

DAA Unit 2 – PYQ Answers

► October 2022

Q3) a) What is Best, Average and Worst case Analysis of Algorithms? Analyse the following algorithm Best, Average and Worst case [8]

```
void sort(int a[], int n) {
    int i, j, key;
    for (i = 0; i < n; i++) {
        j = i - 1;
        key = a[i];
        while (j >= 0 && a[j] > key) {
            a[j + 1] = a[j];
            j = j - 1;
        }
        a[j + 1] = key;
    }
}
```

Answer:

1. Best, Average, and Worst Case Analysis – Meaning

- **Best Case:** The input is in the most favorable order, so the algorithm performs the minimum number of operations.
- **Average Case:** The expected performance over all possible inputs of size n .
- **Worst Case:** The input is in the least favorable order, requiring the maximum number of operations.

2. Given Algorithm Identification

The given algorithm is **Insertion Sort**.

3. Analysis of the Algorithm

Outer loop:

for ($i = 0$; $i < n$; $i++$) runs exactly **n times** $\rightarrow O(n)$.

Inner loop:

while ($j \geq 0$ && $a[j] > \text{key}$) depends on the arrangement of the input elements.

(i) Best Case – Array already sorted in ascending order

- Condition $a[j] > \text{key}$ fails immediately for each i .

SPPU-BE-COMP-CONTENT – KSKA Git

- Inner loop executes **0 shifts** each time.
- Total comparisons: $n - 1$.
- **Time Complexity: $O(n)$.**
- **Reason:** No shifting, only comparisons for checking order.

(ii) Average Case – *Random order array*

- On average, the inner loop runs $i/2$ times for the i -th iteration.
- Total comparisons: $\approx n^2 / 4$.
- **Time Complexity: $O(n^2)$.**

(iii) Worst Case – *Array sorted in descending order*

- Every new element is smaller than all the previous elements.
- Inner loop runs i times for the i -th iteration.
- Total comparisons: $(n(n-1))/2$.
- **Time Complexity: $O(n^2)$.**

4. Summary Table

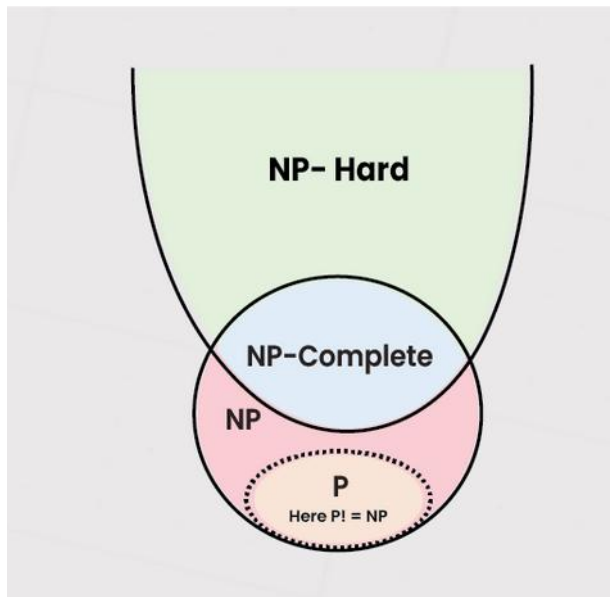
Case	Number of Comparisons	Time Complexity
Best Case	$n - 1$	$O(n)$
Average Case	$\approx n^2 / 4$	$O(n^2)$
Worst Case	$(n(n-1))/2$	$O(n^2)$

Q3) b)

Explain P, NP, NP-Hard and NP-Complete problems with examples. [7 marks]

OR Briefly explain P and NP problems in the context of complexity theory. Give suitable example.

