DAA Unit-1 Notes By OMG 13_159 OmGonjari

---

1. Explain different characteristics of algorithm. (4 Marks)

An algorithm is a step-by-step procedure for solving a problem or accomplishing a task. The key characteristics of a good algorithm are:

- **Finiteness:** An algorithm must terminate after a finite number of steps[1]. It should not enter an infinite loop.
- **Definiteness:** Each step of the algorithm must be precisely and unambiguously defined[2]. The instructions should be clear and leave no room for interpretation.
- **Input:** An algorithm must have zero or more well-defined inputs[3]. These are the quantities that are given to the algorithm initially before it begins.
- **Output:** An algorithm must have one or more well-defined outputs[4]. These are the quantities that are produced by the algorithm, and they should be the desired result.
- **Effectiveness:** Each step of the algorithm must be basic enough that it can be carried out in a finite amount of time[5]. The operations should be simple and feasible.

---

2. Why correctness of the algorithm is essential? (4 or 6 Marks)

The correctness of an algorithm is essential because it guarantees that the algorithm will consistently produce the correct output for all valid inputs[6]. Without correctness, an algorithm is essentially useless, as it may provide wrong or unpredictable results, leading to flawed systems, errors in calculations, and potentially harmful consequences in critical applications like medical software, financial systems, or aviation control[7]. Proving correctness provides a mathematical guarantee of the algorithm's reliability.

---

3. Explain iterative algorithm design issues. (4 or 6 Marks)

Iterative algorithms solve problems by repeating a set of instructions until a certain condition is met. Key design issues include:

- **Termination Condition:** Ensuring the loop will eventually terminate[8]. A missing or incorrect termination condition can lead to an infinite loop.

- **Loop Invariant:** Maintaining a property that is true before and after each iteration of the loop[9]. This property is crucial for proving the algorithm's correctness.

- **Initialization:** The variables used in the loop must be correctly initialized before the loop begins[10]. Incorrect initialization can lead to logical errors.

- **Progress:** Each iteration of the loop must make progress toward the termination condition. If the algorithm doesn't move closer to the end, it may not terminate.

---

4. Explain different means of improving efficiency of algorithm. (4 Marks)

The efficiency of an algorithm is measured by its

time complexity (the time it takes to run) and **space complexity** (the memory it uses)[11]. Efficiency can be improved through:

- **Choosing a better algorithm:** For the same problem, different algorithms can have vastly different complexities. For example, a quicksort algorithm is generally more efficient for sorting a large list than a bubble sort.
- **Optimizing the code:** This includes reducing redundant calculations, using more efficient data structures, and minimizing memory allocations.
- **Pre-computation:** Pre-calculating results that will be needed later to avoid re-computation.
- **Parallelism:** Breaking down the problem into smaller parts that can be processed simultaneously on multiple cores or machines.

---

5. Write a short note on Algorithm as a technology. (4 Marks)

Algorithms are a core technology that underpins modern computing. Just like hardware, an algorithm is a creative and powerful tool for solving problems[12]. They are used in countless applications, from search engines and social media feeds to GPS navigation and data encryption. The development of new and more efficient algorithms often drives technological advancements, enabling faster processing, better resource utilization, and the solution of problems that were previously intractable. The study and design of algorithms are fundamental to computer science and technological innovation[13].

---

6. Write a short note on Evolution of Algorithm. (4 Marks)

The concept of algorithms dates back centuries, with early examples found in ancient mathematics, like the Euclidean algorithm for finding the greatest common divisor[14]. The modern era of algorithms began with the development of computers, which required explicit instructions to perform tasks. The field has evolved from a focus on simple mathematical procedures to a sophisticated discipline encompassing complex data structures, parallel processing, and artificial intelligence. The evolution is marked by the development of formal methods for analysis, the discovery of new design paradigms, and the increasing reliance on algorithms for nearly every aspect of modern life.

---

7. What are the general rules followed while writing the algorithm? (4 Marks)

When writing an algorithm, it's important to follow these general rules to ensure clarity and correctness:

- **Start and End:** The algorithm must have a clear starting and ending point.
- **Step-by-step:** Break down the problem into a sequence of small, manageable steps.
- **Clarity and Simplicity:** Each step should be unambiguous and easy to understand.
- **Input and Output:** Clearly define the inputs the algorithm takes and the outputs it produces[15].

