Course Code : MCS-031
Course Title : Design and Analysis of Algorithms
Assignment Number : MCA (3)/031/Assign/2014-15

Question: 1

a) Discuss briefly the five essential attributes of an algorithm.

Solution:

Five important essential attributes of an algorithm:

1. Finiteness: An algorithm must terminate after a **finite number** of steps and further each step must be executable in **finite amount of time**. In order to establish a sequence of steps as an algorithm, it should be established that it**terminates** (in finite number of steps) on *all allowed* inputs.

2. **Definiteness* (*no ambiguity*):** Each step of an algorithm must be precisely defined; the action to be carried out must be rigorously and unambiguously specified for each case. Through the next example, we show how an instruction may *not* be definite.

3. Inputs: An algorithm **has zero or more, but only finite,** number of inputs. Examples of algorithms requiring *zero* inputs:

(i) Print the largest integer, say MAX, representable in the computer system being used.

(ii) Print the ASCII code of each of the letter in the alphabet of the computer system being used.

(iii) Find the sum *S* of the form *1+2+3+…,* where *S* is the largest integer less than or equal to MAX defined in Example (i) above.

4. Output: An algorithm has **one or more outputs**. The requirement of at least one output is obviously essential, because, otherwise we cannot know the answer/solution provided by the algorithm. The outputs have specific relation to the inputs, where the relation is defined by the algorithm.

5. Effectiveness: An algorithm should be effective. This means that **each of the operation** to be performed in an algorithm **must be sufficiently basic** that it can, in principle, **be done exactly** and in a **finite length of time,** by a person using pencil and paper. *It may be noted that the 'FINITENESS' condition is a special case of 'EFFECTIVENESS'. If a sequence of steps is not finite, then it cannot be effective also.*

b) Write a recursive procedure for the product of first n natural numbers.
Then explain how your algorithm computes product of first 6 natural numbers.

Solution:

```
function product(n)
{
if(n==1)
return n;
else
return (n * product(n-1));
}
```

For computing PRODUCT (6) by the algorithm

As n=6 ≠ 1

Therefore

P6 ≠ n * PRODUCT (n-1) = 6* *PRODUCT* (5)

{It may be noted that in different calls of the procedure PRODUCT, the variable names occurring within the procedure, denote different variables. This is why instead of p we use the names Pi for i=6,5,4,3,……}

Therefore, in order to compute *PRODUCT* (6), we need to compute *PRODUCT*(5)

n=5 ≠ 1, therefore

P5 5 * *PRODUCT* (4)

Continuing like this, we get

P4 4 * *PRODUCT* (3)

P3 3 * *PRODUCT* (2)

P2 2 * *PRODUCT* (1)

P1 1 * *PRODUCT* (1)

At this stage n=0, and accordingly, the algorithm returns value 0. Substituting the value o of *PRODUCT* (1) we get

P1= 1*1=1 which is returned by *PRODUCT* (1).

Substituting this value we get P2=2. Continuing like this, we get P3 =6, P4=12, P5=20 and P6=30

c) Arrange the following growth rates in increasing order:

O (n (log n)2), O ((35)n) , O (35n2 +11), O (1), O (n log n)

d) In respect of understanding a problem for solving it using a computer, explain "analyzing the problem" step.

Solution:

This step is useful in determining the characteristics of the problem under consideration, which may help in solving the problem. Some of the characteristics in this regards are discussed below:

(i) Whether the problem is decomposable into independent smaller or easier subproblems, so that programming facilities like procedure and recursion etc. may be used for designing a solution of the problem. For example, the problem of evaluating can

be do decomposed into smaller and simpler problems viz.,

(ii)Whether steps in a proposed solution or solution strategy of the problem, may or may not be ignorable, recoverable or inherently irrecoverable, i.e., irrecoverable by the (very) nature of the problem. Depending upon the nature of the problem, the solution strategy has to be decided or modified.

Depending on the nature of the problem as ignorable-step, recoverable-step or irrecoverable-step problem, we have to choose our tools, techniques and strategies for solving the problem.

Question: 2

Suppose that instead of binary or decimal representation of integers, we have a representation using 5 digits, viz. 0, 1,2,3,4, along with 5's complement, representation of integers. For example, the integer 147 is represented as $(01042)_5$ = (in decimal) + 1. 5^3 + 0. 5^2 + 4. 5^1 + 2. 3^0, where the leading zero indicates positive sign.

