07/08/23

DATE : / /
PAGE NO. : 9

Analysis of Algorithms and Complixity Theory
Assignment 2

Q1 Differentiate between : ① P and NP Class
② NP Complete and NP Hard.

A (1) P and NP Class

P class	NP class.
(i) An algorithm for ~~which~~ given input, a definite output is generated is called Polynomial time algorithm	(i) An algorithm is called non-deterministically polynomial time algorithm ~~which~~ for given input, there are more than one paths that the algorithm can follow.
(ii) P problems can be solved and verified in polynomial time.	NP problems cannot be solved in polynomial time, but if the solution is given, then it can be verified in polynomial time.
(iii) P problems are a subset of NP problems	NP problems are a superset of P problems
(iv) All P problems are deterministic	All NP problems are non-deterministic
(v) The solution to P class problems is easy to find.	The solution to NP class problems is hard to find.
(vi) Example: Selection sort, linear search	Example: Travelling salesman, Knapsack problem

(ii) NP Complete and NP Hard

	NP Hard	NP Complete
(i)	NP Hard Problems (say X) can be solved if and only if there is an NP-Complete Problem (say Y) that can be reducible to X in polynomial time	(i) NP Complete problems can be solved by non-deterministic algorithm/ Turing Machine in polynomial time
(ii)	It doesn't have to be in NP Complete	(ii) To solve this problem it must be both NP and NP Hard.
(iii)	Time is unknown in NP Hard	Time is known.
(iv)	NP hard is not a decision problem	NP-Complete is exclusively a decision problem
(v)	Not all NP-hard problems are NP complete	All NP complete problems are NP hard.
(vi)	~~Does not have be~~ It is an optimization problem	It is exclusively a decision problem
(vii)	Example: Halting problems, Vertex cover problem, etc	Example: Hamiltonian cycle, boolean satisfaction, circuit-satisfiability problem, etc

Q2 State the average case and worst case complexity of quick sort.

A(I) Quicksort is a sorting algorithm based on the Divide and Conquer algorithm that picks an element as the pivot and partitions the given array around the picked pivot by placing the pivot in its correct sorted position in the sorted array.

(II) Time Complexity Analysis of Quick Sort

(i) $T(K)$: Time complexity of quicksort of K elements.

(ii) $P(K)$: Time complexity for finding the position of pivot among K elements.

(III) Average Case

(i) For the average case, consider the array gets divided into two parts of size k and size $(N-k)$. So:

$$T(N) = T(N-k) + T(k)$$
$$= \frac{1}{N} * \left[\sum_{i=1}^{N-1} T(i) + \sum_{i=1}^{N-1} T(N-i) \right]$$

(ii) As Both $\sum_{i=1}^{N-1} T(i)$ and $\sum_{i=1}^{N-1} T(N-i)$ are equally likely functions,

$$\therefore T(N) = \frac{2}{N} * \left[\sum_{i=1}^{N-1} T(i) \right]$$

$$\therefore N \cdot T(N) = 2 * \left[\sum_{i=1}^{N-1} T(i) \right] \quad \text{—— ①}$$

(iii) Also :

$$(N-1) * T(N-1) = 2 * \left[\sum_{i=1}^{N-2} T(i) \right] \quad \text{—— ②}$$

(iv) ① - ②

$$N * T(N) - (N-1) * T(N-1) = 2 * T(N-1) + N^2 * \text{constant} - (N-1)^2 * \text{constant}.$$

(v) Dividing both sides by $N*(N-1)$:

$$\frac{T(N)}{N+1} = \frac{T(N-1)}{N+2} * \text{constant}$$

(vi) Now, if we keep substituting $N = N-1, N-2 \ldots$ we'll get.

$$\frac{T(N)}{N+1} = T(1)/2 + 2 * \text{constant} * \left[\frac{1}{2} + \frac{1}{3} + \ldots \frac{1}{(N-1)} + \frac{1}{N} + \frac{1}{(N+1)}\right]$$

$$\therefore T(N) = 2 * \text{constant} * \log_2 N * (N+1)$$

(vii) If we ignore the constant, we get:

$$T(N) = \log_2 N * (N+1).$$

$\therefore$ Average Time complexity $= O(N * \log N)$

(iv) Worst Case Complexity.