OR What do you understand by NP-complete and NP-hard problems? Give examples



1. P (Polynomial time)

- **Definition:** Class of decision problems that can be solved by a deterministic algorithm in polynomial time.
- It is the collection of decision problems (problems with a "yes" or "no" answer) that can be solved by a deterministic machine (our computers) in polynomial time.
- If the input size is n , the algorithm runs in time $O(n^k)$ for some constant k .
- These are considered “easy” or efficiently solvable problems.
- **Example:** Checking if a number is prime, Sorting an array using Merge Sort ($O(n \log n)$), Shortest Path Problem using Dijkstra’s algorithm $O(n^2)$.

2. NP (Non-deterministic Polynomial time)

- **Definition:** Class of decision problems for which a given solution can be verified in polynomial time.
- It is the collection of decision problems that can be solved by a non-deterministic machine in polynomial time.

SPPU-BE-COMP-CONTENT – KSKA Git

- **Example: Boolean Satisfiability (SAT) Problem:** Given a Boolean formula, check if there exists an assignment of truth values that makes the formula true,

Hamiltonian Cycle Problem: Verify if a given cycle visits every vertex exactly once.

- **Features:**

The solutions of the NP class might be hard to find since they are being solved by a non-deterministic machine but the solutions are easy to verify.

Problems of NP can be verified by a deterministic machine in polynomial time.

Key Difference:

- **P:** Problems **solvable quickly** (in polynomial time).
- **NP:** Problems whose solutions can be **verified quickly** (but solving may take super-polynomial time).

1. NP-complete Problems:

- **Definition:**

A problem is NP-complete if it is both NP and NP-hard. NP-complete problems are the hard problems in NP.

- If any NP-complete problem is solved in polynomial time, **all NP problems** can be solved in polynomial time.

- **Examples:**

- 3-SAT Problem
- Travelling Salesman Problem (decision version)
- Knapsack Problem (decision version)

2. NP-hard Problems:

- **Definition:**

Class of problems at least as hard as the hardest problems in NP. They may or may not be in NP themselves (may not have polynomial-time verifiable solutions).

- Can be **optimization problems**.
- It is a class of problems such that every problem in NP reduces to NP-hard.

- **Examples:**

- Travelling Salesman Problem (optimization version)

SPPU-BE-COMP-CONTENT – KSKA Git

- Halting Problem
- Chess endgame problem

Key Difference: All NP-complete problems are NP-hard, but **not all NP-hard problems are NP-complete.**

Q) Explain 3-SAT problem using an example. Why is SAT so important in theoretical computer science?

• 3-SAT Problem

- **Definition:** Special case of SAT where each clause in the Boolean formula has exactly 3 literals.

. 3-SAT Problem:

- 3-SAT is a special case of SAT where the Boolean formula is in **CNF (Conjunctive Normal Form)**, and each clause has exactly 3 literals.
- Example:

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_2 \vee x_3 \vee \neg x_4)$$

- This is a **3-SAT formula** with three clauses.
- A satisfying assignment could be: $x_1 = \text{True}, x_2 = \text{True}, x_3 = \text{False}, x_4 = \text{True}$.

Why SAT is Important in Theoretical CS

- **First NP-Complete problem** (Cook-Levin Theorem).
- Many NP problems can be reduced to SAT, making it a central problem for complexity theory.
- Efficient SAT solvers are widely used in AI, verification, and cryptography.

Q4) a) What is NP-complete class problem? How would you prove vertex cover problem is NP-complete class problem? [8]

1. NP-Complete Class Problem

- A decision problem is **NP-Complete** if:
 1. It is in NP (solution can be verified in polynomial time).
 2. Every problem in NP can be polynomially reduced to it.

2. Vertex Cover Problem

- **Definition:** Given a graph $G = (V, E)$ and an integer k , does there exist a subset of vertices of size $\leq k$ such that every edge in E is incident on at least one vertex from the subset?
- **Verification:** Given a set of vertices, we can check in polynomial time if it covers all edges.

3. Proof Idea that Vertex Cover is NP-Complete

Theorem : The vertex cover is NP-complete.

Proof :

To prove that vertex cover is NP-complete we will apply reduction technique. That means reduce 3-SAT to vertex cover. As we know 3-SAT to basically a Boolean expression S containing 3 variables in each clause and value of S is true.

