**Unit 1**

**DAA**

**Introduction to Algorithms**

* **Definition**:  
  An algorithm is a finite, step-by-step set of instructions to solve a specific problem or perform a computation.
* **Key Characteristics**:
  1. **Definiteness**: Each step is unambiguous.
  2. **Input**: Zero or more well-defined inputs.
  3. **Output**: At least one output.
  4. **Finiteness**: Terminates after finite steps.
  5. **Effectiveness**: Steps are basic and feasible.
* **Importance of Algorithms**:
  1. Foundation of computer science.
  2. Ensures efficiency (time/space) and correctness in problem-solving.

**Steps in Developing an Algorithm**

**Step 1: Problem Definition**

* Clearly understand the problem.
* Specify **what needs to be solved** (e.g., sorting, searching).
* Example: "Sort a list of numbers in ascending order."

**Step 2: Input/Output Specification**

* Define **inputs** (e.g., unsorted array) and **outputs** (e.g., sorted array).
* Example:
  + *Input*: A = [5, 2, 9]
  + *Output*: A = [2, 5, 9]

**Step 3: Choose Design Strategy**

* Select an **algorithm design technique** based on the problem:
  + Divide and Conquer
  + Greedy Method
  + Dynamic Programming
  + Backtracking
  + Brute Force
* Example: Use *Divide and Conquer* for sorting (e.g., Merge Sort).

**Step 4: Algorithm Design**

* Develop step-by-step instructions using **pseudocode** or flowcharts.
* Focus on **logic**, not syntax.
* Example:



**Step 5: Correctness Proof**

* Verify that the algorithm works for **all valid inputs**.
* Use mathematical induction or logical reasoning.

**Step 6: Analyze Efficiency**

* **Time Complexity**: Measure steps as a function of input size (e.g., O(n log n) for Merge Sort).
* **Space Complexity**: Memory usage (e.g., O(n) for Merge Sort).

**Step 7: Optimization**

* Refine the algorithm to improve efficiency.
* Example: Reduce redundant steps or use memoization.

**Step 8: Implementation**

* Convert the algorithm into a programming language (e.g., C++, Python).

**Step 9: Testing & Validation**

* Test with different inputs (best case, worst case, edge cases).
* Example: Test sorting with empty arrays, duplicates, etc.

**Step 10: Documentation**

* Maintain clear records of the algorithm’s logic, assumptions, and limitations.

**Methods of Specifying an Algorithm**

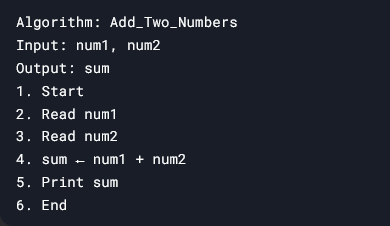
An algorithm must be specified clearly to ensure **correct implementation**. Below are the standard methods described:

**1. Natural Language**

* **Description**:
  + Algorithms are written in simple, human-readable language (e.g., English).
  + Informal and easy to understand but prone to ambiguity.
* **Example**:  
  \*To find the sum of two numbers:
  1. Read the first number.
  2. Read the second number.
  3. Add the two numbers.
  4. Display the result.
* **Advantages**:
  + No technical knowledge required.
  + Suitable for initial brainstorming.
* **Disadvantages**:
  + Ambiguity in complex steps.
  + Not suitable for precise implementation.

**2. Pseudocode**

* **Description**:
  + A **structured mix of natural language and programming constructs** (e.g., loops, conditions).
  + Focuses on logic, not syntax.
* **Example:**



* **Advantages**:
  + Less ambiguous than natural language.
  + Bridges gap between theory and code.
* **Disadvantages**:
  + No universal standard (varies by author).

**3. Flowcharts**

* **Description**:
  + Visual representation using **symbols** (e.g., rectangles for steps, diamonds for decisions).
  + Arrows show flow of control.
* **Symbols**:
  + **Oval**: Start/End
  + **Rectangle**: Process/Step
  + **Diamond**: Decision (Yes/No)
  + **Parallelogram**: Input/Output
* **Example**:
  + Sorting algorithm flowchart:



* **Advantages**:
  + Easy to visualize complex logic.
  + Useful for debugging.
* **Disadvantages**:
  + Time-consuming for large algorithms.
  + Hard to modify.

**4. Programming Language**

* **Description**:
  + Directly writing code in a language like C, Java, or Python.
  + Most precise but language-dependent.
* **Example**:



* **Advantages**:
  + Executable and testable.
  + No ambiguity.
* **Disadvantages**:
  + Requires syntax knowledge.
  + Less abstract (details may obscure logic).

**Decisions Prior to Designing an Algorithm**

To design an efficient algorithm, the following factors must be analyzed first:

**1. Based on the Capabilities of the Device**

* **Definition**:  
  Consider hardware limitations (e.g., memory, processing speed, I/O devices) to ensure the algorithm can run effectively on the target device.
* **Key Considerations**:
  1. **Memory Constraints**:
     + Avoid algorithms with high space complexity (e.g., recursion-heavy algorithms) for devices with limited RAM.
     + Example: Use iterative algorithms instead of recursive ones for embedded systems.
  2. **Processing Power**:
     + Avoid complex computations (e.g., matrix operations) on low-power devices (e.g., IoT sensors).
  3. **Real-Time Systems**:
     + Prioritize deterministic time algorithms (e.g., Round Robin scheduling) over non-deterministic ones.
  4. **Parallel Processing**:
     + Use multi-threaded or distributed algorithms for devices with multiple cores (e.g., servers).
* **Examples**:
  1. Mobile devices: Use lightweight algorithms (e.g., Quick Sort over Merge Sort for memory efficiency).
  2. Supercomputers: Leverage parallel algorithms (e.g., MapReduce).