(i) The worst case will occur when the array gets divided into 2 parts, one part consisting of N-1 elements and the other and so on. So.

$$T(N) = T(N-1) + N * \text{constant}.$$
$$= T(N-2) + (N-1) * \text{constant} + N * \text{constant}$$
$$= T(N-2) + 2 * N * \text{constant} - \text{constant}.$$
$$\vdots$$
$$= T(N-k) + k * N * \text{constant} - (k-1) * \text{constant} - \ldots - 2*\text{cons} - \text{cons}$$
$$= T(N-k) + k * N * \text{constant} - \text{constant} * (k*(k-1))/2$$

(II) If we put $k = N$ in above equation, then

$T(N) = T(0) + N * N * \text{constant} - \text{constant} * (N(N-1)/2)$

$= N^2 - N*(N-1)/2$

$= \frac{N^2}{2} + \frac{N}{2}$

∴ Worst Case complexity is $O(N^2)$

Q3 What are SAT and 3 SAT problems? Prove that 3 SAT problems are NP Complete.

A(I) SAT Problems

(I) Boolean Satisfiability Problem or simply SAT is the problem of determining if a Boolean formula is satisfiable or not.

(II) Satisfiable : If the Boolean variables can be assigned values such that the formula turns out to be TRUE, then we say that the problem is satisfiable

(III) Unsatisfiable : If it is not possible to assign such values, then we can say that the formula is unsatisfiable.

(II) 3SAT Problem

(I) 3SAT problem is the problem of determining the satisfiability of a formula in conjunctive normal form (CNF) where each clause is limited to at the most 3 literals.

(II) It is also known as 3CNFSAT problem. It is a special case of SAT problems. 3SAT is an NP complete problem.

(III) 3SAT and SAT problems are actually not complete synonyms. 3SAT asks whether it is possible to solve the Boolean satisfiability problem provided that there are at the most 3 variables.

(IV) 3SAT is a simpler problem but there is no algorithms to solve that which is both always correct (not heuristic) and always runs in less than exponential

(III) Prove that 3SAT Problem is NP Complete

To prove that a problem is NP Complete (say B)

(i) Select an NP complete language say A

(ii) Construct a function f that maps the members of A to members of B

(iii) Show that x is in A if and only if f(x) is in B

(iv) Now show that the function f can be computed in polynomial time

(v) This if A is NP complete and it can be reduced to B in polynomial time, then B comes out to be NP complete.

Proof

(i) The language 3 SAT is a restriction of SAT. We replace each clause C that represents the SAT problem to a function f by family of D_c of clauses that represent satisfiability.

(ii) For example, say:

$$C = a \vee b \vee c \vee d \vee e$$

One can simulate this by:

$$D_c = (a \vee b \vee x) \wedge (\bar{x} \vee c \vee y) \wedge (\bar{y} \vee d \vee e)$$

where x, y are two variables.

(iii) We verify if: If C is False then D_c is FALSE

If C is True then D_c is TRUE

(iv) If f is satisfiable, then there is assignment where each clause C is TRUE. This can be extended to make D_c TRUE.

(v) Further if f is evaluated to FALSE then some clauses say C must be FALSE and thus corresponding family D_c evaluates to FALSE.

(vi) This conversion process can be done in polynomial time. Thus we have shown that SAT reduces to 3SAT in polynomial time.

(vii) We know that SAT is an NP complete problem therefore 3SAT is also NP complete problem.

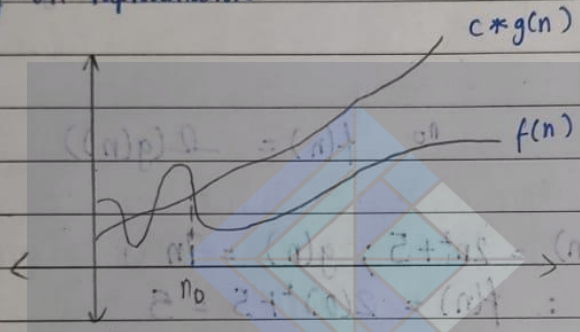
Q4 Explain Big Oh (O), Omega (Ω), Theta (Θ) in detail along with suitable example.

