

# **Name- Bhavya Shah Roll no.- 16010424125 Batch - B3**

1. Efficient testing and debugging? Explain.

Ans)

Efficient testing is the process of systematically checking a program using well-chosen test cases to ensure that it produces correct output for all valid inputs with minimum time and effort.

Purpose of Efficient Testing

- To verify program correctness
- To detect errors before execution failures
- To ensure the program works for all possible cases
- To improve program reliability and performance

Types of Test Cases Used

1. Sample test cases - Provided in the problem statement
2. Normal test cases - Typical valid inputs
3. Edge cases
  - Minimum values
  - Maximum values
  - Empty input
  - Single element
4. Boundary cases - Values just inside and outside constraints

Techniques for Efficient Testing

- Test small inputs first
- Use dry run (manual execution)
- Test independent modules separately
- Validate input and output formats

Example - If a program finds the maximum element in an array:

- Test with sorted array
- Test with all elements same
- Test with one element
- Test with maximum array size

Debugging is the process of identifying, analyzing, and fixing errors (bugs) in a program after they are discovered during testing or execution.

## Purpose of Debugging

- To locate the exact source of an error
- To correct logical, syntax, and runtime errors
- To ensure the program runs without crashing

## Types of Errors Handled

1. Syntax errors - Errors in code structure (missing semicolon, brackets)
2. Logical errors - Program runs but gives incorrect output
3. Runtime errors - Errors during execution (division by zero, segmentation fault)

## Debugging Techniques

- Using print statements
- Step-by-step execution
- Using debugger tools (breakpoints)
- Checking variable values at runtime

## Debugging Process

1. Reproduce the error
2. Isolate the faulty code section
3. Fix the error
4. Re-test the program

## 2. Quickly identify the problem type and analyze the algorithm

Ans)

Quickly identifying the problem type helps in choosing the most suitable algorithm, reducing unnecessary trial and error.

### Steps to Identify the Problem Type

1. Read the problem statement carefully
  - Understand what is asked: find, count, maximize, minimize, sort, or search.
2. Analyze the input and output
  - Data type (array, string, graph, matrix)
  - Input size and constraints
3. Look for keywords
  - Shortest, minimum, maximum - Optimization
  - Sorted array - Binary search
  - Subproblem repetition - Dynamic Programming
  - Traversal - BFS / DFS
4. Check constraints
  - Small input - Brute force

- Large input - Optimized algorithm required

### Common Problem Types and Algorithms

Problem Type	Suitable Algorithm
Searching	Linear Search, Binary Search
Sorting	Merge Sort, Quick Sort
Optimization	Greedy, Dynamic Programming
Graph problems	BFS, DFS, Dijkstra
Counting problems	Hashing, Prefix Sum

### Algorithm Analysis

Once the algorithm is selected:

- Verify correctness
- Analyze time complexity
- Analyze space complexity
- Ensure it meets given constraints

### Example -

If  $n = 10^5$ , using two nested loops gives  $O(n^2)$  which is inefficient.

Hence, an optimized algorithm like  $O(n \log n)$  is used.

3. Calculate rules for time complexity with examples and also define common time complexities and how you estimate efficiency

Ans)

Time complexity measures the amount of time an algorithm takes to run as a function of input size  $n$ .

### Rules for Calculating Time Complexity

Rule 1: Ignore constants

- $O(3n) = O(n)$
- $O(10) = O(1)$

Rule 2: Consider the highest-order term

- $O(n^2 + n + 5) = O(n^2)$

Rule 3: Sequential statements → Add time

- $\text{for}(i = 0; i < n; i++) // O(n)$

```

printf("A");
for(i = 0; i < n; i++) // O(n)
printf("B");

```

- Total =  $O(n + n) = O(n)$

Rule 4: Nested loops → Multiply time

- ```

for(i = 0; i < n; i++)
  for(j = 0; j < n; j++)
    printf("*");

```
- Time Complexity =  $O(n^2)$

Rule 5: Conditional statements

- Take the worst-case time complexity

Common Time Complexities

| Time Complexity | Name         | Example             |
|-----------------|--------------|---------------------|
| $O(1)$          | Constant     | Array access        |
| $O(\log n)$     | Logarithmic  | Binary search       |
| $O(n)$          | Linear       | Linear search       |
| $O(n \log n)$   | Linearithmic | Merge sort          |
| $O(n^2)$        | Quadratic    | Bubble sort         |
| $O(2^n)$        | Exponential  | Recursive Fibonacci |
| $O(n!)$         | Factorial    | Brute force TSP     |

Estimating Algorithm Efficiency

- Count the number of loops
- Check nested loops
- Analyze recursion depth
- Compare complexity with input constraints
- Choose the fastest feasible algorithm