B.E.

Fifth Semester Examination, May-2008

Analysis & Design of Algorithms(CSE-305-E)

Note : Attempt any five questions. All questions carry equal marks.

Q. 1. (a) Write short notes on asymptotic notations. **6**

Ans. Asymptotic Notation : The notations describe the asymptotic running time of an algorithm are defined in terms of functions whose domains are the set of natural numbers N = {0, 1, 2, ... }. There are different types of notations :

(i) θ -Notation : The worst-case running time of insertion sort is $T(n) = \theta(n^2)$. For a given function g(n), we denote by $\theta(g(n))$ the set of functions.

$\theta(g(n)) = \{f(n)$: there exist positive constants C_1, C_2 and no such that $0 \le C_1 g(n) \le f(n) \le C_2 g(n)$ for all $n \ge n_o\}$. The definition of $\theta(g(n))$ requires that every member $f(n) \in \theta(g(n))$ be asymptotically non-negative, that is, that f(n) be non-negative wherever n is sufficiently.

(ii) O-Notation : The θ-notation asymptotically bounds a function from above and below. When we have only an asymptotic upper bound, we use O-notation. For a given function g(n), we denote by O(g(n)) the set of functions.

$O(g(n)) = \{$ f(n) : there exist positive constant C and no such that $O \le f(n) \le Cg(n)$ for all $n \ge n_o\}$.

(iii) Ω-notation : Ω-notation provides an asymptotic lower bound. For a given function g(n), we denote by $\Omega(g(n))$ the set of functions.

$\Omega(g(n)) = \{f(n)$: there exist positive constants C and no such that $O \le Cg(n) \le f(n)$ for all $n \ge n_o\}$.

Q. 1. (b) Explain the procedure Quick-sort with an example. Also analyse it in best, average and worst cases. **14**

Ans. Quick-Sort : Quick sort processes a very good average-case behaviour among all the sorting techniques. This is developed by C. A. R. Hoare.

The Quicksort algorithm works by partitionary the array to be sorted. And each partition is in turn sorted recursively. In partition, one of the array elements is chosen as a key value. This key value can be the first element of an array. That is, if a is an array then key = a[0]. And rest of the array elements are grouped into two partitions such that

- One partition contains elements smaller than the key value.
- Another partition contains elements layer than the key value.

