UNIT V

Reference Book.

- T.H Coreman "Introduction to Algorithms"
- S. Sahni "Fundamentals of Computer Algo"

Algebric Computation

Fast Fourier Transform (FFT)

String Matching

- Naive Algo
- Rabin - Karp Algo
- Finite Automata
- Knuth - Morris - Pratt Algo
- Boyer - Moore Algo

Theory of NP - Completeness

Approximation Algorithms.

- Vertex Cover problem
- Travelling Salesman problem.

Randomized Algorithm.

ASSIGNMENT 5

REFERENCE BOOK

Algebric Computation	–	S. SAHNI
Fast Fourier Transform	–	S. SAHNI
Approximation Algo	–	T. H. COREMAN
Randomize Algo	–	T. H COREMAN.

STRING MATCHING

- String matching consists of finding one or more generally, all of the occurances of a pattern in a text.
  Finding a certain pattern in a text is a problem arises in text-editing programs and web surfing.
- In computer science, string searching algorithms, sometimes called string matching algorithms that try to find a place where one or several string are found within a larger string or text.
- Problem is to find whether a pattern(P) is a substring of a Text(T). The elements of P and T are character drawn from some finite alphabet such as {0, 1} or {A, B, ... Z, a, b ... z.}
- There are many types of string matching algorithms
  1. Naive string matching Algorithm
  2. Rabin-Karp-Algorithm
  3. Finite Automata
  4. Knuth Morris Pratt Algorithm
  5. Boyer-Moore Algorithm.

NAIVE STRING MATCHING

- This algo test 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] to P[1..m].
- Algo finds all valid shift using a loop that checks the condition P[1...m] = T[s+1 ... s+m] for each of the n-m+1 possible values of s
- ALGO

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

RABIN-KARP-ALGO

- String searching algo created by Michael O. Rabin & Richard M Karp that seeks a pattern i.e a substring, within a text by using hashing.
- It calculates a hash value for the pattern, & for each M-character subsequence of text to be compared.
  If hash values are unequal, the algo will calculate the hash value for next M-character sequence
  If hash values are equal, the algo will compare the pattern & the M-character sequence.
  In this way, there is only one comparison per text subsequence, & character matching is only needed when hash values match.

- Given a pattern $P[1 \ldots m]$, we let p denote its corresponding decimal value.
  Similarly, given a text $T[1 \ldots n]$, we let t_s denote the decimal value of the length m- substring $T[s+1 \ldots s+m]$ for $s = 0, 1 \ldots n-m$.
  Certainly $t_s = p$ if and only if $T[s+1 \ldots s+m] = P[1 \ldots m]$;
  Thus, s is a valid shift if and only if $t_s = P$. If we could compute P in time $\Theta(m)$ and all of the t_i values in a total of $\Theta(n)$ time, then we could determine all valid shifts s in time $\Theta(n)$ by comparing P with each of the t_s.
  We can compute P in $\Theta(m)$ time.

$$P = P[m] + 10(P[m-1] + 10(P[m-2] + \ldots + 10(P[2] + 10\,P[1])\ldots))$$

  The value t_0 can be similarly computed from $T[1 \ldots m]$ in $\Theta(m)$.
  It computes remaining values $t_1, t_2, \ldots t_{n-m}$ in $O(n-m)$ time.

- **ALGO**
  1. Compute P mod q
  2. Compute $(T(s+1, s+2, \ldots s+m) \bmod q)$ for $s = 0, 1, \ldots n-m$.
  3. Test against P only those sequences in T having same (mod q) value.
  4. $(T\{s+1, s+2, \ldots s+n\} \bmod q)$ can be incrementally computed by subtracting the higher-order digit, shifting, adding the low-order digit, all in modulo q arithmetic.

STRING MATCHING WITH FINITE AUTOMAIA

String-matching automata are very efficient because they examine each text character exactly once, taking constant time per text character. The time used after the automaton is built is therefore $\theta(n)$.

The time to build the automaton, however can be large if Σ is large.

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

- Q : finite set of states
- q_0 : start state $q_0 \in Q$
- A : distinguished set of accepting state $A \subseteq Q$
- Σ : finite input alphabet.
- δ : Transition function of M.

- Finite Automaton begins in state q_0 & reads the characters of its i/p string one at a time.
- If the automaton is in state q & reads i/p char a, 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 i/p that is not accepted is said to be rejected.
- In order to specify the string-matching automaton corresponding to a given pattern $P[1 \ldots m]$, we first define an auxiliary function σ, called the suffix function corresponding to P.

The function σ is a mapping from Σ^* to $\{0, 1, \ldots, m\}$ such that $\sigma(x)$ is the length of the longest prefix of P that is a suffix of x:

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

- We define the string-Matching automaton corresponding to a given pattern $P[1 \ldots m]$ as follows.