- **Language Independence:** An algorithm should be written in a way that is not tied to a specific programming language. Pseudocode is often used for this purpose.
- **Correctness:** The algorithm must be logically sound and produce the correct output for

all valid inputs.

---

8. How to confirm correctness of algorithm also explain with example? (4 Marks)

The correctness of an algorithm can be confirmed by using formal methods such as

mathematical induction, **proof by contradiction**, and **loop invariants**[16]. A common method is using

loop invariants for iterative algorithms.

Example: Summation of n numbers

Algorithm:

```
SUM(A, n)
 sum = 0
 for i = 1 to n:
   sum = sum + A[i]
 return sum
```

Proof using Loop Invariant:
Let the loop invariant be: "At the start of each iteration i, the variable sum holds the sum of the elements from A[1] to A[i−1]."

- **Initialization:** Before the first iteration (i=1), sum is 0. The sum of elements from A[1] to A[0] is an empty sum, which is 0. The invariant holds.
- **Maintenance:** Assume the invariant holds at the start of iteration i. In the loop body, we add A[i] to sum. The new value of sum will be the sum of elements from A[1] to A[i−1] plus A[i], which is the sum of elements from A[1] to A[i]. This is exactly the invariant for the next iteration.
- **Termination:** The loop terminates when i=n+1. The invariant says that at the start of this iteration, sum holds the sum of elements from A[1] to A[n]. This is the correct final result, proving the algorithm is correct.

---

9. Explain the concept of PMI and prove the correctness of an algorithm to find factorial of a number using PMI. (6 Marks)

Principle of Mathematical Induction (PMI): PMI is a mathematical proof technique used to prove a statement is true for all natural numbers. It involves two steps:

1. **Base Case:** Prove the statement is true for the first natural number (usually n=0 or n=1).
2. **Inductive Step:** Assume the statement is true for an arbitrary natural number k (the **inductive hypothesis**). Then, prove that the statement must also be true for the next number, k+1.

Algorithm for Factorial:

```
FACTORIAL(n)
 if n == 0:
   return 1
 else:
   return n * FACTORIAL(n-1)
```

Proof using PMI:
Let P(n) be the statement that the FACTORIAL(n) algorithm correctly computes n for all integers ngeq0.

1. **Base Case:** For n=0, the algorithm returns 1. By definition, 0=1. So, P(0) is true.
2. **Inductive Step:** Assume that P(k) is true for some arbitrary integer kgeq0. That is, FACTORIAL(k) correctly computes k. We need to prove that P(k+1) is true, i.e., FACTORIAL(k+1) correctly computes (k+1).
   - According to the algorithm, FACTORIAL(k+1) returns $(k+1) * FACTORIAL(k)$.
   - By our inductive hypothesis, FACTORIAL(k) returns k.
   - Therefore, FACTORIAL(k+1) returns $(k+1) * k\!$.
   - By the definition of factorial, $(k+1)\! = (k+1) * k\!$.
   - Thus, the algorithm correctly computes (k+1), and P(k+1) is true.

Since the base case and the inductive step hold, by the Principle of Mathematical Induction, the algorithm is correct for all non-negative integers.

---

10. Contrast and compare between iterative and recursive process

with an example. (4 or 6 Marks)

Feature	Iterative Process	Recursive Process
Structure	Uses loops (for, while) to repeat a block of code[17].	Calls itself repeatedly with smaller inputs until a base case is reached[18].
Memory	Uses a fixed amount of memory regardless of the input size (constant space)[19].	Uses the call stack, which grows with each recursive call, potentially leading to stack overflow[20].
Complexity	Often more difficult to write for complex problems.	Can be more elegant and concise for problems that have a self-similar structure.
Control Flow	Explicitly controlled by the loop counter.	Implicitly controlled by the function call and return mechanism.

