B.E.

Fifth Semester Examination, December-2007

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

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

Q. 1. (a) Differentiate between space and time complexity of an algorithm. 6

Ans. An algorithm is a set of rules for carrying out calculations either by hand or on a machine. Algorithm is a branch of Computer Science that consists of designing and analyzing computer algorithms:

(i) The "design" pertains to :

(a) The description of algorithm at an abstract level by means of a pesudo language and

(b) Proof of correctness that is, the algorithm solves the given problem in all cases.

(ii) The "analysis" deals with performance evaluation **(Complexity analysis)**.

Algorithm Analysis : The complexity of an algorithm is a function g(n) that gives the upper bound of the number of operations (or running time) performed by an algorithm when the input size is n.

In general, the time taken by an algorithm grows with the size of the input, so it is traditional to describe the running time of a program as a function of the size of its input. To do so, we need to define the terms "running time" and "size of input".

The best notion for **input size** depends on the problem. For many problems, such as sorting or computing discrete Fourier transforms, the most natural measure is the number of items in the input, for example, the array size n for sorting.

The **running time** of an algorithm on a particular input is the number of primitive operations or "steps" executed. It is convenient to define the notion of step so that it is as machine–independent.

Q. 1. (b) Explain Merge-Sort with an example. Also analyze it in best, average and worst cases.

Ans. Merge-Sort is a sorting technique which divides the array into subarrays of size 2 and merge adjacent pairs. We then have approximately n/2 array of size 2. This process is repeated until there is only one array remaining of size n.

Suppose an array a with n elements a[1], a[2], ..., a[N] is in memory. The merge-sort a will first be described by means of a specific example.

Example : Suppose the array a contains 12 elements as follows :

85, 76, 46, 92, 30, 41, 12, 19, 93, 3, 50, 11

Each pass of the merge sort algorithm will start at the beginning of the array A and merge pairs of sorted subarrays as follows :

Pass I : Merge each pair of elements to obtain the following list of sorted pairs :

76	85	46	92	30	41	12	19	3	93	11	50

Pass II : Merge each pair of pairs to obtain the lists of sorted elements.

46	76	85	92	12	19	30	41	3	11	50	93

Pass III : Again merge the two subarrays to get two lists :

12	19	30	41	46	76	85	92

3	11	50	93

Pass IV : Merging the above two lists, we get

3	11	12	19	30	41	46	50	76	85	92	93

In Pass IV, we get sorted list of elements of array a with the size 12.

Q. 2. (a) Write algorithms for Union and find operations for disjoint sets. **10**

Ans. A disjoint-set data structure maintain a collection $S = \{S_1, S_2, ..., S_K\}$ of disjoint dynamic sets. Each set is identified by a representative, which is some member of the set. In some applications, it doesn't matter which member is used as the representative; we only care that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times. In other applications, there may be a prespecified rule for choosing the representative, such as choosing the smallest member in the set.

Union (x, y) unites the dynamic sets that contain x and y, say S_x and S_y into a new set that is the union of these two sets. The two sets are assumed to be disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x and S_y as the new representative. Since, we require the sets in the collection to be disjoint, we "destroy" sets S_x and S_y, re g them from the collection 's'.

Algorithm (UNION) :

UNION (x, y)

(i) LINK (FIND_SET (x), FIND_SET (y))

FIND_SET (x) returns a pointer to the representative of the (unique) set containing x.

Algorithm :

FIND_SET (x)

(i) If $x \neq p[x]$

(ii) Then $p[x] \leftarrow$ FIND_SET (p [x])

(iii) Return p [x]

The FIND_SET procedure is a two-pass method : It makes one pass up the find path to find the root and it makes a second pass back down the find path to update each node so that it points directly to the root. Each call of FIND_SET (x) return p [x] in line 3. If x is the root, then line 2 is not executed and p [x] = x is returned.

Q. 2. (b) Explain Binary search with suitable example and find its complexity in best, average and worst cases. **10**

Ans. Binary search is an extremely efficient algorithm. This search technique searches the given item in minimum possible comparisons. To do the binary search, first we had to sort the array elements. The logic behind this technique is given below :

(i) First find the middle element of the array.

