Ques! Discuss the following :-

a. fundamental data structure.
b. Algorithm design Paradigms
c. Properties of Asymptotic Notation.

Ans. a. Fundamental Data Structure :-

Fundamental data structures are the foundational building blocks of computer science and programming. They are essential for storing, organizing, and manipulating data efficiently.
Some key fundamental data structures include:

→ Arrays: Ordered collections of elements with constant-time access to individual elements.
→ Linked Lists: Chains of nodes where each node points to the next, allowing dynamic memory allocation.
→ Stacks: LIFO (Last-in-first-out) data structures often used for managing function calls or undo operations.
→ Queues: FIFO (first-in-first-out) structures used for tasks like task scheduling or breadth-first search.
→ Trees: Hierarchical structures with nodes connected by edges, commonly used for searching and organizing data.
→ Graphs: Collections of nodes and edges used for modelling complex relationships and network structures.

b. Algorithm Design Paradigms :-

Algorithm design paradigms are strategies or approaches used to solve problems algorithmically. Some common algorithm design paradigms include:

- ↔ Divide and Conquer: Break a problem into smaller subproblems, solve them recursively, and combine their solutions to solve the original problem.
- ↔ Greedy Algorithms: Make a series of locally optimal choices at each step to construct a globally optimal solution.
- ↔ Dynamic Programming: Solve a problem by breaking it into smaller overlapping subproblems, solving each subproblem only once, and storing their solutions for future use.
- ↔ Brute Force: Generate all possible solutions and choose the best one, often suitable for small input sizes.

6. Properties of Asymptotic Notation :-

Asymptotic notation is used to describe the growth rate of algorithms in terms of input size.

1) Three common notations are Big O (O), Omega (Ω) and theta (θ).

Here are some properties of asymptotic notation:

1. Big O (O) Notation:
   - Represents the upper bound on the algorithm's running time.
   - $O(g(n))$ describes an upper limit on the running time, where $g(n)$ is a function.
   - It provides an "at most" analysis.

2. Omega (Ω) Notation:
   - Represents the lower bound on the algorithm's running time.
   - $\Omega(g(n))$ describes a lower limit on the running time, where $g(n)$ is a function.
   - It provides an "at least" analysis.

3. Theta (Θ) Notation:
   - Represents both upper and lower bounds on the algorithm's running time.
   - $\Theta(g(n))$ describes a tight bound on the running time, where $g(n)$ is a function.
   - It provides a precise analysis of the algorithm's growth rate.

Properties

1. Transitivity :

If one function grows slower than another and that one grows slowers than a third, then the first grows slower than the third

Mathematical :

If $f(n)$ is $O(g(n))$ and $g(n)$ is $O(h(n))$, then $f(n)$ is $O(h(n))$.

Similarly, for Ω and Θ notations.

2. Symmetry :

If one function is a good fit for another, then the other is also a good fit for the first.

Mathematical :

If $f(n)$ is $\Theta(g(n))$, then $g(n)$ is also $\Theta(f(n))$.

3. Reflexivity :

Any function is both at least as slow as itself and at most as slow as itself

Mathematical :

For any function $f(n)$, $f(n)$ is both $\Omega(f(n))$ and $O(f(n))$.

Scanned with OKEN Scanner

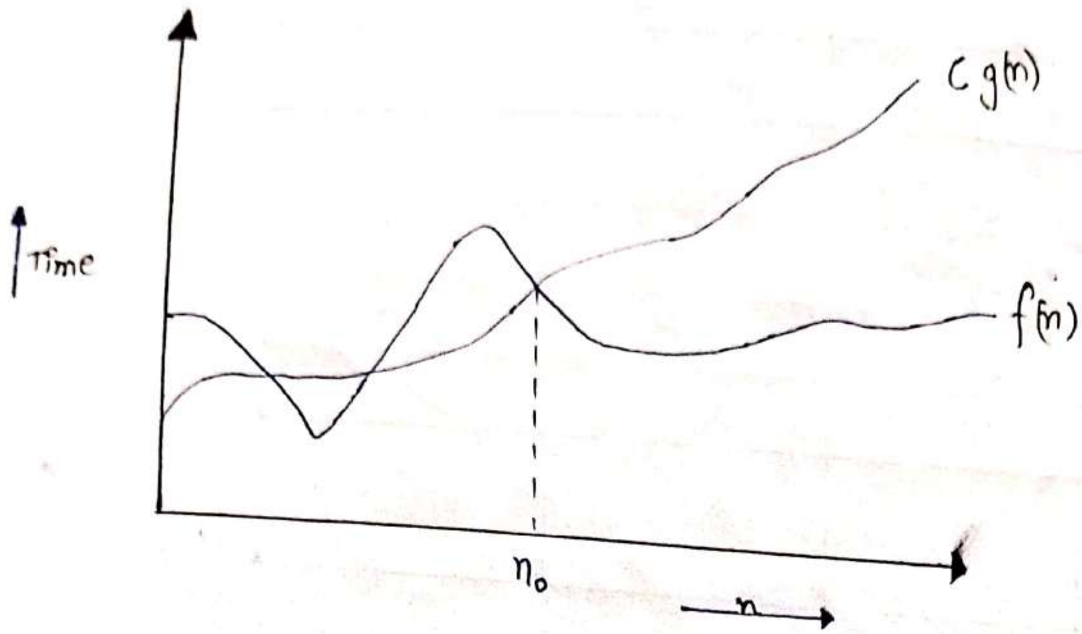
	Reflexive	Symmetry	Transitive
O	✓	✗	✓
Ω	✓	✗	✓
θ	✓	✓	✓
o	✗	✗	✓
ω	✗	✗	✓