Example: Finding the factorial of a number

Iterative:

```
function factorial_iterative(n):
  result = 1
  for i from 1 to n:
    result = result * i
  return result
```

Recursive:

```
function factorial_recursive(n):
 if n == 0:
  return 1
 else:
  return n * factorial_recursive(n-1)
```

---

11. Explain the importance of algorithm in computing with example. (4 Marks)

Algorithms are the backbone of computing. They are the logical procedures that tell a computer what to do, how to process data, and how to arrive at a solution. Without algorithms, a computer is just a piece of hardware; it is the algorithm that gives it purpose and functionality. They are essential for:

- **Problem Solving:** Breaking down complex problems into manageable steps.
- **Efficiency:** Determining the best way to solve a problem with minimum time and resources. For example, a search engine uses a sophisticated algorithm to quickly find relevant information from billions of webpages. A brute-force search would be too slow.
- **Innovation:** The development of new algorithms has enabled major advancements in fields like artificial intelligence, cryptography, and data science.

---

12. Explain the various algorithm design methodology to solve a problem. (6 Marks)

Algorithm design methodologies are systematic approaches to creating algorithms. Some common ones include:

- **Divide and Conquer:** This strategy breaks a problem into smaller subproblems of the same type, solves them recursively, and then combines their solutions to get the final answer. Examples include **Merge Sort** and **Quick Sort**.
- **Greedy Approach:** This method makes the locally optimal choice at each step with the hope of finding a global optimum. It doesn't always guarantee the best solution but is often fast and simple. **Dijkstra's algorithm** for finding the shortest path is a classic

example.

- **Dynamic Programming:** This is used for problems that have overlapping subproblems. It solves each subproblem only once and stores the results in a table to avoid re-computation. It's often used for optimization problems. The **Fibonacci sequence** can be solved efficiently with dynamic programming.
- **Backtracking:** This is a systematic search algorithm that explores all possible solutions. It builds a solution incrementally and, if a partial solution cannot lead to a complete solution, it "backtracks" to a previous state and tries a different path. This is used in problems like the **N-Queens problem** and Sudoku solvers.

---

13. Explain Big O, Omega and Theta notations. Explain what are they used for. (4 or 6 Marks)

These are asymptotic notations used to describe the running time or space requirements of an algorithm as the input size grows. They provide a way to classify algorithms based on their growth rate, which is essential for comparing their efficiency.

- **Big O notation (O):** Describes the **upper bound** of an algorithm's running time[21]. It provides a worst-case scenario. If an algorithm is

  O(n2), its running time will not grow faster than a quadratic function of the input size n. For example, an algorithm with O(n2) complexity is acceptable for small inputs, but for larger inputs, its performance degrades quickly.
- **Big Omega notation (Omega):** Describes the **lower bound** of an algorithm's running time[22]. It represents the best-case scenario. If an algorithm is

  Omega(n), its running time will be at least a linear function of the input size n.
- **Theta notation (Theta):** Describes the **tight bound** of an algorithm's running time[23]. It represents both the upper and lower bounds, giving an exact asymptotic behavior. If an algorithm is

  Theta(nlogn), its running time is always proportional to nlogn for large inputs. It is the most precise of the three notations.

---

14. How do we analyse algorithms? Explain what is meant by space and time complexity. (4 or 6 Marks)

Algorithm analysis is the process of determining the resources (time and memory) required by an algorithm to solve a given computational problem. It allows us to predict an algorithm's performance and compare it to other algorithms.

- **Time Complexity:** This measures the amount of time an algorithm takes to run as a function of the input size[24]. It is typically expressed using asymptotic notations like Big O, Big Omega, and Theta. We don't measure time in seconds, but rather in terms of the number of elementary operations (like arithmetic operations, comparisons, or assignments) performed.

- **Space Complexity:** This measures the amount of memory an algorithm requires to run as a function of the input size[25]. It includes both the space required to store the input and the auxiliary space used by the algorithm (e.g., for variables, data structures, or the call stack).

---

15. Discuss different ways of algorithm design with suitable example. (6 Marks)

(This question is similar to question 12. The answer below combines and elaborates on the concepts from that question.)

Algorithm design is a creative process of devising a method to solve a problem. The choice of design strategy often depends on the problem's nature.

- **Greedy Algorithms:** The greedy approach always makes the choice that looks best at the moment. For example, in the **coin change problem**, a greedy algorithm would always pick the largest denomination coin that is less than or equal to the remaining amount. While simple, this approach doesn't work for all coin systems.
- **Divide and Conquer:** This is a powerful technique for breaking down problems into smaller, more manageable subproblems. **Merge Sort** is a prime example: it splits an array into two halves, recursively sorts them, and then merges the sorted halves.
- **Dynamic Programming:** This is used for problems with optimal substructure and overlapping subproblems. The **knapsack problem** is a classic example: to find the most valuable combination of items that fit in a knapsack, a dynamic programming solution builds up a table of optimal solutions for smaller knapsacks and smaller subsets of items.
- **Backtracking:** This is used for finding all possible solutions to a problem, typically constraint satisfaction problems. The **N-Queens problem** is a good example. The algorithm places queens one by one and "backtracks" if a queen is placed in a position that is attacked by another.