To prove this theorem consider S be some Boolean formula in CNF (conjunctive Normal Form) having 3 variables in each clause no for each variable a we will create,



Fig. 2.9.6

If the clause is $a + b + c$ we will create a triangle (as there are 3 - variables)

Using 3-SAT we will build a graph as follows for given expression.

$(a + b + c) (a + b + \bar{c}) (\bar{a} + c + \bar{b})$

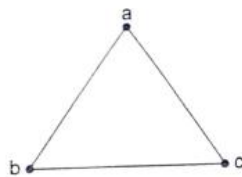


Fig. 2.9.7

The graph has vertex cover of size $K = n + 2m$, where n is number of variables in 3-SAT, m is number of clauses in CNF, K is total number of vertices that are present in the set of vertex cover.

For a clause $(a + b + c)$ we can build a graph as :

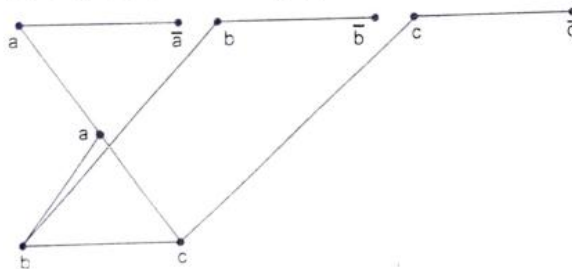


Fig. 2.9.8

SPPU-BE-COMP-CONTENT – KSKA Git

Thus for given expression $(a + b + c)(a + b + \bar{c})(\bar{a} + c + \bar{d})$ we get following graph.

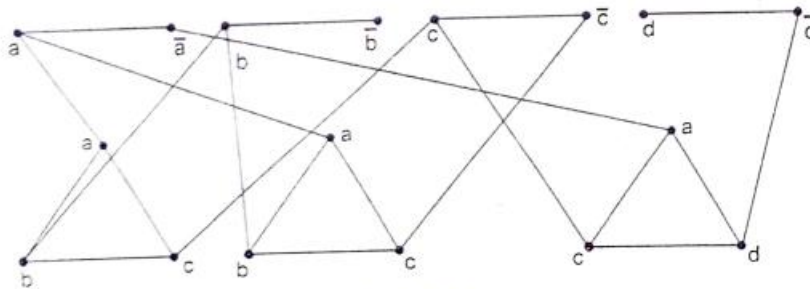


Fig. 2.9.9

Here n = Number of variable = 4

m = Number of clauses = 3

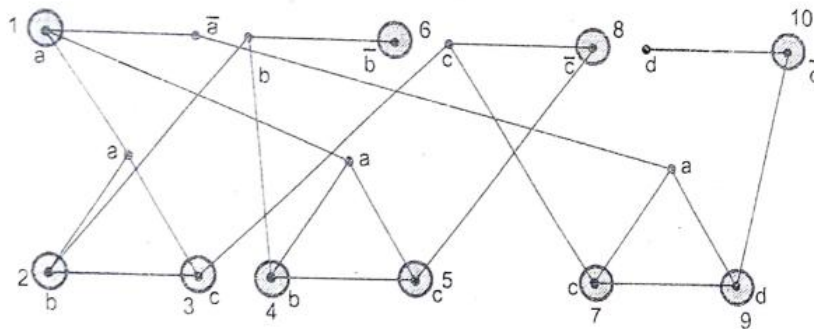
$K = n + 2m = 4 + 2(3) = 10$

Analysis of Algorithms and Complexity Theory

In vertex cover we get $K = 10$ vertices which cover all the vertices. The vertex cover is shown by following graph in which the covering vertices are rounded.

Thus $K = n + 2m$ is proved as $K = 10$. This is how 3-SAT can be reduced to vertex cover. The 3-SAT is NP-complete. Hence vertex is NP-complete is proved.