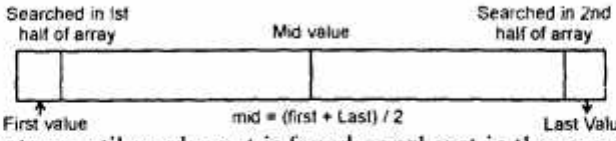
(ii) Compare the mid element with an item.

(iii) There are three cases.

(a) If it is a desired element then search is successful.

(b) If it is less than desired item then search only the first half of the array.

(c) If it is greater than the desired element search in the second half of the array.



Repeat the same steps until an element is found or exhaust in the search area. In this algorithm every time we are reducing the search area. So, number of comparisons keep on decreasing. **In worst case the number of comparison is at most log (N + 1).** So, it is an efficient algorithm when compared to linear search but the array has to be sorted before doing binary search.

Suppose we have an array a of 7 elements.

0	1	2	3	4	5	6
9	12	24	30	36	45	70

The steps to search 45 using binary search in array a [7] are :

Step (i) : The given array is in ascending order; item to be searched for is 45.

beg = 0 , Last = 6

mid = int ((beg + last)/2) = int (6/2) = 3

beg	0	1	2	3	4	5	6	last
	9	12	24	30	36	45	70	

Step (ii) : a [mid] i.e., a [3] is 30

30 < 45 then

beg = mid + 1 = 3 + 1 = 4

Step (iii) : mid = int ((beg + last)/2) = int (4 + 6)/2 = 5

beg	4	5	6	last
	36	45	70	

a [mid] i.e., a [5] is 45

45 = 45

Search successful !! At location number (element number 6).

Q. 3. (a) Explain 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. The Divide and Conquer Strategy : 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 sub-problems. These algorithms typically follow a divide-and-conquer approach : They break the problem into several sub-problems that are similar to the original problem but smaller in size, solve the sub-problems recursively and then combine these solutions to create a solution to the original problem.

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

Divide the problem into a number of sub-problems.

Conquer the sub-problems by solving them recursively.

If the sub-problem sizes are small enough, however, just solve the sub-problems in a straight forward manner.

Combine the solutions to the sub-problems into the solution for the original problem.

The Greedy Strategy : A greedy algorithm obtains an optimal solution to a problem by making a sequence of choice. For each decision point in the algorithm, the choice that seems 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.

Generally, we design greedy algorithms accordings to the following sequence of steps :

(i) Cast the optimization problem as one in which we make a choice and are left with one sub-problem to solve.

(ii) Prove that there is always an optimal solution to the original problem 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 sub-problem with the property that if we combine an optimal solution to the sub-problem with the greedy choice we have made we arrive at an optimal solution to the original problem.

The greedy-choice property and optimal sub-structure are the two key ingredients, which solve a particular optimization problem.

Greedy-Choice Property : The first key ingredient is the greedy-choice property : A globally optimal solution can be arrived at by making a locally optimal (greedy) choice. In other words, when we are considering which choice to make, we make the choice that looks best in the current problem, without considering results from sub-problems.

The greedy choice property often gains us some efficiency in making our choice in a sub-problem. For example, in the activity-selection problem, assuming that we had already sorted the activities in monotonically increasing order of finish times, we needed to examine each activity just once. It is frequently the case that by preprocessing the input or by using an appropriate data structure, we can make greedy choices quickly, thus yielding an efficient algorithm.

Optimal Substructure : A problem exhibits optimal substructure if an optimal solution to the problem contains within it optimal solution to sub-problems. This property is a key ingredient of assessing the applicability of dynamic programming as well as greedy algorithms. This scheme implicitly uses induction on the sub-problems to prove that making the greedy choice at every step produces an optimal solution.

Q. 3. (b) How we can insert an element in Binary Search Tree ? Write an algorithm and give suitable example. **10**

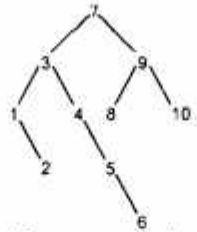
Ans. Binary Search Tree : A binary search tree is a binary tree which is either empty or satisfies the following rules :

(i) The value of the key in the left child or left subtree is less than the value of the root.

(ii) The value of the key in the right child or right subtree is more than or equal to the value of the root.

(iii) All the sub-trees of the left and right children observe the two rules.

Example :



The number 7 is the root node of the binary tree. It has two sub-trees, the left subtree with node 3 and right subtree with node 9. The value of left subtree node is lower than the value of the root and the value of the right subtree node is higher than the value of the root. This attribute is seen in all the down-below nodes to the left and right of the root.