16. Why correctness of the algorithm is essential? Define Loop Invariant property and prove the correctness of finding summation of n numbers using loop invariant property. (8 Marks)

Why Correctness is Essential:
The correctness of an algorithm is paramount because it ensures the algorithm is reliable and trustworthy26. An incorrect algorithm can produce unpredictable or wrong results, which can have catastrophic consequences in fields like medicine, finance, and engineering. A proven-correct algorithm provides a formal guarantee that it will behave as expected for all valid inputs27. This reliability is the foundation of robust and secure software systems.
Loop Invariant Property:
A
loop invariant is a property that holds true before and after each iteration of a loop[28]. It is a crucial tool for proving the correctness of iterative algorithms. To use a loop invariant, you must prove three things:

1. **Initialization:** The invariant is true before the first iteration of the loop begins[29].
2. **Maintenance:** If the invariant is true before an iteration, it remains true after the iteration[30].
3. **Termination:** When the loop terminates, the invariant provides a useful property that helps to prove the algorithm's correctness[31].

Proof of Correctness for Summation of N Numbers:
Algorithm:

```
SUM(A, n)
 sum = 0
 for i = 1 to n:
   sum = sum + A[i]
 return sum
```

Loop Invariant: At the start of each iteration i, the variable sum holds the sum of the elements from A[1] to A[i−1][32].

- **Initialization:** Before the first iteration, i=1 and sum is 0[33]. The sum of elements from

  A[1] to A[1−1] (i.e., A[0]) is an empty sum, which is 0. The invariant holds.
- **Maintenance:** Assume the invariant holds at the start of iteration i. In the loop, we add A[i] to sum[34]. The new value of

  sum becomes the sum of elements from A[1] to A[i−1] plus A[i], which is the sum of elements from A[1] to A[i]. At the start of the next iteration, the loop counter will be i+1, and sum will hold the sum of elements from A[1] to A[(i+1)−1] (which is A[i])[35]. The invariant holds for the next iteration.

- **Termination:** The loop terminates when i=n+1[36]. At the start of this iteration, the invariant states that

  sum holds the sum of elements from A[1] to A[(n+1)−1], which is the sum of elements from A[1] to A[n][37]. This is the correct result of summing the first

  n numbers. Thus, the algorithm is correct.

---

17. Write a short note on any 4 problem solving strategies. (8 Marks)

(This is a repetition of concepts from questions 12 and 15, so the answer will be a more concise summary).

Problem-solving strategies, or algorithm design paradigms, are common approaches used to develop algorithms. Here are four key strategies:

1. **Divide and Conquer:** This strategy breaks a complex problem into smaller, identical subproblems. These subproblems are solved independently and then their results are combined to solve the original problem. This recursive approach is highly effective for tasks like sorting (e.g., **Merge Sort**) and searching.
2. **Greedy Approach:** A greedy algorithm makes the best possible choice at each stage of a problem, hoping that this series of local optimal choices will lead to a globally optimal solution. While simple and fast, this strategy doesn't always guarantee the absolute best outcome for all problems but is often used in optimization scenarios like finding the shortest path in a network with positive edge weights (**Dijkstra's Algorithm**).
3. **Dynamic Programming:** This technique is used for problems with overlapping

subproblems and optimal substructure. Instead of re-computing solutions for the same subproblems repeatedly, dynamic programming stores the results of these subproblems in a table. This memoization significantly improves efficiency, particularly in problems like the **knapsack problem** or calculating the **Fibonacci sequence**.

4. **Backtracking:** This is a systematic search strategy used to find a solution to a problem by exploring all possible candidates. It builds a solution incrementally and "backtracks" (reverts to a previous state) whenever it determines that a partial solution cannot be completed to a valid final solution. It's often used for solving puzzles and constraint satisfaction problems, such as the **N-Queens problem** or Sudoku.