Thus $K = n + 2m$ is proved as $K = 10$. This is how 3-SAT can be reduced to vertex cover. The 3-SAT is NP-complete. Hence vertex is NP-complete is proved.



Q4) b) Best, Average and Worst Case Analysis of Linear Search [7]

```
int Linear_search(int a[], int n, int item) {
    int i;
    for (i = 0; i < n; i++) {
        if (a[i] == item) {
            return a[i];
        }
    }
}
```

SPPU-BE-COMP-CONTENT – KSKA Git

```
return -1;  
}
```

1. Definitions

- **Best Case:** Minimum number of operations; item found at first position → **$O(1)$** .
- **Average Case:** Item is at a random position; $\sim n/2$ comparisons → **$O(n)$** .
- **Worst Case:** Item is last or not present; n comparisons → **$O(n)$** .

2. Analysis Table

Case	Comparisons	Time Complexity
Best	1	$O(1)$
Average	$n/2$	$O(n)$
Worst	n	$O(n)$

► September 2023

Q3) a) ALREADY DONE

Q3) b) If $f(n) = O(g(n))$ then does it imply $g(n) = O(f(n))$? Discuss [5]

- **Definition of Big-O:** $f(n) = O(g(n))$ means that $f(n)$ grows no faster than $g(n)$ up to a constant factor for large n .
- **Implication:** No, it does not imply $g(n) = O(f(n))$.

Example:

Let $f(n) = n$ and $g(n) = n^2$:

- $f(n) = O(g(n)) \rightarrow n$ grows slower than n^2 .
- But $g(n) \neq O(f(n)) \rightarrow n^2$ does not grow slower than n .

Conclusion: Big-O is not symmetric.

Q3) c) Comment on the statement “Best case analysis of algorithm may not give clear idea of performance.” [2]

- **Reason:** The best case occurs in rare or ideal scenarios (e.g., searching for the first element in linear search).

SPPU-BE-COMP-CONTENT – KSKA Git

- It does not reflect the typical or worst-case performance, so it may be misleading for evaluating algorithm efficiency.

Q4) a) What is SAT and 3-SAT problem? Prove that 3-SAT problem is NP-complete. [8]

SAT (Boolean Satisfiability Problem)

- **Definition:** Given a Boolean formula, determine whether there exists an assignment of truth values to variables that makes the formula true.
- **Example:** $(x_1 \vee \neg x_2) \wedge (x_2 \vee x_3) \rightarrow \text{True}$ if $x_1 = \text{True}, x_3 = \text{True}$.

3-SAT

- **Definition:** Special case of SAT where each clause contains **exactly 3 literals**.
- **Example:**

$$(x_1 \vee \neg x_2 \vee x_3) \wedge (\neg x_1 \vee x_2 \vee x_4) \wedge (x_3 \vee x_4 \vee \neg x_2)$$

Proof that 3-SAT is NP-Complete

Prove that 3-SAT is NP complete

Proof : 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.

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 and y are new variables.

Need to verify :

- 1) If C is FALSE, then D_C is FALSE; and
- 2) If C is TRUE, then one can make D_C TRUE.

If f is satisfiable then there is assignment where each clause C is TRUE. This can be extended to make D_C TRUE.

Further if f is evaluated to FALSE, then some clauses say C' must be FALSE and thus corresponding family $D_{C'}$ evaluates to FALSE.

This conversion process can be done in **polynomial time**. Thus we have shown that SAT reduces to 3-SAT in polynomial time. As we know that SAT is a NP complete problem, so we must say that 3-SAT is also NP complete problem.

SPPU-BE-COMP-CONTENT – KSKA Git

1. **3-SAT** \in **NP**:

- Given a truth assignment, verifying that all clauses are satisfied takes polynomial time.

2. **NP-hardness**:

- SAT is known to be NP-complete (Cook–Levin theorem).
- We can transform any SAT problem into an equivalent 3-SAT instance in polynomial time by rewriting longer clauses into clauses with exactly 3 literals.