Insertion of Nodes (Algorithm) : To insert a new value v into a binary search tree T, we use the procedure TREE_INSERT. The procedure is passed a node z for which key [z] = v, left [z] = NIL and right [z] = NIL. It modifies T and some the fields of z in such a way that z is inserted into an appropriate position in the tree.

```
    TREE_INSERT (T, z)
1.    y ← NIL
2.    x ← root[T]
3.      While x ≠ NIL
4.      do y ← x
5.      if key[z] < key [x]
6.      then x ← left [x]
7.      else x ← right [x]
8.  p[z] ← y
9.  if y = NIL
10.       then root [T] ← z
11.       else if key [z] < key[y]
12.       then left [y] ← z
13.       else right [y] ← z
```

TREE_INSERT begins at the root of the tree and traces a path downward. The pointer x traces the path and the pointer y is maintained as the parent of x. After initialization, the while loop in lines 3-7 causes these two pointers to move down the tree, going left or right depending on the comparison of key [z] with key [x], until x is set to NIL. The NIL occupies the position where we wish to place the input item z. Lines 8-13 set the pointers that cause z to be inserted.

Q. 4. (a) Let n = 7; (P_1, P_2 ..... P_7) = (3, 5, 20, 18, 1, 6, 30) and (d_1, d_2 .....d_7) = (1, 3, 4, 3, 2, 1, 2). What is the solution generated by Job Sequencing algorithm for the given problem ? **10**

Ans. Job Sequencing : Job sequencing is the arrangements of the tasks required to be carried out sequentially. There are two techniques of job sequencing :

(i) Priority Rules : Provides guidelines for the sequence in which job should be done.

(ii) **Johnson's Rules :** A technique that can be used for minimize the throughput (completion) time for a group of jobs that are to be processed on two machines or facilities.

Basic Assumption for Job Sequencing :

(i) Only one job can be processed by a machine

(ii) Once the operation is started, it must be performed till completion.

(iii) An operation can start only if the previously started operation gets completed.

(iv) There is only one machine in each type.

(v) A job is processed as soon as possible as per the ordering requirements.

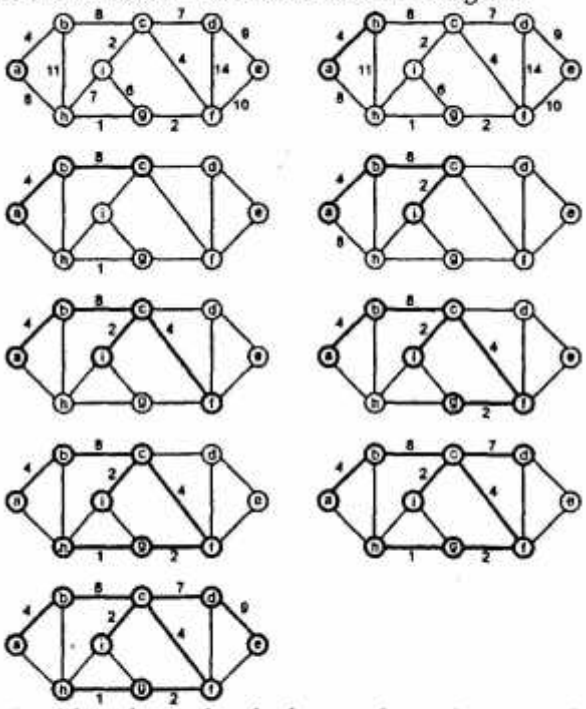
(vi) Only static scheduling is considered; the processing time of all the jobs is known in prior.

(vii) The time taken to transfer jobs in between the machine are negligible.

Q. 4. (b) Write Prim's algorithm to find Minimum Spanning tree of the graph with an example.

Ans. Prim's algorithm is a special case of the generic minimum-spanning tree algorithm. Prim's algorithm operates much like Dijkstra's algorithm for finding shortest paths in a graph. Prim's algorithm has the property that the edges in the set A always from a single tree.

The execution of Prim's algorithm on the graph from figure. The root vertex is a shaded edges are in the tree being grown and the vertices in the tree are shown in diagram.



At each step of the algorithm, the vertices in the tree determine a cut of the graph and a light edge crossing the cut is added to the tree.