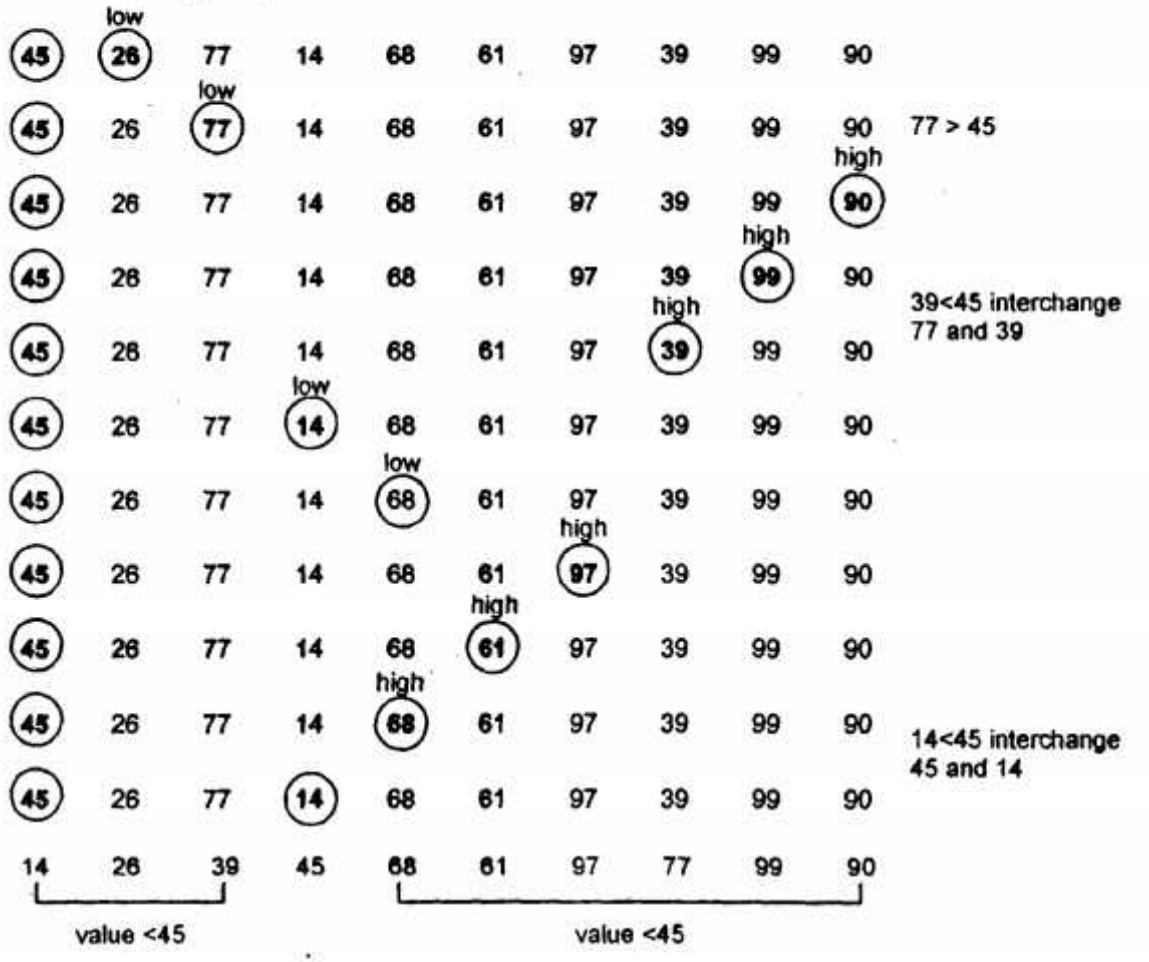
Example :

45 26 77 14 68 61 97 39 99 90

Here, 45 is selected as a key value. Then, two indices namely, low and high are used to indicate the element (key + 1) and the last element. The low index starts on the left and selects an element that is greater than the key value. Then these elements are interchanged. The process is repeated until all

elements to the left of the key are smaller than the key value. And all element to the right of the key are greater than the key value.

The steps involved in placing the key value, 45 in its proper position in the array are given below. The elements being compared are encircled on each lines.



The given array has been partitioned into two subarrays. The first subarray is { 14, 26, 39 } and the second is {68, 61, 97, 77, 99, 90}. We can repeatedly apply this procedure on each of these subarrays until the entire array is sorted.

Efficiency of Quick-Sort : The timing analysis of quicksort algorithm is

$$T(n) = \begin{cases} a & n=1 \\ Cn + T(n/2) + T(n/2) \end{cases}$$

Cn → time required to partition the array

T (n/2) → time required to sort the left or right subarray.

Worst-Case Analysis : In this case, on every function call, the given array is partitioned into two subarrays.

$$
\begin{aligned}
T(n) &= Cn + T(0) + T(n-1) \\
&= Cn + T(n-1) \\
&= Cn + C(n-1) + T(n-2) \\
&= Cn + C(n-1) + C(n-2) + T(n-3) \\
&\quad \vdots \\
&= Cn + C(n-1) + C(n-2) + \ldots\ldots + C(1) + \rceil (0) \\
&= C[n + (n-1) + (n-2) + \ldots\ldots\ldots + 2 + 1] \\
&= C\left[\frac{n(n+1)}{2}\right] \\
&= \frac{Cn^2}{2} + \frac{Cn}{2} = O(n^2)
\end{aligned}
$$

Best-Case Analysis : The best case timing analysis is possible when the arrary is always portitioned in half

$$
\begin{aligned}
T(n) &= Cn + T(n/2) + T(n/2) \\
&= Cn + 2T(n/2)
\end{aligned}
$$

