GUJARAT TECHNOLOGICAL UNIVERSITY

BE-5 SEMESTER – OLD PAPER – S22 TO W25 – QUESTION BANK ANSWER

Subject Name & Code:

ANALYSIS AND DESIGN OF ALGORITHMS (3150703)

Unit 1: Basics of Algorithms and Mathematics

Q1 – Define algorithm and its characteristics. (4 marks)
Ans:
An **algorithm** is a finite, step-by-step procedure to solve a specific problem, independent of any programming language.
Characteristics of an algorithm:

- **Finiteness** – Must terminate after a finite number of steps.
- **Definiteness** – Each step must be precisely and unambiguously defined.
- **Input** – Accepts zero or more inputs.
- **Output** – Produces at least one output.
- **Effectiveness** – Each step must be basic enough to be performed exactly in finite time.

Real-world application: Sorting a list of student roll numbers manually uses an algorithm (e.g., insertion sort).

Q2 – Explain worst-case, best-case, and average-case complexity with an example. (4 marks)
Ans:
These are measures of how an algorithm's running time grows with input size (n).

Case	Meaning	Example: Linear Search (key present or not)
Best-case	Minimum time required. Input that allows fastest completion.	Key is the **first element** → comparisons = 1 → O(1)
Worst-case	Maximum time required. Input that forces most steps.	Key is **last element** or **absent** → comparisons = n → O(n)
Average-case	Expected time over all possible inputs (probabilistic).	On average, key found in n/2 comparisons → O(n)

Key takeaway: Worst-case is most commonly used for comparing algorithms because it guarantees an upper bound.

Q3 – Define time complexity and space complexity. (4 marks)
Ans:

Complexity	Definition	Example
Time complexity	The amount of computer time an algorithm needs as a function of input size (n). Measured in number of basic operations (comparisons, assignments).	Bubble sort – $O(n^2)$ time
Space complexity	The amount of memory an algorithm needs (including input, auxiliary space, and stack) as a function of input size.	Recursive Fibonacci – O(n) stack space; Iterative – O(1) space

Note: Space complexity = Fixed part (constants, instructions) + Variable part (input, dynamic memory). Trade-off: sometimes we use extra space to reduce time (e.g., memoization).

Q4 – What is an algorithm? Explain various properties of an algorithm. (4 marks)
Ans:
An **algorithm** is a well-defined, ordered set of instructions that transforms inputs into outputs, solving a computational problem.
Properties (Characteristics) – expanded:

1. **Finiteness** – Must terminate after finite steps; infinite loops are not allowed.
2. **Definiteness** – Every step must be clear, unambiguous, and free from confusion.
3. **Input** – Takes zero or more externally supplied values.
4. **Output** – Produces at least one result.
5. **Effectiveness** – Each step must be feasible (can be done manually on paper in finite time).
6. **Generality** – Should work for all valid inputs, not just a specific case.

Example: A recipe for making tea is an algorithm – steps are finite, definite, effective, produce tea (output).

Q5 – Explain the need for amortized analysis with a suitable example. (7 marks)
Ans:
Amortized analysis averages the time required over a sequence of operations, rather than analysing each operation individually. It gives a realistic upper bound when some operations are expensive but occur rarely.
Need for amortized analysis:

- Worst-case analysis of a single operation may be high (e.g., O(n)), but a sequence of operations may have much lower average cost.
- Helps design data structures with **occasional expensive operations** but overall efficiency.
- Provides a **tight bound** for dynamic data structures (e.g., dynamic arrays, splay trees, disjoint sets).

Suitable example – Dynamic Array (e.g., C++ vector, Java ArrayList):

- Operation: push_back(x) – append element.
- When array has free capacity → O(1) time.
- When array is full → allocate new array of double size, copy all n elements → O(n) time (expensive).

Amortized analysis using Aggregate Method:

- Sequence of n push_back operations.
- Expensive copies happen at sizes 1, 2, 4, 8, … (powers of two).
- Total cost = n inserts + sum of copy costs.
- Sum of copy costs = $1 + 2 + 4 + \ldots + n/2 + n \approx 2n$.