And the integer (− 147) in 5's complement is represented by 13403, the leading 1 indicates negative sign. The other digits, except the right-most, in the representation of (− 147) are obtained by subtracting from 4 the corresponding digit in 147"s representation, and then adding 1 (the representation of –147 is obtained as 13402 + 00001).

Write a program for the arithmetic (negation of an integer, addition, subtraction and multiplication of two integers) of integers using 5's complement representation. The program should include a procedure for calculating each of negation of an integer, addition, subtraction and multiplication of two integers. The integers will use 8 5-digit positions, in which the left-most position will be used for sign.

Using your program finds the 5-digit representation of each of the decimal numbers/ expression: 345, − 297, 18 and ((345 − 297) * 18)

Question: 3

a) Write a short note on each of the following:

i) Best case analysis

Solution:

Best- Case Analysis of an algorithm for a given problem involves finding the shortest of all the times that can (theoretically) be taken by the various instances of a given size, say n, of the problem. In other words, the best-case analysis is concerned with

(i) Finding the instances (types) for which algorithm runs fastest and then

(ii) With finding running time of the algorithm for such instances (types).

If C-best (n) denotes the best-case complexity for instances of size n, then by definition, it is guaranteed that no instance of size n, of the problem, shall take less time than C-best (n).

In general, it can be shown that

(i) For an already sorted list of w elements, sorted in the required order, the Insertion-

Sort algorithm will make (n- 1) comparisons and 4 x (n – 1) assignments and
(ii) (n – 1) comparisons and 4 (n – 1) assignments are the minimum that are required of sorting any list of n elements, that is already sorted in the required order.
Thus the best time complexity of Insertion Sort algorithm is a linear polynomial in the size of the problem instance.

ii) Amortized analysis

Solution:

We observed that

(i) *Worst-case* and *best-case* analyses of an algorithm may not give a good idea about the behavior of the algorithm on a *typical or random* input.
(ii) Validity of the conclusions derived from *average-case* analysis depends on the quality of the assumptions about probability distribution of the inputs of a given size.
Another important fact that needs our attention is the fact that most of the operations, including the most time-consuming operations, on a data structure (used for solving a problem) do not occur in isolation, but different operations, with different time complexities, occur as a part of a *sequence* of operations. *Occurrences of a particular operation in a sequence of operations are dependent on the occurrences of other operations in the sequence*. As a consequence, it may happen that the most time consuming operation can occur but only rarely or the operation only rarely consumes its theoretically determined maximum time. But, we continue with our argument in support of the need for another type of analysis, viz., ***amortized analysis*****,** for better evaluation of the behavior of an algorithm. However, this fact of *dependence*of both the occurrences and complexity of an *operation* on the occurrences of other operations is not taken into consideration in the earlier mentioned analyses.

b) Using one-by-one (i) insertion sort (ii)heap sort and (iii) merge sort, sort the following sequence in increasing order and analyze (i.e., find number of comparisons and assignments in each of) the algorithm: 84, 35, 47, 18, 82, 17, 56, 40, 12 ,67

Question:4

***a)* The following pseudo-code is given to compute (ab)mod n, where a, b and n are positive integers. Trace the algorithm to compute 11^{362} mod 561**

MODULAR-EXPONENTION (a, b, n)

Let (bk, bk − 1, … , b0) be binary representation of b, in which bk=1

1. d ⇓ a; d stores partial results of (ab) mod n

2. for i ⇓ (k –1) downto 0

3. do

4. d ⇓ (d . d) mod n

5. if bi = 1

6. d ⇓ (d . a) mod n

7. end-do

8. return d.

Note: **The above algorithm is a sort of implementation of the ideas explained in Section 1.9 of Block2 of MCS- 031, and should be learned along with Section 1.9.**

Solution:

b) Explain the essential idea of Dynamic Programming. How does Dynamic Programming differ from Divide and conquer approach for solving problems?

Solution:

There are two key attributes that a problem must have in order for dynamic programming to be applicable: optimal substructure and overlapping sub problems. If a problem can be solved by combining optimal solutions to *non-overlapping* sub problems, the strategy is called "divide and conquer" instead.

Optimal substructure means that the solution to a given optimization problem can be obtained by the combination of optimal solutions to its sub problems. Consequently, the first step towards devising a dynamic programming solution is to check whether the problem exhibits such optimal substructure. Such optimal substructures are usually described by means of recursion. Hence, one can easily formulate the solution for finding shortest paths in a recursive manner, which is what the Bellman–Ford algorithm or the Floyd–Warshall algorithm does.