**2. Based on the Nature of Solutions**

* **Definition**:  
  Choose an approach based on whether the problem requires an **exact**, **approximate**, **heuristic**, or **optimal solution**.
* **Key Considerations**:
  1. **Problem Constraints**:
     + **Exact Solutions**: Required for critical problems (e.g., encryption, financial calculations).
     + **Approximate/Heuristic Solutions**: Acceptable for NP-hard problems (e.g., Traveling Salesman Problem).
  2. **Optimality**:
     + Use **Greedy Algorithms** for locally optimal choices (e.g., Dijkstra’s Algorithm).
     + Use **Dynamic Programming** for globally optimal solutions (e.g., Knapsack Problem).
  3. **Problem Type**:
     + Numerical vs. Non-numerical problems (e.g., matrix multiplication vs. graph traversal).
* **Examples**:
  1. **Shortest Path**: Dijkstra’s Algorithm (exact) vs. Genetic Algorithms (heuristic).
  2. **Sorting**: Merge Sort (stable) vs. Quick Sort (unstable).

**3. Based on the Most Suitable Data Structures**

* **Definition**:  
  Select data structures that align with the algorithm’s operations (e.g., insertion, deletion, search) to optimize efficiency.
* **Key Considerations**:
  1. **Access Patterns**:
     + **Arrays**: Fast random access (e.g., Binary Search).
     + **Linked Lists**: Efficient insertions/deletions (e.g., real-time data streams).
  2. **Search Efficiency**:
     + **Hash Tables**: O(1) average-case search (e.g., database indexing).
     + **Trees**: O(log n) search (e.g., hierarchical data like file systems).
  3. **Space vs. Time Trade-off**:
     + **Stacks/Queues**: Low memory but limited operations (e.g., backtracking).
     + **Graphs**: High space complexity but versatile for modeling relationships.
* **Examples**:
  1. **Priority Queues**: Use heaps for efficient extraction of min/max (e.g., scheduling tasks).
  2. **Symbol Tables**: Use hash tables for compilers.

**Model of Computation**

Two key theoretical models for analyzing algorithms:

**1. RAM Model (Random Access Machine)**

* **Definition**:  
  A **sequential** computational model where a single processor executes instructions one at a time.
* **Key Features**:
  1. **Memory**: A large, contiguous array of cells (each storing data).
  2. **Operations**: Basic instructions (arithmetic, data movement, control) take **constant time**.
     + Example: ADD, MOVE, COMPARE.
  3. **Assumptions**:
     + Each operation (e.g., addition, memory access) takes **1 time unit**.
     + Memory access is **immediate** (no delay).
* **Advantages**:
  1. Simplifies algorithm analysis by ignoring hardware specifics (e.g., cache, disk).
  2. Focuses on **time complexity** (steps) and **space complexity** (memory).
* **Limitations**:
  1. Ignores real-world factors (e.g., pipelining, multi-core).
  2. Assumes unlimited memory (unrealistic).
* **Use Case**:
  1. Basis for analyzing **sequential algorithms** (e.g., Bubble Sort, Binary Search).

**2. PRAM Model (Parallel Random Access Machine)**

* **Definition**:  
  A **parallel** computational model with multiple processors working simultaneously on shared memory.
* **Key Features**:
  1. **Processors**: Multiple processors (P₁, P₂, ..., Pₙ) execute instructions in parallel.
  2. **Shared Memory**: All processors access a common memory space.
  3. **Conflict Handling**:
     + **EREW** (Exclusive Read, Exclusive Write): No concurrent read/write.
     + **CREW** (Concurrent Read, Exclusive Write): Concurrent reads allowed.
     + **CRCW** (Concurrent Read, Concurrent Write): Resolve write conflicts via rules (e.g., priority).
  4. **Assumptions**:
     + Uniform memory access time for all processors.
     + Synchronized execution (no clock skew).
* **Advantages**:
  1. Models **parallel algorithms** (e.g., matrix multiplication, parallel sorting).
  2. Measures speedup (e.g., time reduced by adding processors).
* **Limitations**:
  1. Unrealistic assumptions (e.g., no memory contention, synchronization costs).
  2. Does not account for communication delays.
* **Use Case**:
  1. Theoretical analysis of **parallel algorithms** (e.g., Parallel Merge Sort).

**RAM vs. PRAM**

|  |  |  |
| --- | --- | --- |
| **Aspect** | **RAM Model** | **PRAM Model** |
| **Processing** | Single processor | Multiple processors |
| **Memory Access** | Sequential | Concurrent (shared memory) |
| **Complexity** | Time/Space complexity | Time/Space + **Parallelism metrics** |
| **Realism** | Simplifies hardware | Idealized parallelism |
| **Example** | Sequential search (O(n)) | Parallel search (O(n/p) with p processors) |