Let n is power of 2 i.e., $n = 2^k$

Thus,

$$
\begin{aligned}
T(2^k) &= 2T(2^{k/2}) + C(2^k) \\
&= 2T(2^{k-1}) + C.2^k \\
&= 2\left[2T(2^{k-2}) + C2^{k-1}\right] + C.2^k \\
&= 2^2 T(2^{k-2}) + 2C2^k \\
&= 2^2\left[2T(2^{k-3}) + C2^{k-2}\right] + 2C^k \\
&= 2^3 T(2^{k-3}) + 3C2^k \\
&= 2^k T(2^{k-k}) + KC2^k \\
&= 2^k T(1) + k\,Cn
\end{aligned}
$$

Q. 3. (a) Describe the Greedy method and Divide-AND-Conquer strategy to find optimal solution to the problem. To which type of problems they are applied ? **10**

Ans. Greedy Method : A greedy algorithm obtains one optimal solution to a problem by making a sequence of choices. For each decision point in the algorithm, the choice that seens best at the moment is chosen. This heuristic strategy does not always produce an optimal solution, but as we saw in the activity-selection problem, sometimes it does. To develop a greedy algorithm, we consider the following steps :

(i) Determine the optimal substructure of the problem.

(ii) Develop a recursive solution.

(iii) Prove that at any stage of the recursion, one of the optimal choices is the greedy choice. Thus, it is always safe to make the greedy choice.

(iv) Show that all but one of the subproblems induced by having made the greedy choice are empty.

(v) Develop a recursive algorithm that implements the greedy strategy.

(vi) Convert the recursive algorithm to an iterative algorithm.

We design greedy algorithms according to the following sequence of steps :

(i) Cost the optimization porblem as one in which we make a choice and are left with one subproblem to solve.

(ii) Prove that there is always an optimal solution to the original problems that makes the greedy choice, so that the greedy choice is always safe.

(iii) Demonstrate that, having made the greedy choice, what remains is a subproblem with the property that if we combine an optimal solution to the subproblem with the greedy choice we have made, we arrive at an optimal solution to the original problem.

Divide-and-Conquer Approach : Many useful algorithms are recursive in structure to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach : they break the problem into several subproblems that are similar to the original problem but smaller in size, solve that subproblems recursively and than combine these solution to create a solution to the original problem.

The divide-and-conquer paradigm involves three steps at each level of the recursion :

Divide : The probelm into a number of subproblems.

Conquer : The subproblems by solving them recursively. In the subproblem sizes are small enough, just solve the subproblems in a straightforward manner.

Combine : The solutions to the subproblems into the solution for the original problem.

The fractional knapsack problem is solvable by or greedy strategy. To solve the fractional problem, we first compute the value per pound v_i/w_i for each item. Obeying a greedy strategy, the thief begins by

taking as much as possible of the item with the greatest value per pound. If the supply of that item is exhausted and he can still carry more, he takes as much as possible of the item with the next greatest value per pound and so forth until he can't carry any more.

Q. 3. (b) Explain Strassen's matrix multiplication with suitable example. **10**

Ans. Strassen's Matrix Multiplication : Strassen's algorithm is used for multiplying n×n matrices, which run in $\theta(n^{\lg 7}) = O(n^{2.81})$ time. For sufficiently large values of n, therefore, it out performs the naive $\theta(n^3)$ natrix-multiplication algorithm, MATRIX-MULTIPLY.