Overlapping sub problems means that the space of sub problems must be small, that is, any recursive algorithm solving the problem should solve the same sub problems over and over, rather than generating new sub problems. Even though the total number of sub problems is actually small (only 43 of them), we end up solving the same problems over and over if we adopt a naive recursive solution such as this. Dynamic programming takes account of this fact and solves each sub problem only once.

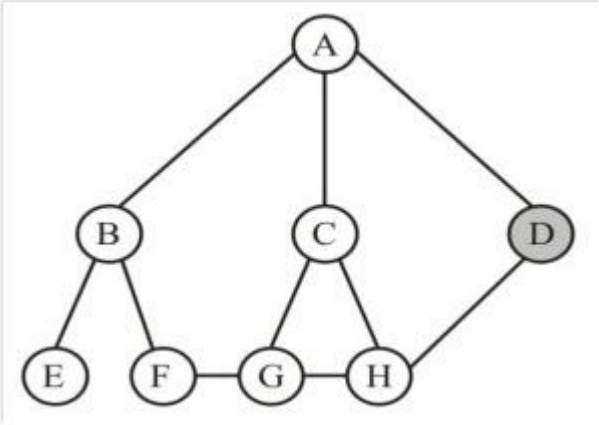
This can be achieved in either of two ways:

- **Top-down approach**: This is the direct fall-out of the recursive formulation of any problem. If the solution to any problem can be formulated recursively using the solution to its sub problems, and if its sub problems are overlapping, then one can easily memorize or store the solutions to the sub problems in a table. Whenever we attempt to solve a new sub problem, we first check the table to see if it is already solved. If a solution has been recorded, we can use it directly; otherwise we solve the sub problem and add its solution to the table.
- **Bottom-up approach:** Once we formulate the solution to a problem recursively as in terms of its sub problems, we can try reformulating the problem in a bottom-up fashion: try

solving the sub problems first and use their solutions to build-on and arrive at solutions to bigger sub problems. This is also usually done in a tabular form by iteratively generating solutions to bigger and bigger sub problems by using the solutions to small sub problems. For example, if we already know the values of F_{41} and F_{40}, we can directly calculate the value of F_{42}.

Question 5:

a) For the graph given in Figure below, use (i) BFS (ii) DFS to visit various vertices. The vertex B is taken as the starting vertex and, if there are more than one vertices adjacent to a vertex, then the adjacent vertices are visited in lexicographic order.



b) In context of graph search, explain the minimax principle.

Solution:

Whichever search technique we may use, it can be seen that many graph problems including game problems, complete searching of the associated graph is not possible. The alternative is to perform a partial search or what we call a limited horizon search from the current position. This is the principle behind the minimax procedure.

Minimax is a method in decision theory for minimizing the expected maximum loss. It is applied in two players games such as tic-tac-toe, or chess where the two players take

alternate moves. All these games have a common property that they are logic games. This means that these games can be described by a set of rules and premises. So it is possible to know at a given point of time, what are the next available moves. We are using an assumption that MAX moves first and after that the two players will move alternatively.

Question 6:

a) Is there a greedy algorithm for every interesting optimization problems? Justify your Claim.

Solution:

A ***greedy algorithm*** always makes the choice that looks best at the moment. That is, it makes a locally optimal choice in the hope that this choice will lead to a globally optimal solution. A greedy algorithm for the activity-selection problem is given in the following pseudocode. We assume that the input activities are in order by increasing finishing time:

â$_1$ â$_2$ . . . â$_n$.

If not, we can sort them into this order in time *O*(*n* 1g *n*), breaking ties arbitrarily. The pseudocode assumes that inputs *s* and â are represented as arrays.

GREEDY-ACTIVITY-SELECTOR(*s, f*)

1 *n* *length*[*s*]

2 *A* {1}

3 *j* 1

4 for *i* 2 to *n*

5 do if *s*$_i$ â$_j$

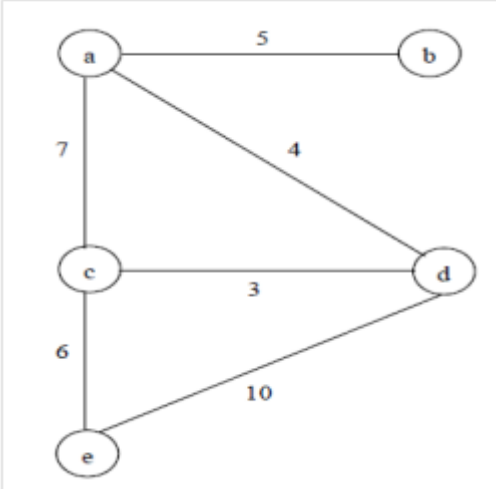
6 then *A* *A* {*i*}

7 *j* *i*

8 return *A*

The activity picked next by GREEDY-ACTIVITY-SELECTOR is always the one with the earliest finish time that can be legally seheduled. The activity picked is thus a “greedy” choice in the sense that, intuitively, it leaves as much opportunity as possible for the remaining activities to be scheduled. That is, the greedy choice is the one that maximizes the amount of unscheduled time remaining.

