|V|}\rangle$ of $|V|$ vertices. It is easy to check in polynomial time that $\langle v_i, v_{i+1}\rangle \in E$ for $i = 1, 2, 3, \ldots |V|-1$ & that $\langle v_{|V|}, v_1\rangle \in E$ as well.

Any problem in P is also in NP, since if a problem is in P then we can solve it in polynomial time without even being given a certificate.

Note :- If any NP-Complete problems can be solved in polynomial time, then every NP Complete problem has a polynomial time algorithm.

→ Overview of showing problems to be NP-Complete

① Decision problems vs. optimization problems.

optimization problems have a solution which has an associated value, and we wish to find the feasible solution with the best value. Example, in a problem that we call SHORTEST-PATH, we are given an directed graph G & vertices u and v, and we wish to find the path from u to v that uses the fewest edges.

NP-Completeness applies directly not to optimization problems, however, but to decision problems, in which the answer is simply "yes" or "no" (or, more formally, "1" or "0").

The relationship b/w an optimization problem & its related decision problem works when we try to show that the optimization problem is "hard". That is because the decision problem is in a sense "easier" or at least "no harder". In other words, if an optimisation problem is easy, its related decision problem is easy as well. If we can provide evidence that a decision problem is hard, we also provide evidence that its related optimisation problem is also hard. Thus, even though it reflects attention to decision problems, the theory of NP-completeness often has implications for optimisation problems as well.

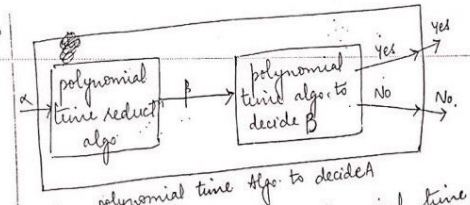
② Reductions

Let us consider a decision problem, say A, which we would like to solve in polynomial time. We call the input to a particular problem an instance of that problem; for example, in PATH, an instance would be a particular graph G, particular vertices u and v of G, & a particular integer k.

Now suppose there is a different decision problem, say B, that we already know how to solve in

polynomial time. Finally, suppose we have a 88 procedure that transforms any instance α of A into some instance β of B with the following characteristics:-

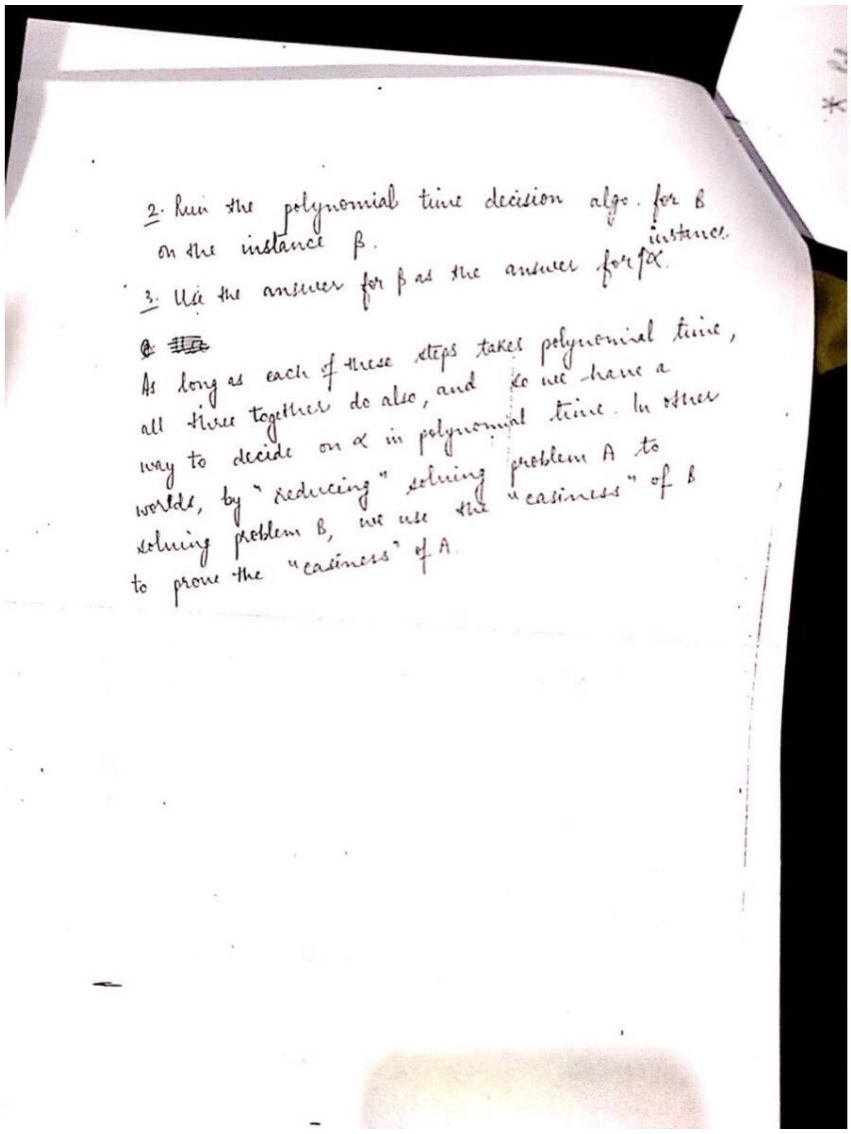
1. Transformation takes polynomial time.
2. The answers are the same. that is, the answer for α is 'yes' if & only if the answer for β is also 'yes'.