The state set Q is $\{0, 1 \ldots m\}$. The start state q_0 is state 0, & state m is the accepting state.

- The transition fun^n δ is defined by the following equat^n, for any state q & character a:

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

P.T.O

```
FINITE_AUTOMATON_MATCHER(T, δ, m)
{ 1. n ← length [T]
  2. q ← 0
  3. for i ← 1 to n
  4.     do q ← δ(q, T[i])
  5.         if q = m
  6.             then s ← i - m
  7.                 print "Pattern occurs with shift" s
}
```

- Computing transition function δ from a given pattern P[1....m]

```
COMPUTE_TRANSITION_FUNCTION (P, Σ)
{ 1. m ← length [P]
  2. for q ← 0 to m
  3.     do for each character a ∈ Σ
  4.     do k ← min(m+1, q+2)
  5. repeat k ← k - 1
  6. until δ(q,a) ← k
  7. return δ
}
```

V.V. Jmp.

KNUTH MORRIS PRATT (KMP) ALGORITHM

- Knuth, Morris and Pratt discovered first linear time string matching algo by following a tight analysis of the naive algo.
- KMP algo keeps the "informat" that naive approach wasted gathered during the scan of the text.
- By avoiding waste of "info", it achieves running time of $O(n+m)$, which is achieved by using the auxiliary function π.
- In worst case KMP algo examine all the character in the text & pattern at least once.
- function π computes the shifting of pattern matches within itself.
- Basic idea is to slide the pattern towards the right along the string so that the longest prefix of P that we have matched matches the longest suffix of T that we have already matched.
- If the longest prefix of P that matches a suffix of T is nothing then we slide the whole pattern towards right.
- ALGO for COMPUTE-PREFIX-FUNCTION (P)

```
{ 1. m ← length[P]
  2. π[1] ← 0
  3. k ← 0
  4. for q ← 2 to m
  5.    do while k>0 & P[k+1] ≠ P[q]
  6.         do k ← π[k]
  7.    if(P[k+1] = P[q])
  8.       then k ← k+1
  9.       π[q] ← k
 10. return
}
```

- KMP matching algo is given as the procedure KMP-MATCHER. KMP-MATCHER( ) calls the auxiliary procedure COMPUTE-PREFIX FUNCTION( ) to compute π.

P.T.O →

KMP

- ALGO

KMP_MATCHER (T,P)

```
{1. n ← length[T]
2. m ← length[P]
3. π ← COMPUTE-PREFIX-FUNCTION(P)
4. q ← 0
5. for i ← 1 to n
6.     do while q>0 & P[q+1] ≠ T[i]
7.         do q ← π[q]
8.     if(P[q+1] = T[i])
9.       then  q ← q+1
10. if(q=m)
11.   then print "pattern occurs with shift" i-m
12. q ← π[q]
}
```

Q. Compute the Prefix fun' π for pattern ababbabbabbababbabb when the alphabet is Σ = {a,b}

length of pattern P

Soln Here the length of Pattern P = 19 so m = 19

Initially π[1] = 0 k = 0 q = 2 P[k+1] ≠ P[q] ∴ π[2] = 0

Now P[k+1] = P[q] for q = 3 then k = 1 & π[3] = 1

if k > 0 & P[k+1] = P[q] for q = 4 & k = 2 ∴ π[4] = 2

k > 0 & P[k+1] ≠ P[q] q = 5 k = π[2] = 0 π[5] = 0

Similarly we compute other functions

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
P[i]	a	b	a	b	b	a	b	b	a	b	b	a	b	a	b	b	a	b	b
π[i]	0	0	1	2	0	1	2	0	1	2	0	1	2	3	4	5	6	7	8

BOYER - MOORE ALGO

- It is considered the most efficient string matching algo in usual applications.
  ex: text editors and commands substitutions.
- Algo was developed by Bob Boyer & J Strother Moore in 1977.
- Algo scans the characters of the pattern from right to left begining with the rightmost character, which is also referred to as "looking Glass Heuristic".
- If text symbol that is compared with the rightmost pattern symbol does not occur in the pattern at all, then the pattern can be shifted by m positions behind this text symbol.
- For determining the smallest possible shifts, Boyer Moore works with two different preprocessing strategies.
- Each time when a mismatch occurs the algo computes both & chooses the largest possible shift. Thus making use of the most efficient strategy for each individual case.
  - Bad character Heuristics
  - Good Suffix Heuristics