3. Since 3-SAT is in NP and is NP-hard, it is **NP-complete**.

Q4) b) What do you understand by best case, worst case and average-case behaviour of an algorithm? Is an average case efficiency an average of best-case, worst-case efficiencies? Justify. [7]

1. **Best Case Analysis**:

- The time complexity when the algorithm takes the **minimum possible time** for an input of size n .
- Represents the **fastest execution scenario**.
- **Example**: In **Linear Search**, if the item is the **first element**, only one comparison is needed \rightarrow **Best case: $O(1)$**

2. **Worst Case Analysis**:

- The time complexity when the algorithm takes the **maximum possible time** for input size n .
- Guarantees an **upper bound** on running time.
- **Example**: In **Linear Search**, if the item is **last element or not present**, it checks all elements \rightarrow **Worst case: $O(n)$** .

3. **Average Case Analysis**:

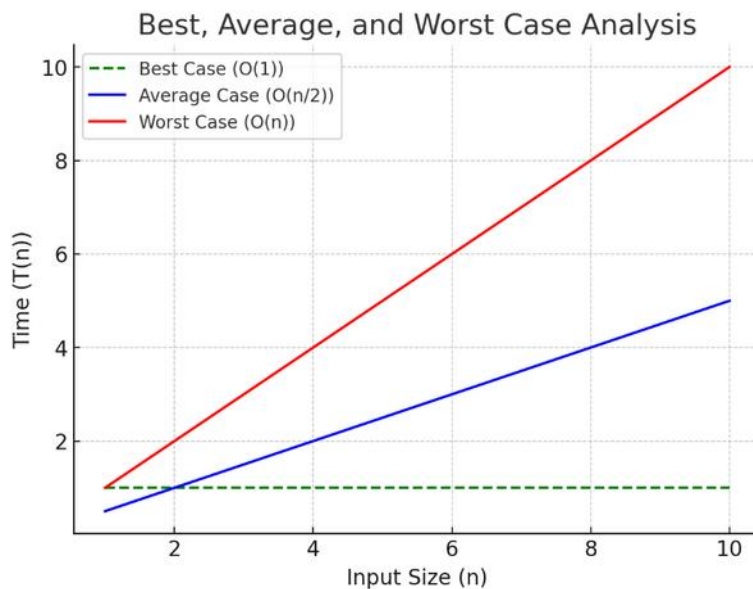
- The expected time complexity of the algorithm **over all possible inputs**, assuming each input is equally likely.
- Provides a more **realistic measure** of performance.

SPPU-BE-COMP-CONTENT – KSKA Git

- **Example:** In Linear Search, if item is equally likely to be anywhere → **Average case:** $O(n/2) = O(n)$.

4. Is Average Case = Average of Best & Worst Cases?

- **No**, average case is **NOT** simply the mean of best and worst cases.
- Average case depends on the **probability distribution of inputs**.
- For example, in Linear Search:
 - Best case = 1 comparison, Worst case = n comparisons.
 - Average case $\neq (1+n)/2$ always, but rather depends on **probability of item position**.



Here's a visual representation of **best-case, average-case, and worst-case behavior** of an algorithm:

- **Best case (green, $O(1)$)** → Minimum time required (e.g., linear search when the item is at the first position).
- **Average case (blue, $O(n/2)$)** → Expected performance over random inputs.
- **Worst case (red, $O(n)$)** → Maximum time required (e.g., item not present in linear search).

The **average case is not just the arithmetic mean of best and worst cases**; instead, it's derived from the **probabilistic distribution of inputs**.

SPPU-BE-COMP-CONTENT – KSKA Git

Final Justification:

- **Best case** is often too optimistic.
- **Worst case** is too pessimistic.
- **Average case** gives the most realistic efficiency but requires **probabilistic analysis**.