The key to implement Prim's algorithm efficiently is to make it easy to select a new edge to be added to the tree formed by the edges in A. Below, the connected graph G and the root r of the minimum spanning tree to be grown are inputs to the algorithm. During execution of the algorithm, all vertices that

are not in the tree reside in a min-priority queue Q based on a key field. For each vertex v, key [v] is the minimum weight of any edge connecting v to a vertex in the tree; by convention, key [v] = ∞ if there is no such edge.

```
MST_PRIM (G, w, r)
1.  for each v ∈ V [G]
2.        do key [v] ← ∞
3.           π [u] ← NIL
4.  key [r] ← 0
5.  Q ← V [G]
6.  While Q ≠ φ
7.        do v ← EXTRACT_MIN (Q)
8.           for each v ∈ Adj [u]
9.              do if v ∈ Q and w (u, v) < key [v]
10.                then π [v] ← u
11.                  key [v] ← w (u, v)
```

The total time for Prim's algorithm is O (V lg V + E lg V) = O (E lg V), which is asymptotically the same as for implementation of Kruskal's algorithm.

Q. 5. Explain Dynamic Programming Method to solve a problem. Use the same method to solve the travelling salesman problem with given distance–matrix : **20**

$$\begin{bmatrix} 0 & 10 & 15 & 20 \\ 5 & 0 & 9 & 10 \\ 6 & 13 & 0 & 12 \\ 8 & 8 & 9 & 0 \end{bmatrix}$$

Ans. Dynamic Programming Method : Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to sub-problems. Divide-and-conquer algorithms partition the problem into independent sub-problems, solve the sub-programs recursively and then combine their solution to solve the original problem. In contrast, dynamic programming is applicable where the sub-problems are not independent, that is, when sub-problems share sub-problems.

A dynamic-programming algorithms solves every sub-subproblem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the sub-subproblem is encountered.

Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value and we wish to find a solution with the optimal (minimum or maximum) value.

The development of dynamic-programming algorithm can be broken into a sequence of four steps :

(i) Characterize the structure of an optimal solution.

(ii) Recursively define the value of an optimal solution.

(iii) Compute the value of an optimal solution in a bottom-up fashion.

(iv) Construct an optimal solution from computed information. There are many key ingredients that an optimaization problem must have in order for dynamic programming to be applicable : Optimal sub-structure and overlapping sub-problems etc.

Optimal Sub-structure : The first step in solving an optimization problem by dynamic programming is to characterize the structure of an optimal solution. In dynamic programming, we build an optimal solution to the problem from optimal solutions to sub-problems.

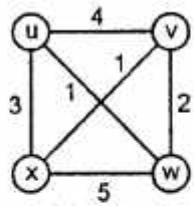
Dynamic programming uses optimal sub-structure in a bottom-up fashion. That is, we first find optimal solutions to sub-problems and having solved the sub-problems, we find an optimal solution to the problem. Finding an optimal solution to the problem entails making a choice among sub-problems as to which we will use in solving the problem. The cost of the problem solution is usually the sub-problem costs plus a cost that is directly attributable to the choice itself.

Overlapping Sub-problems : The second ingredient that an optimization problem must have for dynamic programming to be applicable is that the space of sub-problems must be "small" in the sense that a recursive algorithm for the problem solves the same sub-problems over and over, rather than always generating new sup-problem. When a recursive algorithm revisits the same problem over and over again, we say that the optimization problem has overlapping sub-problems.

Dynamic-programming algorithms typically take advantage of overlapping sub-problems by solving each sub-problem once and then storing the solution in a table where it can be worked up when needed, using constant time per look-up.

The Travelling-Salesman Problem : In the travelling-salesman problem, which is closely related to the Hamiltonian-cycle 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 (u, w, v, x, u), with cost 7. The formal language for the corresponding decision problem is :

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

Q. 6. (a) Explain Branch-and-Bound strategy to solve a particular problem. **5**

Ans. Branch-and-Bound : Branch-and-Bound divide a problem to be solved into a number of sub-problems, similar to the strategy backtracking. The animation uses the travelling-salesman problem as an example. The travelling-salesman problem comprises of finding the cheapest round path on a weighted graph where each node is only visited once.

Branch-and-Bound :

Basic Idea :

(i) Split the problem into sub-problems.

(Similar to the decision tree used in backtracking)

(ii) Constrain the possible choices according to the current best solution.