- ALGO

BOYER_MOORE_MATCHER (T, P, Σ) // O((n-m+1)m + |Σ|)

```
{ 1. n ← length [T]
  2. m ← length [P]
  3. λ ← COMPUTE-LAST-OCCURENCE-FUNCTION (P, m, Σ)
  4. γ ← COMPUTE-GOOD-SUFFIX-FUNCTION (P, m)
  5. s ← 0
  6. while (s ≤ n-m)
  7.     do j ← m
  8.        while (j > 0 and P[j] = T[s+j])
  9.           do j ← j - 1
 10.        if (j=0)
 11.           print "pattern occur at shift" s
 12.           s ← s + γ[0]
 13.        else s ← s + max(γ[j], j - λ[T[s+j]])
}
```

- COMPUTE-LAST-OCCURRENCE-FUNCTION (P, m, Σ) // O(m + |Σ|)

```
{ 1. for each character a ∈ Σ
  2.     do λ[a] = 0
  3. for j ← 1 to m
  4.     do λ[P[j]] ← j
  5. return λ
}
```

- COMPUTE-GOOD-SUFFIX-FUNCTION (P, m) // O(m)

```
{
1.  π ← COMPUTE-PREFIX-FUNCTION (P)
2.  P' ← reverse (P)
3   π' ← COMPUTE-PREFIX-FUNCTION (P')
4.  for j ← 0 to m
5.      do γ[j] ← m - π[m]
6.  for l ← 1 to m
7.      do j ← m - π'[l]
8.          if γ[j] > l - π'[l]
9.              then γ[j] ← l - π'[l]
10. return γ
}
```

- Boyer-Moore algo (like the Rabin Karp algo) spens O(m) time validating each valid shifts.

THEORY OF NP-COMPLETENESS.

- Almost all the algorithms we have studied thus far have been polynomial time algo:
  i.e on some i/p of size n, their worst case running time is $O(n^k)$ for some constant k.

- All the problems can not be solved in polynomial time.
  ex: there are problems, such as Turing famous "HALTING PROBLEM" that can't be solved by any computer, no matter how much time is provided.

- There are also problems that can be solved, but not in time $O(n^k)$ for any constant k.

- NP complete problem is a class of problems whose status is unknown.
  No polynomial-time algo has yet been discovered for an NPC prob, nor has anyone yet been able to prove that no polynomial time algo can exist for any one of them.

  This so called P ≠ NP question has been one of the deepest most perplexing open research problem in theoretical computer Science since 1971.

- In each of the following pair of problems, one is solvable in polynomial time and other is NP-Complete, but the difference between problems appears to be slight.

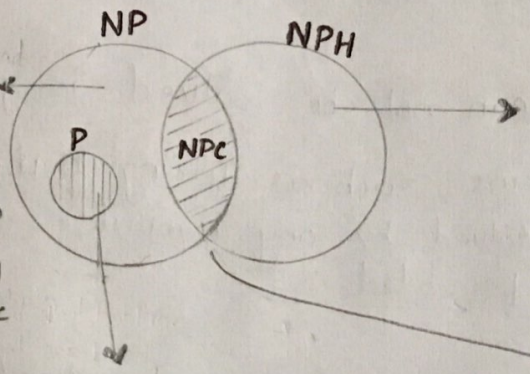
  a.) Shortest Vs. Longest simple paths.
  finding shortest path in directed graph $G=(V,E)$ takes $O(VE)$ time
  finding Longest path b/w two vertices in NPC.

  b.) Euler Tour Vs. Hamiltonian Cycle.

  c.) 2CNF-SAT Vs 3CNF-SAT

- There are diff. types of problem in Decidability theory

1. P- Problem
2. NP Problem
3. NP- H Problem



- Include both Good & bad problems.
- May not be solved in polynomial time on deterministic Automaton but solution can be verified in polynomial time on Deterministic Automaton.
- following prob are NP
  - $P \subseteq NP$
  - Primality
  - Factoring
  - Discreate log
  - Graph isomorphism
  - $NPC \subseteq NP$
  - Compositness prob.

- If $\forall Y \in NP$, $Y \leq_p X$ then X is called NPH prob.
- following are NPH prob
  - Turing Machine Halting Prob.
  - All approximation prob
  - All NPC prob.
  - Many prob on General Graph

- Include Good problems
- Computable by deterministic algo in polynomial time
- following prob are P prob
  - 1-SAT
  - 2-SAT
  - Shortest path
  - Euler cycle
  - Equivalence of DFA
  - Matrix Multiplicat"
  - Sorting, path prob & relative primality prob.
  - Parsing
  - Linear programming.

- A prob X is NPC iff $\forall Y \in NP$, $Y \leq_p X$, $X \in NP$
- Hardest problem in NP is called as NP-complete prob
- following prob are NPC prob.
  - 3-SAT prob
  - Traveling Salesman prob
  - Hamiltonian Circuit
  - Vertex Cover
  - Chromatic No: prob
  - LCS
  - Clique Prob
  - Subgraph isomorphism.