► September 2024

Q3) a) Consider the algorithm for selection sort:

```
void select (int A[])
{ // A is an array of n numbers.
  for (i = 1; i < n - 1; i++)
  {
    min_pos = i;
    min_val = A[i];
    for (j = i + 1; j <= n; j++)
    {
      if (A[j] < min_val)
      {
        min_pos = j;
        min_val = A[j];
      }
    }
    A[min_pos] = A[i];
    A[i] = min_val;
  }
}
```

Analyse the complexity of this algorithm. Clearly indicate the assumptions made, if any.

[6 Marks]

Step 1: Assumptions

- The input array A has **n elements** indexed from 1 to n.
- min_pos and min_val assignment and comparison operations take **O(1)** time.
- The swap operation also takes **O(1)** time.

Step 2: Complexity Analysis

- **Outer loop** runs from i = 1 to n - 1 → executes **(n - 1) times**.

SPPU-BE-COMP-CONTENT – KSKA Git

- **Inner loop** runs from $j = i + 1$ to $n \rightarrow$ executes **$(n - i)$ times** for each i .

Total comparisons:

$$T(n) = (n-1) + (n-2) + (n-3) + \dots + 1$$

This is an **arithmetic series**:

$$T(n) = n(n-1)/2$$

Step 3: Time Complexity

- **Best case:** Comparisons still occur $\rightarrow O(n^2)$
- **Worst case:** Comparisons still occur $\rightarrow O(n^2)$
- **Average case:** Also **$O(n^2)$** (comparisons are independent of arrangement).

Step 4: Space Complexity

- Only a few extra variables (min_pos , min_val) $\rightarrow O(1)$ auxiliary space.

Final Answer: The time complexity of selection sort is **$O(n^2)$** in all cases, and space complexity is **$O(1)$** .

b) If an algorithm has a time complexity of both $O(f(n))$ and $\Omega(g(n))$ for the same function $f(n)$ and $g(n)$, then what does it imply? [3 Marks]

If for the same function we have:

$$T(n) = O(f(n)) \text{ and } T(n) = \Omega(g(n))$$

and **$f(n) = g(n)$** , then:

$$T(n) = \Theta(f(n))$$

This means the **algorithm's running time grows asymptotically exactly as $f(n)$** (tight bound).

c) What do Ω and Θ notations mean? When do we use O notation? [6 Marks]

1. Ω (Omega) Notation:

- This notation is used to represent the lower bound of algorithm's running time.
- Using omega notation we can denote shortest amount of time taken by algorithm.
- **Definition :** A function $f(n)$ is said to be in $\Omega(g(n))$ if $f(n)$ is bounded below by some positive constant multiple of $g(n)$:

Example :

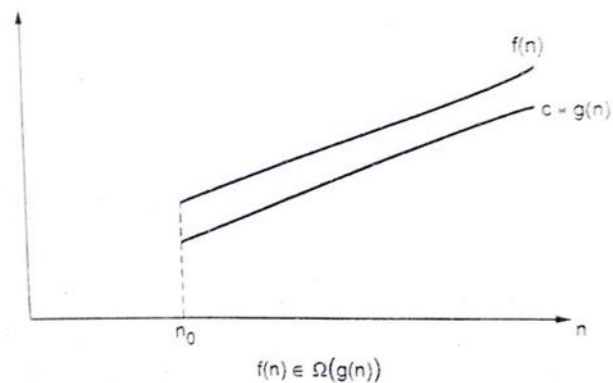


Fig. 2.2.2 Omega representation

- Defines the **asymptotic lower bound** of an algorithm.
- Example: If $T(n) = \Omega(n \log n)$, the algorithm takes **at least** proportional to $n \log n$ time.

2. Θ (Theta) Notation:

- By this method the running time is between upper bound and lower bound.
- Defines the **tight bound** of an algorithm (both lower and upper bounds are same).
- Example: $T(n) = \Theta(n^2)$ means it grows exactly as n^2 .