Strassen's algorithm can be viewed as an application of a familiar design technique : divide and conquer. Suppose we wish to compute the product C = AB, where each of A, B and C are n×n matrices. Assuming that n is an exact power of 2, we divide each of A, B and C into four n/2×2/2 matrices, rewriting the equation C = AB as follows.

$$\begin{pmatrix} r & s \\ t & u \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} e & f \\ g & h \end{pmatrix} \quad \text{..... (i)}$$

Equation (1) corresponds to the four equations

$$r = ae + bg,$$
$$s = af + bh,$$
$$t = ce + dg,$$
$$u = cf + dh$$

Each of these four equations specifies two multiplication of n/2×n/2 matrices and the addition of their, n/2×n/2 products. Using these equations to define a straightforward divide-and-conquer strategy, we drive the following recurrence for the time T(n) to multiply two n×n matrices :

$$T(n)\ 8T(n/2) + \theta(n^2)$$

Strassen's method has four steps :

(i) Divide the input matrices A and B into n/2 × n/2 submatrices.

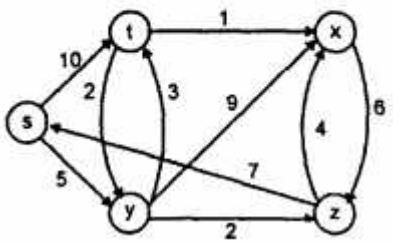
(ii) Using $\theta(n^2)$ scalar additions and subtractions, compute 14 matrices $A_1, B_1, A_2, B_2,$ ..... A_7, B_7 each of which is n/2 × n/2.

(iii) Recursively compute the seven matrix products

$$P_i = A_iB_i \text{ for } i=1, 2, \ldots\ldots, 7.$$

(iv) Compute the desired submatrices r, s, t, u of the result matrix C by adding and/or subtracting various combinations of the P_i matrices. Using only $\theta(n^2)$ scalar additions and subtractions.

Q. 4. (a) Use Dijkstra's algorithm to find the single source shortest paths for the following graph taking vertex 's' as the source. **10**



Ans. Dijkstra's Algorithm : Dijkstra's algorithm solves the single-source shartest-paths problems on a weighted, directed graph $G = (V, E)$ for the case in which all edge weighs are non-negative.

Dijkstra's algorithm maintains a set of S of vertices whose final shortest-path weights form the sources. That is, for all vertices $V \in S$, we have $d[v] = \delta(s, v)$. The algorithm repeatedly selects the vertex $v \in V-S$ with the minimum shortest path estimate, inserts v into S and relaxes all edges leaving μ in the following implementation.

Dijkstra (G, w, S) :

(i) Initialize Single-Source (G, S)

(ii) $S \leftarrow \phi$

(iii) $\theta \leftarrow$ [G]

(iv) While $\theta \neq \phi$

(v) Do $\mu \leftarrow$ EXTRACT_MIN (θ)

(vi) $S \leftarrow S \cup [\mu]$

(vii) For each v (i.e. $v \in$ Adj [μ]

(viii) Do RELAX (μ, v, w)