A (I) Big Oh notation

(i) It is denoted by 'O'. It is a method of representing the upper bound of algorithm's running time.

(ii) Using Big Oh notation, we can give longest amount of time taken by the algorithm to complete.

(iii) Big Oh Representation



$0 < f(n) < c * g(n)$
$n \geq n_0$

(iv) Example: $f(n) = 2n+2$; $g(n) = n^2$

For constant c, $f(n) \leq c * g(n)$

For $n = 1$: $f(n) = 2n+2 = 4$
$g(n) = n^2 = 1$
$f(n) > g(n)$

For $n = 2$: $f(n) = 2(2)+2 = 6$
$g(n) = 2^2 = 4$
$f(n) > g(n)$

For $n = 3$: $f(n) = 2(3)+2 = 8$
$g(n) = 3^2 = 9$
$\therefore f(n) < g(n)$.

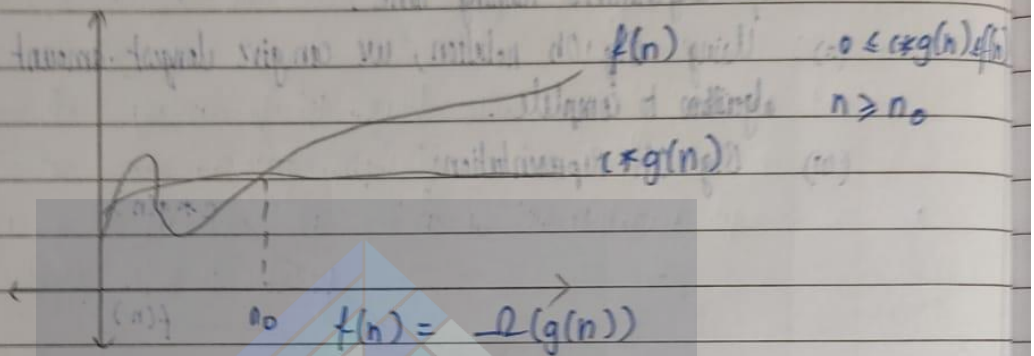
$\therefore$ For $n > 2$, $f(n) < g(n)$

(II) Omega Notation (Ω Notation)

(i) Omega notation is denoted by Ω. It is used to present the lower bound of algorithm's running time.

(ii) Using omega notation, we can denote shortest amount of time taken.

(iii) Big Omega Representations :



(iv) Example: $f(n) = 2n^2+5$; $g(n) = 7n$

If $n=0$: $f(n) = 2(0)^2+5 = 5$

$g(n) = 7(0)$

$\therefore f(n) > g(n)$

If $n=1$: $f(n) = 2(1)^2+5 = 7$

$g(n) = 7(1) = 7$

$\therefore f(n) = g(n)$.

If $n=3$ $\therefore f(n) = 2(3^2)+5 = 18$

$g(n) = 7(3) = 21$

$\therefore f(n) > g(n)$

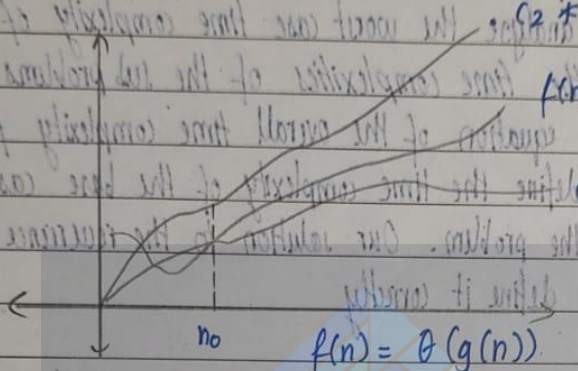
$\therefore$ For $n>3$, we get $f(n) > c*g(n)$

(III) θ Notation

(i) The theta notation is denoted by θ. By this method, the running time is between the upper bound and lower bound.

(ii) It is used for analyzing the average case complexity of an algorithm.

(iii) Representation of θ



$0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$

$n \geq n_0$

$f(n) = \theta(g(n))$

(iv) Example: $f(n) = 2n+8$ $g(n) = 5n$

For $n \geq 2$ $f(n) = 2n+8$

$g(n) = 7n$

i.e. $5n < 2n+8 < 7n$ for $n \geq 2$.