Ques 2. Write down mathematically definition of big O, Big omega and Big theta with suitable example.

Ans

1. Big Oh (O) Notation :

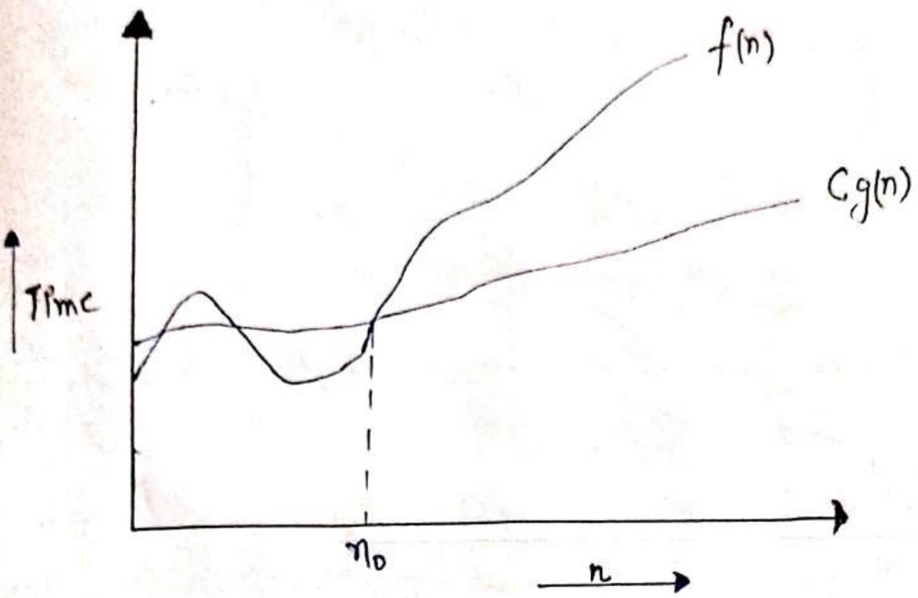
- Mathematical Definition :
  - $f(n)$ is $O(g(n))$ if and only if there exist positive constants c and n_0 such that $0 \le f(n) \le c * g(n)$ for all $n \ge n_0$
- Example :
  - Suppose we have an algorithm with a time complexity of $O(n^2)$. This means that for large enough values of n, the running time of the algorithm is at most $c * n^2$, where c is a positive constant



2. Big omega (Ω) Notation:

- Mathematical Definition:
  - $f(n)$ is $\Omega(g(n))$ if and only if there exist positive constants c and n_0 such that $0 \le c\,g(n) \le f(n)$ for all $n \ge n_0$
- Example:
  - Consider an algorithm with a time complexity of $\Omega(n)$. This implies that for sufficiently large n, the algorithm's running time is at least $c*n$, where c is a positive constant.