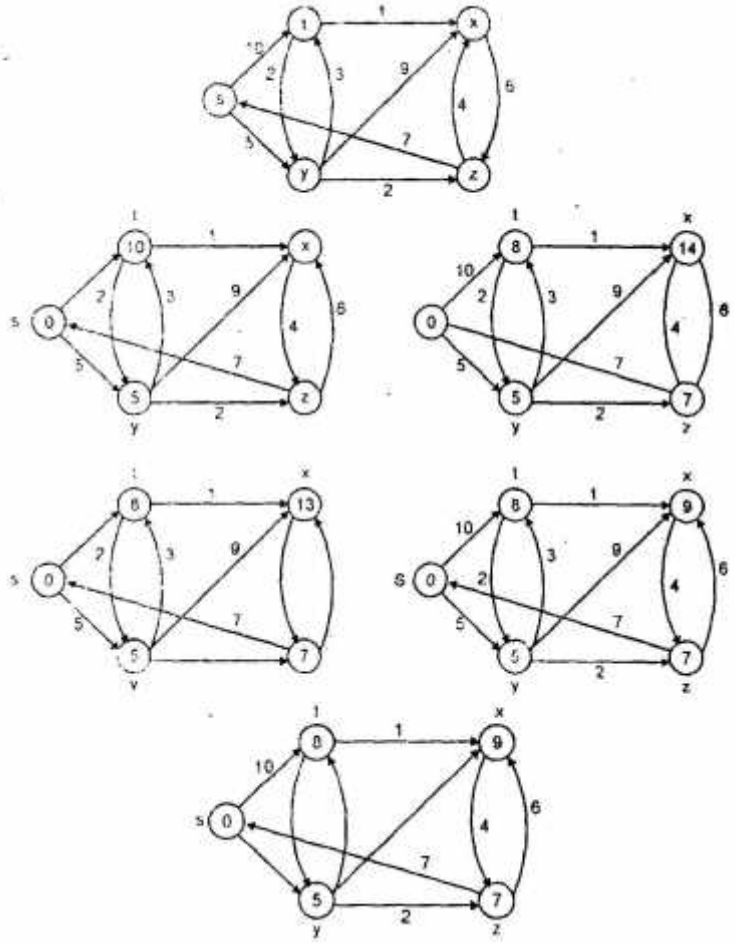
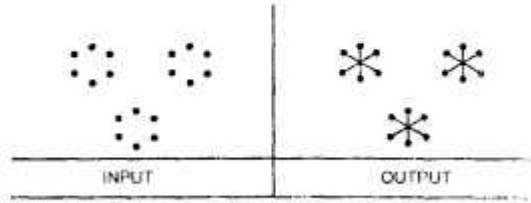


Figure shows the execution of Dijkstra's algorithm. The source is the leftmost vertex. The shortest-path estimates are shown within the vertices and shaded edges indicate predecessor values : if edge (t, x) is shaded, the π (x) = t. Black vertices are in the set S and white vertices are in priority queue θ = V − S (a). The situation just before the first iteration of the while loop of lines 4–8. The shaded vertex has the minimum d-value and is chosen as vertex t in line 5(b)-(f). The situation after each successive



iteration of the while loop. The shaded vertex in each part is chosen as vertex 't' in line 5 of the next iteration. The d and π values shown in part (f) are the final values.

Q. 4. (b) Write any one algorithm to find minimum spanning tree of the graph. Take suitable example to explain it. 10

Ans. Minimum Spanning Tree :

Input Description : A graph G = (V, E) with weighted edges.

Probelm : The subset of E of G of minimum weight form a tree on V.

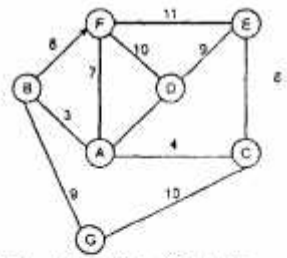
Excerpt From : The Algorithm Design Manual : The minimum spanning tree (MST) of a graph defines the cheapest subset of edges that keeps the graph in one connected component. Telephone companies are particularly interested in minimum spanning trees, because the minimum spanning tree of a set of sites defines the wiring scheme that connects the sites using as little wire as possible.

A tree T is a spanning tree of a connected graph G (V, E) such that

(i) every vertex of G belongs to an edge in T and

(ii) The edges in T form a tree.

Figure represent the feasible communication lines between 7 cities and the cost of an edge could be



interpreted as the actual cost of building that link (in lakhs of rupees). The minimal spanning tree problem for this situation could be the building of a least cost communication network. MST are defined only on undirected graphs.

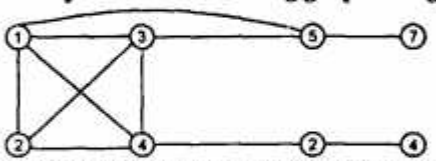
Q. 5. (a) Write backtracking algorithm to find the chromatic number of a given graph. 10

Ans. Backtracking : The backtracking method is based on the systematically in question of the possible solution where through the procedure, set of possible solutions are rejected before even examined, So their number is getting a lot smaller. An important requirement which must be fulfilled is that there must be the proper hierarchy in the systematically procedure of solutions so that sets of

solutions that do not fulfil a certain requirement are rejected before the solutions are produced. For this reason the examination and produce of the solutions, follows a model of non-cycle graph for which in this case we will consider as a tree. The root of the tree represents the set of all the solutions. Nodes in lower level represents even smaller sets of solutions, based on their properties.