b) Apply each of (i) Prim's and (ii) Kruskal's algorithms one at a time to find minimal spanning tree for the following graph



The student should include the explanation on the lines of However, the steps and stages in the process of solving the problem are as follows.

Q7. Write note on each of the following:

i) Unsolvability/ undecidability of a problem

Solution:

A function is said to be uncomputable if no such machine exists. There may be a Turing machine that can compute f on part of its domain, but we call the function computable only if there is a Turing machine that computes the function on the whole of its domain. For some problems, we are interested in simpler solution in terms of "yes" or "no". For example, we consider problem of context free grammar i.e., for a context free grammar G, Is the language L(G) ambiguous. For some G the answer will be "yes", for others it will be "no", but clearly we must have one or the other. The problem is to decide whether the statement is true for any G we are given. The domain for this problem is the set of all context free grammars. We say that a problem is decidable if there exists a Turing

machine that gives the correct answer for every statement in the domain of the problem. A class of problems with two outputs “yes” or “no” is said to be decidable (solvable) if there exists some definite algorithm which always terminates (halts) with one of two outputs “yes” or “no”. Otherwise, the class of problems is said to be undecidable (unsolvable).

ii) Halting problem

Solution:

We start with a problem which is important and that at the same time gives us a platform for developing later results. One such problem is the halting problem. Algorithms may contain loops that may be infinite or finite in length. The amount of work done in an algorithm usually depends on the data input. Algorithms may consist of various numbers of loops, nested or in sequence. Informally, the **Halting problem** can be put as:

Given a Turing machine M and an input *w* to the machine M, determine if the machine M will eventually halt when it is given input *w*.

Trial solution: Just run the machine M with the given input *w*.

- If the machine M halts, we know the machine halts.
- But if the machine doesn’t halt in a reasonable amount of time, we cannot conclude that it won’t halt. May be we didn’t wait long enough.

iii) Reduction of a problem for determining decidability

Solution:

Once we have shown that the halting problem is undecidable, we can show that large classes of other problems about the input / output behaviour of programs are undecidable.

Examples of undecidable problems

- **About Turing machines**:
- Is the language accepted by a TM empty, finite, regular, or context-free?
- Does a TM meet its “specification ?” that is, does it have any “bugs.”
    - **About Context Free languages**
    - Are two context-free grammars equivalent?
    - Is a context-free grammar ambiguous?

Reducing problem B to problem A means finding a way to convert problem B to problem A, so that a solution to problem A can be used to solve problem B. One may ask, Why is this important? A reduction of problem B to problem A shows that problem A is at least as difficult to solve as problem B.Also, we can show the following:

- To show that a problem A is undecidable, we reduce another problem that is known to be undecidable to A.

- Having proved that the halting problem is undecidable, we use problem reduction to show that other problems are undecidable.

iv) Rice theorem

Solution:

Rice's theorem (proof is not required)

- "Any functional property of programs is undecidable."
- A functional property is:

(i) a property of the input/output behaviour of the program, that is, it describes the mathematical function the program computes.

(ii) nontrivial, in the sense that it is a property of some programs but not all programs.

Examples of functional properties

- The language accepted by a TM contains at least two strings.
- The language accepted by a TM is empty (contains no strings).
- The language accepted by a TM contains two different strings of the same length.

Rice's theorem can be used to show that whether the language accepted by a Turing machine is context-free, regular, or even finite, are undecidable problems. Not all properties of programs are functional.

v) Post correspondence problem

Solution:

Undecidable problems arise in language theory also. It is required to develop techniques for proving particular problems undecidable. In 1946, Emil Post proved that the following problem is undecidable:

Let ∑ be an alphabet, and let L and M be two lists of nonempty strings over ∑, such that L and M have the same number of strings. We can represent L and M as follows:

L = (w1, w2, w3, …, wk)

M = (v1, v2, v3, …, vk)

Does there exist a sequence of one or more integers, which we represent as (i, j, k, …, m), that meet the following requirements:

- Each of the integers is greater than or equal to one.
- Each of the integers is less than or equal to k. (Recall that each list has k strings).
- The concatenation of wi, wj, wk, …, wm is equal to the concatenation of vi, vj, vk, …, vm.