$$\left[0 \le c\,g(n) \le f(n) \text{ for all } n \ge n_0 \right]$$



3. Big Theta (θ) Notation:

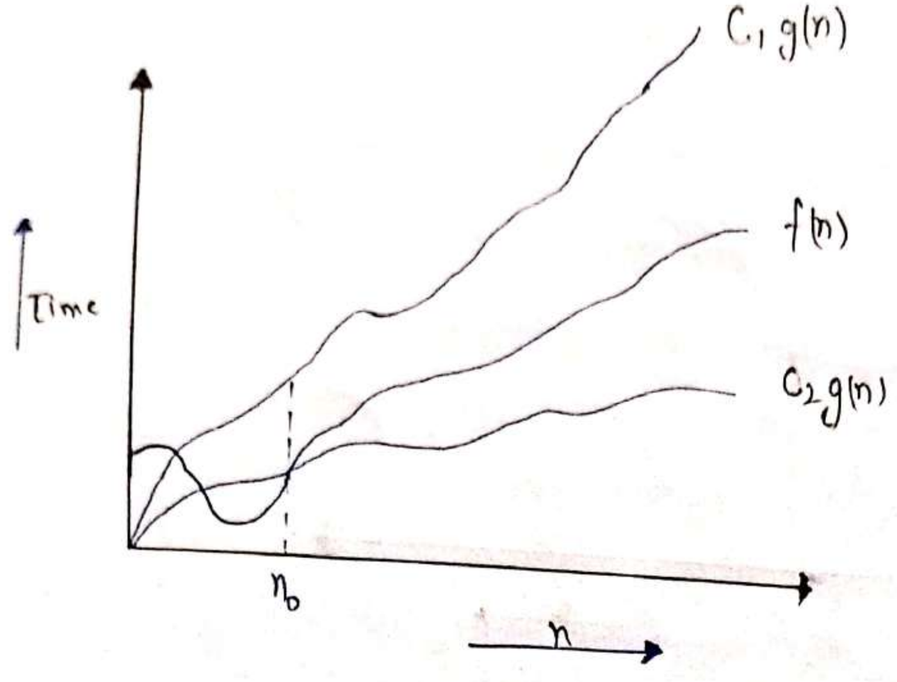
- Mathematical Definition:
  - $f(n)$ is $\theta(g(n))$ if and only if there exist positive constants c_1, c_2, and n_0 such that $0 \le c_1 * g(n) \le f(n) \le c_2 * g(n)$ for all $n \ge n_0$.
- Example:
  - If an algorithm has a time complexity of $\theta(n)$, it means that for large enough n, the running time is bounded both above and below by $c_1 * n$ and $c_2 * n$, where c_1 and c_2 are positive constants.

$$0 \le f(n) \le c_1 g(n) \quad \text{for } n \ge n_0$$

$$0 \le c_2 g(n) \le f(n) \quad \text{for } n \ge n_0$$

Merge both eq.

$$0 \le c_2 g(n) \le f(n) \le c_1 g(n) \quad \text{for } n \ge n_0$$



Ques 3. What is an Algorithm? Explain its characteristics.

An algorithm is a well-defined, step-by-step procedure or set of instructions designed to solve a specific problem or perform a particular task. It is like a recipe for solving a problem. It's a set of clear and specific instructions that tell you exactly what to do step by step.

1. Clear Instructions: An algorithm is like a recipe with very clear steps. It tells you exactly what to do at each step, like "mix ingredients" or "count to 10".

2. Start and Finish: It must have a clear beginning and an end. Just like a recipe starts with gathering ingredients and ends with a finished dish, an algorithm starts with some information and ends with a solution.

3. No Guesswork: You shouldn't have to guess what to do next. It should be like following a map; each step is defined, and you know where you're going.

4. Works for Everyone: The instructions in an algorithm should work for anyone who follows them. Its like if you give the same recipe to different people, they should all make the same dish.

5. Not too Long: An algorithm can't be endless. It should finish in a reasonable amount of time. Imagine you're telling a story; it can't go on forever; it must have an ending.

6. Solves a Problem: Algorithms are used to solve problems. Think of them as problem-solving tools. They take some information and transform it into a solution or answer.

7. No Randomness: It doesn't involve guessing or randomness. Every step should be certain and predictable.

8. Efficient: Completes the tasks in a efficient manner.

9. Consistent Results: If you use the same algorithm multiple times with the same input, it should give you the same result every time.

Ques 4 What is Asymptotic Notation. Explain small or little oh, omega Notation

Asymptotic Notation is a way of describing how the runtime or resource usage of an algorithm grows as the input size becomes very large.

It simplifies the analysis of algorithms by focusing on their behavior as they approach infinity.

Three common asymptotic notations are Big O (O), Omeg (Ω) and Theta (θ), but there are also "little oh" (o) and Omega (ω)

1. Little oh (o) Notation:

- Little oh (o) notation describes a tighter upper bound than Big O(O) notation. It indicates that an algorithm's performance grows strictly slower than a given function, but the difference is not too big.
- Mathematically, $f(n)$ is in $o(g(n))$ if, for any positive constant $c > 0$, there exists a point n_0 such that for all $n > n_0$, $0 \leq f(n) < c\,g(n)$
- In simpler terms, $o(g(n))$ means that $g(n)$ grows significantly faster than $f(n)$, but there's still a gap b/w them.

Page-1
or little

2. Little Omega (ω) Notation:

- Little omega (ω) notation describes a tighter lower bound than Big omega (Ω) notation. It indicates that an algorithm's performance grows strictly faster than a given function.
- Mathematically, $f(n)$ is in $\omega(g(n))$ if, for any positive constant $c > 0$, there exists a point n_0 such that for all $n > n_0$, $0 \leq c * g(n) < f(n)$
- In simpler terms, $\omega(g(n))$ means that $f(n)$ grows significantly faster than $g(n)$, leaving a gap b/w them.

In summary "little o" (o) and "little omega" (ω) notations provide more precise information about how an algorithm's growth rate compares to a given function, offering tighter bounds than their counterparts (Big O and Big omega)