For example : If the current costs at a given node exceed the current minimum costs (for minimization) the rest of the sequence need not be calculated.

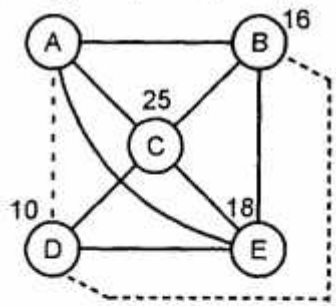
Travelling-Salesman Problem :

Bounsd = 28

Current costs = 28

Cheapest current round trip : A-D-E-B-C-A

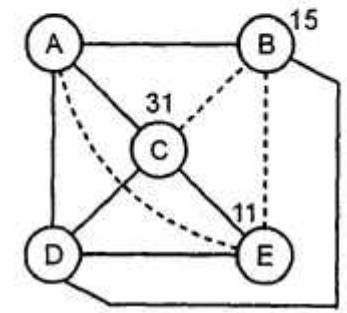
	A	B	C	D	E
A	–	20	30	10	11
B	15	–	16	4	2
C	3	5	–	2	4
D	19	6	18	–	3
E	16	4	7	16	–



Bound = 28

Current cost = 31 more expensive than cheapest trip

Cheapest current round trip : A-D-E-B-C-A



Q. 6. (b) What is 0/1 Knapsack problem ? Solve this problem using Branch-and-Bound method taking suitable example. **5**

Ans. 0/1 Knapsack Problem : A thief robbing a store finds n items; the i^{th} item is worth V_i dollars and weighs w_i pounds where v_i and w_i are integers. He wants to take as valuable a load as possible, but he can carry at most W pounds in his knapsack for some integers W. Which items should he take ? **(This is called the 0/1 kanpsack problem because each item must either be taken or left behind; the thief cannot take a fractional amount of an item or take an item more than once).**

For the 0/1 problem, consider the most valuable load that weighs at most W pounds. If we remove item j from this load, the remaining load must be the most valuable load weighing at most $W - w_j$ that the thief can take from the $n - 1$ original items excluding j.

0/1 problem is not solvable by a greedy strategy.

Branch-and-Bound is a general algorithm for finding optimal solution of various optimization problems, especially in discrete and combinational optimization.

For definiteness, we assume that the goal is to find the minimum value of function f(x). Where x renges over some set S of admissible or candidate solutions.

A branch and bound procedure require two tools. The first one splitting procedure that, given a set S of candidates, returns two or more smaller set S_1, S_2, ... whose union cover S. Note that the minimums of f(x) over S is min $\{v_1, v_2 .....\}$. Where each V_i is the minimum of f(x) within S_i. This step is called branching, since its recursive application define a tree structure where nodes are the subsets of S. Another tool is a procedure that computers upper and lower bounds for the minimum value of f(x) with in a given sub-set S. This step is called bounding.

Q. 8. (a) Write short notes on the following : **14**

(i) Cook's Theorem

(ii) NP-Hard and NP-Complete Problems.

(iii) Prove that travelling salesman problem is NP-Hard. **6**

Ans. (i) Cook's Theorem (Satisfiability is NP-Complete) :

Theorem : Satisfiability of Boolean formulas is NP-complete.

Proof : We start by arguing that SAT $\in$ NP. Then we prove that CIRCUIT_SAT is NP-hard by showing that CIRCUIT_SAT $\leq_p$SAT;

To show that SAT belongs to NP, we show that a certificate consisting of a satisfying assignment for an input formula ϕ can be verified in polynomial time. The verifying algorithm simply replaces each variable in the formula with its corresponding value and then evaluates the expression.

$$\text{SAT} \in \text{P} \leftrightarrow \text{P} = \text{NP OR}$$

If $$\text{SAT} \in \text{P} \Rightarrow \text{P} = \text{NP OR}$$

For every problem

$$\pi \in \text{NP } \pi \; \alpha \text{ SAT OR}$$

$$\text{SAT} \in \text{NPC}$$

It can be shown that

(i) Graph colouring $\in$ NP

(ii) SAT α colouring

$\Rightarrow$ Graph colouring $\in$NPC

Similarly, K – clique, HC, TSP etc. are also NPC.