polynomial time algo. to decide A

Such a procedure is called polynomial time reduction algorithm - and it provides us a way to solve the problem A in polynomial time:-

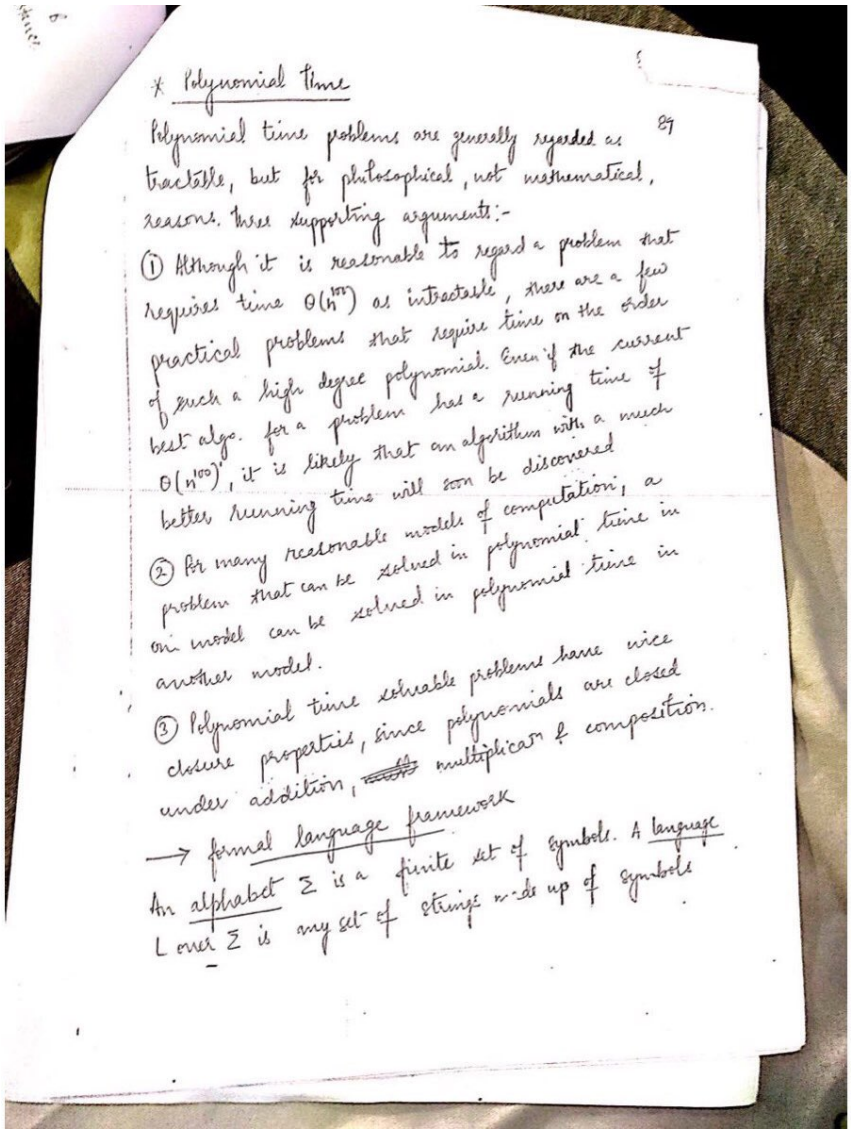
(a) Given an instance α of problem A, use a polynomial time reduction algorithm to transform it to an instance β of problem B.