If there exists the sequence (i, j, k, …, m) satisfying above conditions then

(i, j, k, …, m) is a solution of PCP.

v) NP-complete problem

Solution:

A Problem P or equivalently its language L is said to be NP-complete if the following two conditions are satisfied:

(i) The problem L2 is in the class NP

(ii) For any problem L2 in NP, there is a polynomial-time reduction of L1 to L2.

In this context, we introduce below another closely related and useful concept**.**

A problem from the class NP can equivalently but in more intuitive way, be defined as one for which a potential solution, if given, can be **verified** in polynomial time whether the potential solution is actually a solution or not.

The problems in this class, are called **NP-Complete problems** (to be formally defined later). *More explicitly, a problem is NP-complete if it is in NP and for which no polynomial-time Deterministic TM solution is known so far.* Most interesting aspect of NP-complete problems, is that for each of these problems*neither, so far*, it has been possible to design a Deterministic polynomial-time TM solving the problem *nor* it has been possible to show that Deterministic polynomial – time TM solution *cannot* exist.

vii) K-colourability problem

Solution:

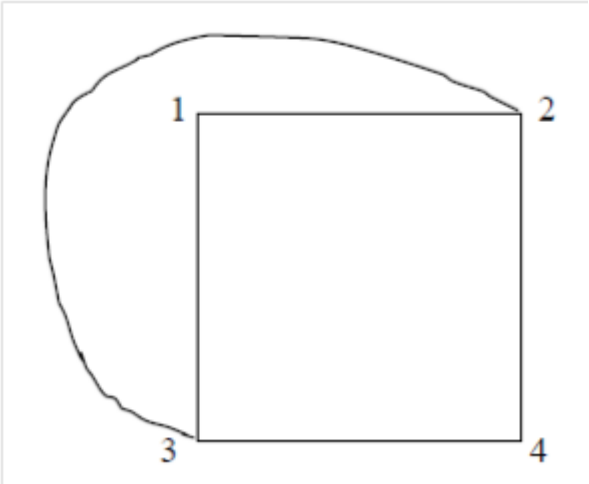
The restricted k-list coloring problem is the k-list coloring problem in which the lists l(v) of colors are subsets of {1, 2,… ,k}.

Our general approach is to take an instance of a specific coloring problem Φ for a given graph and replace it with a polynomial number of instances φ1,φ2,φ3,… such that the answer to Φ is “yes” if and only if there is some instance φk that also answers “yes”.

In general, if we recursively apply such an approach we would end up with an exponential number of equivalent coloring instances to Φ. A k-colouring of G is an assignment to each vertex of one of the k colours so that no two adjacent vertices have the same color. It may be recalled that two vertices in a graph are adjacent if there is an edge between the two vertices.

As the vertices 1, 2, 3 are mutually adjacent therefore, we require at least three colours

for k-colouring problem.



viii) Independent set problem

Solution:

All you have to do is to model the problem as a maximum independent set in a graph. The graph has one vertex for each game; two vertices are connected by an edge whenever at least one student has signed up for both games corresponding to the two vertices. Scheduling the maximum number of games for the Saturday open house corresponds to _nding the maximum independent set in this graph. A co-op student has already built the graph based on the student sign up sheets; you can concentrate on the independent-set algorithm. After that, you say to your boss: \Oh Yes, of course." Then, you go back to your \Introduction to Algorithms" book by Cormen, Leiserson and Rivest to study the de_nitions for independent sets…

An independent set in a graph G = (V;E) is a subset I _ V such that for all x; y 2 I, fx; yg 62 E.

The independent-set problem consists of _nding a maximum (largest) independent set in a graph. This optimization problem can be transformed into the decision problem

corresponding to the following language:
IndSet = f< G; k >: G is a graph with an independent set of size kg
Then you recall:
\Oh. I have shown in the CSI 4105 midterm test that IndSet is NP-complete. I even recall that I used a reduction from the Clique problem."
A subset V1 of the set of vertices V of graph G is said to be independent, if *no two* distinct vertices in V1 are adjacent. For example, in the above graph V1 = {1, 4} is an independent set.