3. O (Big-O) Notation:

- It is a method of representing the upper bound of algorithm's running time.
- Using big oh notation we can give longest amount of time taken by the algorithm to complete.
- Defines the **asymptotic upper bound** of an algorithm.
- Used to express the **worst-case** performance or growth rate.

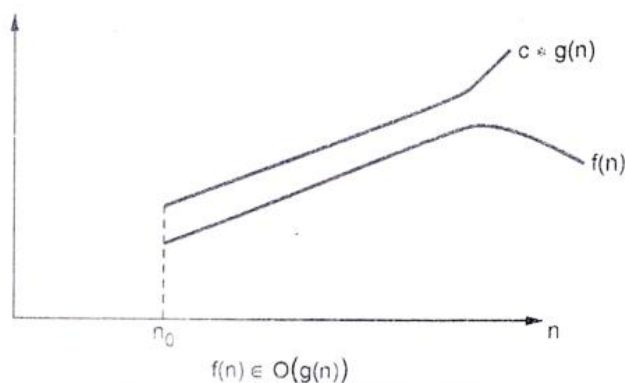


Fig. 2.2.1 Big oh representation

SPPU-BE-COMP-CONTENT – KSKA Git

When do we use O (Big-O) Notation?

- O-notation is used to describe the **asymptotic upper bound** of an algorithm.
- It provides the **worst-case guarantee** on the running time or space.
- It is most commonly used in algorithm analysis because it ensures performance will not exceed the stated bound.
- Example: Bubble sort is **$O(n^2)$** in the worst case.

Example Summary:

- **O** → upper bound (at most).
- **Ω** → lower bound (at least).
- **Θ** → tight bound (exact rate).

Q4) a) What is polynomial time reducibility? What is its importance in computational complexity theory? [6 Marks]

1. Definition:

- **Polynomial Time Reducibility** means problem **A** can be transformed into problem **B** in **polynomial time**, such that solving **B** gives a solution to **A**.
- Notation:

$$A \leq_p B$$

This means: if **B** can be solved in polynomial time, **A** can also be solved in polynomial time.

2. Importance:

- Helps in **classifying problems** into complexity classes (e.g., P, NP, NP-complete).
- Used in proving NP-completeness (reduction from a known NP-complete problem).
- Shows **relative difficulty** between problems.

3. Example:

- Reducing **3-SAT** to **Clique problem** in polynomial time.

Importance in Computational Complexity Theory:

1. Classifying Problems:

SPPU-BE-COMP-CONTENT – KSKA Git

- It helps in categorizing problems into **P, NP, NP-Hard, and NP-Complete** classes.

2. Proving NP-Completeness:

- Polynomial time reducibility is the **main tool** to prove that a problem is NP-Complete.
- If problem **A** (already NP-Complete) can be reduced to problem **B**, and **B** \in NP, then **B** is NP-Complete.

3. Transfer of Hardness:

- It allows us to **transfer difficulty** from one problem to another.
- If one problem is hard, any other problem to which it reduces is also hard.

4. Foundation for Optimization and Approximation:

- It provides the theoretical basis for studying whether a problem can be **approximated** or needs **heuristics**.

Conclusion: Polynomial time reducibility is essential for **identifying hard problems** and proving NP-completeness.

c) Is $6n^3 = \Theta(n^2)$? Justify your answer.

[3 Marks]

Answer:

- Given: $f(n) = 6n^3, g(n) = n^2$
- Definition of Θ :

$$f(n) = \Theta(g(n)) \quad \text{if } \exists c_1, c_2 > 0, n_0 > 0 \text{ such that}$$

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \forall n \geq n_0$$

- Check Growth:

$$\frac{f(n)}{g(n)} = \frac{6n^3}{n^2} = 6n$$

As $n \rightarrow \infty, 6n \rightarrow \infty$.

- This means $6n^3$ grows **faster** than n^2 .

✓ Conclusion:

$$6n^3 \neq \Theta(n^2)$$

Instead:

$$6n^3 = \Omega(n^3) \quad \text{and} \quad 6n^3 = O(n^3)$$