2. Run the polynomial time decision algo. for B on the instance β.

3. Use the answer for β as the answer for α.

As long as each of these steps takes polynomial time, all three together do also, and so we have a way to decide on α in polynomial time. In other words, by "reducing" solving problem A to solving problem B, we use the "easiness" of B to prove the "easiness" of A.



* Polynomial time

Polynomial time problems are generally regarded as tractable, but for philosophical, not mathematical, reasons. Three supporting arguments:-

(1) Although it is reasonable to regard a problem that requires time $\Theta(n^{100})$ as intractable, there are a few practical problems that require time on the order of such a high degree polynomial. Even if the current best algo. for a problem has a running time of $\Theta(n^{100})$, it is likely that an algorithm with a much better running time will soon be discovered

(2) For many reasonable models of computation, a problem that can be solved in polynomial time in one model can be solved in polynomial time in another model.

(3) Polynomial time solvable problems have nice closure properties, since polynomials are closed under addition, multiplication & composition.

→ formal language framework

An alphabet Σ is a finite set of symbols. A language L over Σ is any set of strings made up of symbols.

from Σ Example- if Σ={0,1}, the set L={10, 11, 101, 111, 1011, 1101, 10001, ...} is the language of binary representations of prime nos. We denote the empty string by ∈ & the empty language by ϕ.

Example If Σ={0,1}, then Σ*={0, 1, 01, 00, 10, 11, 000, ...} is the set of all binary strings. Every language L over Σ is a subset of Σ*

Operations on languages

1. Union
2. Intersect^n
3. Complement of L by $\bar{L} = \Sigma^* - L$
4. Concatenat^n of 2 languages L_1 & L_2

$$L=\{x_1x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}.$$

5. Closure / Kleene star of a language L is a language

$$L^* = \{\in\} \cup L \cup L^2 \cup L^3 \cup \ldots$$

where L^k is the language obtained by concatenating L to itself k times.

from Σ. Example- if Σ={0,1}, the set L={10, 11, 101, 111, 1011, 1101, 10001, ---} is the language of binary representations of prime nos. We denote the empty string by ∈ & the empty language by ϕ.

Language of all strings over Σ is denoted as Σ^*.

Example if Σ={0,1}, then Σ^*={0, 1, 01, 00, 10, 11, 000, ---} is the set of all binary strings. Every language L over Σ is a subset of Σ^*.

Operations on languages

1. Union
2. Intersect^n
3. Complement of L by $\bar{L} = \Sigma^* - L$
4. Concatenat^n of 2 languages L_1 & L_2.

$$L = \{x_1 x_2 : x_1 \in L_1 \text{ and } x_2 \in L_2\}.$$

5. Closure / Kleene star of a language L is a language

$$L^* = \{\in\} \cup L \cup L^2 \cup L^3 \cup \ldots$$

where L^k is the language obtained by concatenating L to itself k times.

A language L is accepted in polynomial time by an algo. A if it is accepted by A & if in addition there is a constant k such that for any length-n string $x \in L$, algorithm A accepts x in time $O(n^k)$.