- Total cost ≈ n + 2n = 3n.
- **Amortized cost per operation = 3n / n = O(1).**

Conclusion: Even though one push_back can cost O(n), the **average** over many operations is O(1). This justifies using dynamic arrays in practice.
Real-world application: Text editors (undo/redo stacks), hash tables with rehashing, garbage collection.

Q6 – Sort the best-case running times of given algorithms in non-decreasing order. (3 marks)
List of algorithms: LCS, Quick-Sort, Merge-Sort, Counting-Sort, Heap-Sort, Selection-Sort, Insertion-Sort, Bucket-Sort, Strassen's Algorithm.
Best-case time complexities:

Algorithm	Best-case time
Counting-Sort	O(n + k) (k = range of keys)
Bucket-Sort	O(n + k) (uniform distribution)
Insertion-Sort	O(n) (already sorted)
LCS (Longest Common Subsequence)	O(mn) (always, no best-case improvement)
Quick-Sort	O(n log n) (balanced partition)
Merge-Sort	O(n log n) (always)
Heap-Sort	O(n log n) (always)
Selection-Sort	O(n^2) (always)
Strassen's Algorithm	O(n^2.81) (matrix multiplication, best case = worst case)

Non-decreasing order (by asymptotic growth):

1. **Counting-Sort** – O(n + k) (if k = O(n) → O(n))
2. **Bucket-Sort** – O(n + k) (if k = O(n) → O(n))
3. **Insertion-Sort** – O(n)
4. **LCS** – O(mn) (typically O(n^2) for equal lengths)
5. **Quick-Sort** – O(n log n)
6. **Merge-Sort** – O(n log n)
7. **Heap-Sort** – O(n log n)
8. **Strassen's Algorithm** – O(n^2.81)
9. **Selection-Sort** – O(n^2)

Note: For equal complexities (e.g., O(n log n)), order is not unique; constants ignored. Counting-Sort and Bucket-Sort are linear under ideal conditions.

Q7 – State whether the statements are correct or incorrect with reasons. (Marks not specified)
⚠️ **Incomplete question:** The CQB lists only the instruction "State whether the statements are correct or incorrect with reasons" but **does not provide any statements**. Please provide

the exact statements to receive a proper evaluation.
Example format (if statements were given):
Statement: "An algorithm must have at least two inputs."
Incorrect. Reason: An algorithm can have zero inputs (e.g., a program that prints "Hello World").

Q8 – Explain asymptotic analysis with all notations and mathematical inequalities. (4 marks)
Appeared in: W22 (Q1c, 04 marks)
Ans:
Asymptotic analysis describes the limiting behaviour of an algorithm's time/space complexity as input size $n \to \infty$. It ignores constants and lower-order terms to compare growth rates.
Common asymptotic notations & mathematical inequalities:

Notation	Name	Mathematical definition	Meaning
O(g(n))	Big-Oh	$\exists\ c>0, n_0$ such that $0 \le f(n) \le c \cdot g(n)\ \forall\ n \ge n_0$	**Upper bound** (worst-case)
Ω(g(n))	Big-Omega	$\exists\ c>0, n_0$ such that $0 \le c \cdot g(n) \le f(n)\ \forall\ n \ge n_0$	**Lower bound** (best-case)
Θ(g(n))	Theta	$\exists\ c_1,c_2>0, n_0$ such that $c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)\ \forall\ n \ge n_0$	**Tight bound** (both upper & lower)
o(g(n))	Little-oh	$\forall\ c>0, \exists\ n_0$ such that $0 \le f(n) < c \cdot g(n)\ \forall\ n \ge n_0$	Strict upper bound (not asymptotically tight)
ω(g(n))	Little-omega	$\forall\ c>0, \exists\ n_0$ such that $0 \le c \cdot g(n) < f(n)\ \forall\ n \ge n_0$	Strict lower bound (not asymptotically tight)

Examples:
- $2n^2 + 3n + 5 = O(n^3) = O(n^2) = \Theta(n^2)$ (tightest is $\Theta(n^2)$)
- $n \log n = o(n^2)$ ($n \log n$ grows strictly slower than n^2)

Real-world application: Choosing between sorting algorithms – $O(n \log n)$ merge sort vs $O(n^2)$ bubble sort for large datasets.

****