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$.

The θ notation is more precise with both big oh and omega notation.

Q5 Discuss a general plan for analyzing Time Efficiency of Recursive Algorithms.

A Steps to analyze time complexity of recursive algorithms.

(I) Identify input size and smaller subproblems.

(i) We first identify the input size of the larger problem.

(ii) Then we recognize the total number of smaller sub-problems.

(iii) Finally, we identify the input size of smaller sub problems.

(II) Write recurrence relation for the time complexity

(i) We define the time complexity function for the larger function in terms of the input size. For example, if the input size is n, then the time complexity would be T(n).

(ii) Then we define the time complexities of smaller problems.

(iii) Next, we analyse the worst case time complexity of additional operations.

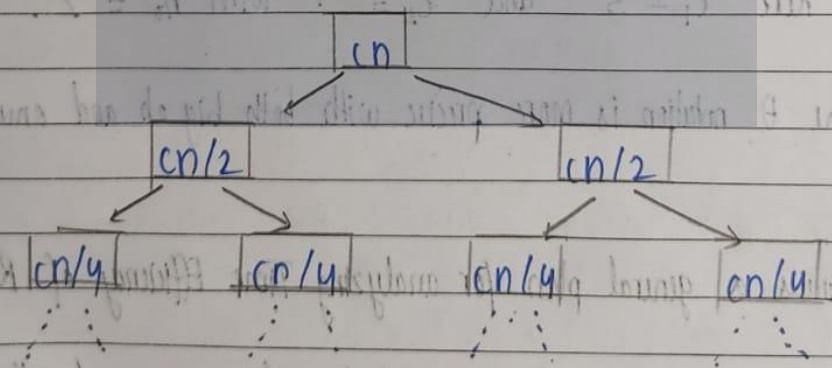
(iv) We add the time complexities of the sub problems and additional operations to get the equation of the overall time complexity function T(n)

(v) We also define the time complexity of the base case i.e. the smallest version of the problem. Our solution to the recurrence depends on this, so we need to define it correctly

(III) Solving recurrence relation to get time complexity

(i) We mainly use the following two methods to solve recurrence relations in algorithms and data structure.

(ii) ① Method 1: Recursion Tree Method
② Method 2: Master Theorem



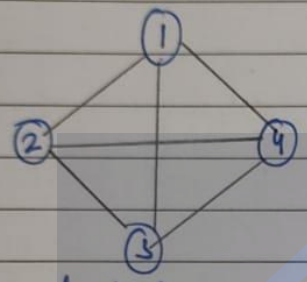
Recursion Tree Diagram

Q6 Explain Non-Deterministic clique problem with algorithm.

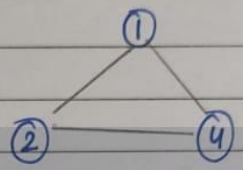
A (I) Clique is nothing but a maximal complete subgraph of a graph G.

(II) The graph G is a set of (V, E) where V is a set of vertices and E is a set of edges. The size of the clique is the number of vertices present in it.

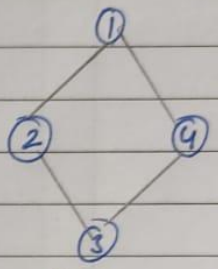
(III)



Graph G



Clique size = 3



Clique size = 4

(IV) Max Clique Problem

(i) It is an optimization problem in which the largest size clique is to be obtained.

(ii) Let DClique (G, k) be a deterministic decision algorithm for determining whether graph G has clique or not of at least size k.

(iii) Then we apply DClique for k = n, n-1, n-2 ... until the output is 1.

(iv) If the clique decision problem is solved in polynomial time then max clique problem can be solved in polynomial time.

(V)

```
Algorithm NonDClique (G, n, k)
{   S := 0
    for i := 1 to k do
    {   t := choice (1, n);
        if t is from set S
            Failure ();
        Add t to set S
    }
```

DATE : / /
PAGE NO. :

// Now S contains k distinct vertices
for all pairs (i,j) i and j are from set S and i != j do
    if (i,j) is not an edge of graph G then
        Failure();
Success();
}