A language L is decided in polynomial time by an algo. A if there is a constant k such that for any length-n string $x \in \{0,1\}^*$, the algorithm correctly decides whether $x \in L$ in time $O(n^k)$.

Thus, to accept a language, an algo. need only worry about strings in L, but to decide a language, it must correctly accept or reject every string in $\{0,1\}^*$.

Polynomial time verification

→ Verification Algorithms

A verification algo. is a two argument algorithm A, where one argument is an ordinary input string x & the other is a binary string y called a certificate. A two argument algo. A verifies an input

string x if there exists a certificate y such that $A(x,y) = 1$.

The language verified by a verification algorithm A is

$L = \{x \in \{0,1\}^* : \text{there exists } y \in \{0,1\}^* \text{ such that } A(x,y) = 1\}$.

Hence, an algo. A verifies a language L if for any string $x \in L$, there is a certificate y that A can use to prove $x \in L$. Moreover, for any string $x \notin L$, there should be no certificate proving that $x \in L$.

Example - Suppose a given graph is Hamiltonian and you are given a set of vertices in order along the hamiltonian cycle.

It is certainly easy enough to verify the proof :- simply verify that the provided cycle is hamiltonian by checking whether it is a permutation of the vertices of V & whether each of the consecutive edges along the cycle actually exists in the graph. This verification algo. can certainly be implemented to run in $O(n^2)$ time. Thus, a hamiltonian cycle exists can be verified in polynomial time.

* NP-Completeness and Reducibility

→ Reducibility

A problem Q can be reduced to another problem Q' if any instance of Q, can be "easily rephrased" as an instance of Q', the solution to which provides a solution to the instance of Q.

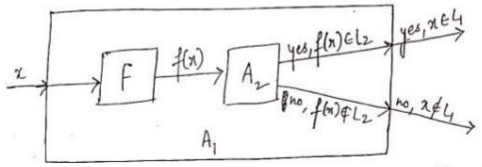
Example - the problem of solving linear equations in an indeterminate x reduces to the problem of solving quadratic equations. Given an instance $ax+b=0$, we transform it to $0x^2+ax+b=0$, whose sol^n. provides a sol^n to $ax+b=0$. Thus, if a problem Q reduces to a problem Q', then Q is, in a sense, "no harder to solve" than Q'.

A language L_1 is polynomial time reducible to a language L_2, written $L_1 \leq_p L_2$ if there exists a polynomial time computable function $f: \{0,1\}^* \to \{0,1\}^*$ such that for all $x \in \{0,1\}^*$,

$x \in L_1$ if only if $f(x) \in L_2$.

This function f is called as the reduction function and a polynomial time algo. that computes f

is called a reduction algorithm.



Algo. F is a reduction algo. that computes the reduction function f from L_1 to L_2 in polynomial time, and A_2 is a polynomial time algo. that decides L_2. A_1 decides whether $x \in L_1$ by using F to transform any input x into f(x) & then using A_2 to decide whether $f(x) \in L_2$

→ NP-Completeness

If $L_1 \leq_p L_2$, then L_1 is not more than a polynomial factor harder than L_2, which is why "less than or equal to" notation for reduction is mnemonic. time ↑

Definition of NP complete languages :-

A language $L \subseteq \{0,1\}^*$ is NP-Complete if :-

① $L \in NP$, and

② $L' \leq_p L$ for every $L' \in NP$.

if a language satisfies property ... necessarily property 1, we say that ... is NP-hard.

---

* MATROID (UNIT-3)

A ~~matroid~~ matroid is an ordered pair $M = (S, \mathcal{I})$ satisfying the following conditions:-

(1) S is a finite nonempty set

(2) $\mathcal{I}$ is is nonempty family of subsets of S, called the independent subsets of S, such that if $B \in \mathcal{I}$

Content Feedback

Feel free to submit your issues, suggestions and ratings regarding this course content

Click Here

Join & Share WhatsApp Group of COLLATE

(Click the button to join group)

MSIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

MAIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

ADGITM (NIEC)

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

GTBIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

BVP

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	

BPIT

Semester 3	CSE	IT	ECE
Semester 5	CSE	IT	