Common use of backtracking is in path finding algorithms where function traces once a graph of nodes and backtracks until it finds the least cost path. Backtracking is used in the implementation of P.L. (such as Icon, Planner and Prolog) and other areas such as text passing.

Q. 5. (b) Find all Hamiltonian cycles in the following graph using backtracking. **10**



Ans. Hamiltonian Cycles : A Hamiltonian cycle of a directed graph G = (V, E) is a simple cycle that contains each vertex in V. Determining whether a directed graph has a Hamiltonian cycle is MP-complete.

The problem of finding a Hamiltonian cycle in an undirected graph has been studied for over a hundred years. Formally, a Hamiltonian cycle of an undirected graph G = (V, E) is a simple cycle that contains each vertex in V. A graph that contains a Hamiltonian cycle is said to be Hamiltonian; otherwise, it is non-Hamiltonian.

We can define the Hamiltonian-cycle problem, "Does a graph G have a Hamiltonian cycle ?" as a formal language :

$$HAM\_CYCLE = \{ (G) : G \text{ is a Hamiltonian graph}\}$$

How might an algorithm decide the language HAM_CYCLE ? Given a problem instance (G), one possible decision algorithm lists all permutations of the vertices of G and then checks each permutation to see if it is a Hamiltonian path. Whta is the running time of this algorithm ? If we use the "reasonable" encoding of a graph as its adjacency matrix, the number m of vertices in the graph is $\Omega(\Omega_n)$, where $n = |(G)|$ is the length of the encoding of G. There are m! possible permutation of the vertices and therefore the running time is $\Omega(m!) = \Omega(\sqrt{n'}) \mid - \Omega(2\sqrt{n})$, which is not $O(n^k)$ for any constant K. Thus, this naive algorithm does not run in polynomial time.

Q. 6. Explain Dynamic Programming method to solve a problem and use the same method to solve the following Travelling-salesperson Problem with given distance matrix : **20**

$$\begin{bmatrix} 0 & 5 & 10 & 15 \\ 3 & 0 & 6 & 8 \\ 4 & 12 & 0 & 11 \\ 7 & 7 & 8 & 0 \end{bmatrix}$$

Ans. Dynamic programming like the divide-and-conquer method, solves problems by combining the solutions to subproblems. A dynamic-programming algorithm solves every subproblem just once and then saves its conquer in a table, thereby avoiding the work of recomputing the conquer every time the sub subproblem is encountered.

Dynamic programming is an algorithm that solves the problem of matrix-chain multiplication. We are given a sequence (chain) $(A_1, A_2, \ldots\ldots, A_n)$ of n matrices to be multiplied and we wish to compute

$A_1 A_2 \ldots\ldots A_n$

We can evaluate the expression $A_1 A_2 \ldots\ldots A_n$ using the standard algorithm for multiplying pairs of matrices as a subroutine once we have parenthesized it to resolve all ambiguities in how the matrices are multiplied together. A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses. Matrix multiplication is associative and so all parenthesizations yield the same product. For example, if the chain of matrices is (A_1, A_2, A_3, A_4), the product $A_1 A_2 A_3 A_4$ can be fully parenthesized in five distinct way :

$(A_1 (A_2 (A_3 A_4)))$

$(A_1 (A_2 A_3) A_4))$

$(A_1 A_2) (A_3 A_4))$

$((A_1 (A_2 A_3)) A_4)$

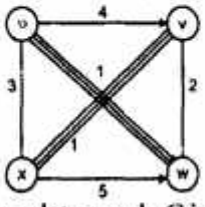
$(((A_1 A_2) A_3) A_4)$

The matrix-chain multiplication problem can be stated as follows : given a chain $(A_1, A_2, \ldots\ldots, A_n)$ of n matrices, where for $i = 1, 2, \ldots\ldots, n$, matrix A_i has dimension $P_{i-1} \times P_i$, fully parenthesize the product $A_1 A_2 \ldots\ldots A_n$ in a way that minimizes the the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost. Typically, the time invested in determining this optimal order is more than paid for by the time saved later or when actually performing the matrix multiplication.