(ii) NP-Hard and NP-Complete : 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 $L_1 \leq_p L_2$, then L_1 is not more than a polynomial factor harder than 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

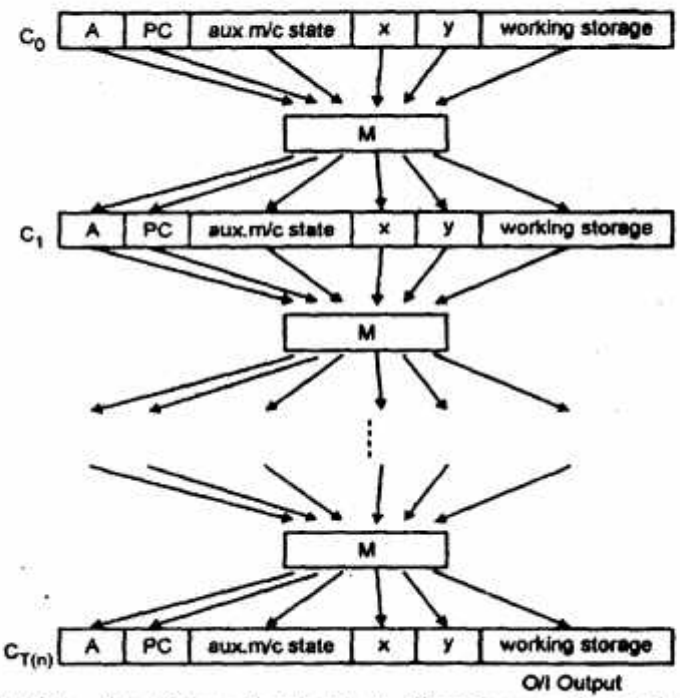
(ii) $L' \leq_p L$ for every $L' \in NP$

If a language L satisfies property 2, but not necessarily property 1, we say that **L is NP-hard.**

The Circuit-satisfiability Problem is NP-hard : Let L be any language in NP. Polynomial-time algorithm F computing a reduction function 'f' that maps every binary string x to a circuit $C = f(x)$, such that $x \in L$ if only if $C \in$ CIRCUIT-SAT.

Since $L \in NP$, there most exist an algorithm A that verifies L in polynomial time. The algorithm F that we shall construct will use the two input algorithm A to compute the reduction function.

The basic idea of the proof is to represent the computation of A as a sequence of configurations. As shown in figure, each configuration can be broken into parts consisting of the program for A, the program counter and auxiliary machine state, the input x, the certificate y and working storage. Starting with an initial configuration C_0, each configuration C_i is mapped to a subsequent configuration C_{i+1} by the combinational circuit M implementing the computer hardware. The output of the algorithm A – 0 to 1 – is written to some designed location in the working storage when A finishes executing and if we assume that thereafter A halts, the value never changes. Thus, if the algorithm runs for at most T(n) steps, the output appear as one of the bits in $C_T(n)$.



NP-Complete Problem (The Clique Problem) : A clique is an undirected graph G – (V, E) is a subset $V' \subseteq V$ of vertices, each pair of which is connected by an edge in E. In other words, a clique is a

complete sub-graph of G. The size of a clique is the number of vertices it contains. The clique problem is the optimization problem of finding a clique of maximum size in a graph.

The formal definition is

CLIQUE = { < G, K > : G is a graph with a clique of size K}.

(iii) To prove that TSP is NP-hard, we show that HAM-CYCLE $\leq_p$ TSP. Let G = (V, E) be an instance of HAM-CYCLE. We construct an instance of TSP as follows. We form the complete graph $G' = (V, E)$,

Where, $E' = \{(i, j) : i, j \in V \text{ and } i \neq j\}$ and we define the cost function C by

$$C(i, j) = \begin{cases} 0 & \text{if } (i, j) \in E \\ 1 & \text{if } (i, j) \in E \end{cases}$$

The instance of TSP is then $(G', C, 0)$, which is easily formed in polynomial time.

We now show that graph G has a Hamiltonian cycle if and only if graph G' has a tour of cost at most 0. Suppose that graph G has a Hamiltonian cycle h. Each edge in h belongs to E and thus has cost 0 in G'. Thus, h is a tour in G' with cost 0. Conversely, suppose that graph G' has a tour h' of cost at most 0. Since, the costs of the edges in E' are 0 and 1, the cost of tour h' is exactly 0 and each edge on the tour must have cost 0. Therefore, h' contains only edges in E, we conclude that h' is a Hamiltonian cycle in graph G.