Travelling-Salesman Problem : Travelling-salesman problem is the problem of determining the shortest closed tour that connects a given set of n points in the plane. Figure shows the solution to a 7-point problem. The general problem is NP-complete and its solution is therefore b.

In the travelling-salesman problem, a salesman must visit n cities. Modelling the problem as a complete graph with n vertices, we can say that the salesman wishes to make a tour, or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is an integer cost c(i, j) to travel from city i to city j and the salesman wishes to make the tour whose total cost is minimum, where the total cost is the sum of the individual costs along the edges of the tour, for example a minimum-cost tour is (v, w, x, μ) with cost 7. The formal language for the corresponding decision problem is



TSP = {(G, C, K) : G = (V, E) is a complete graph, C is a function from $V \times V \rightarrow Z$, $K \in Z$, and G has a travelling salesman tour with cost at most k}.

Q. 8. Write short notes on the following : **20**

(a) Differentiate between deterministic and non-deterministic algorithms

(b) Cook's theorem

(c) NP-Hard and NP-Complete problems

Ans. (a) Difference between Deterministic and Non-deterministic Algorithms : Algorithms such that the result of every operation is uniquely defined are called deterministic algorithms.

A non-deterministic algorithm is an algorithm with one or more choice points where multiple different continuations are possible, without any specification of which one will be taken. A shopping list can be viewed as very simple non-deterministic algorithm. Every item on the list is a directive to find the indicated product, but the order in which to find them is not indicated.

A deterministic algorithm is an algorithm which, in formal terms, behaves predictably. Given a particular input, it will always produce the same output, and the underlying machine will always pass through the same sequence of states.

One simple model for deterministic algorithms is the mathematical function; just as a function always produces the same O/P given a certain input, so do deterministic algorithm.

(b) Cook's Theorem : In computational complexity theory, the Cook-Levin theorem, aslo known as Cook's theorem. States that the Boolean satisfiability problem is NP-complete. That is, any problem in NP can be reduced in polynomial time by a deterministic turing machine to a problem of determining whether a Boolean formula is satisfiable.

An important consequence of the theorem is this : If there were a deterministic polynomial time algorithm for solving Boolean satisfiability, then there would exists a deterministic polynomial time algorithm for solving all problems in NP.

Cook's Theorem :

$f(x) = 1$ iff SAT $(F(x)) = 1$

That (a) holds has already been shown. Now Let f be any decision problem in NP. We know that f also belongs to ADA-NP. So there is a non-deterministic Ada program NA_f for f and a polynomial p(n) such that, for any x in $\{0, 1\}^n$, $f(x) = 1$ iff

NCOMP $(NA_f\ x, p(n)) = 1$

To show that SAT is NP-complete, we build an instance of SAT, H (NA_f) $(X_{m(n)})$ with the property

H (NA_f) $(X_{m(n)})$ is satisfiable

iff

NOCM P $(NA_f\ x, p(n)) = 1$

(c) **NP-Hard and NP-Complete Problems** : NP-hard and NP-complete problem polynomial time reductions provide a formal means for showing that one problem is at least as hard as another, to within a polynomial-time factor. That is, if $L_1 \leq pL_2$, then L_1 is not more than a polynomial factor harder them L_2, which is why the "less than or equal to" notation for reduction is mnemonic.

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

(i) $L \in NP$, and

(ii) $L' \leq_p L$ for every $L' \in NP$.

If a language L satisfies property 2, but not necessarily property 1, we say that 1, is NP-Hard. We also define NPC to be the class of NP-complete languages .