**What is an Algorithm?**

**Informal Definition:**

* An **algorithm** is a step-by-step process or procedure to solve a problem.
* It takes inputs (data or values) and produces outputs (results) after performing specific steps.

**Formal Definition:**

* An **algorithm** is a finite set of clear instructions designed to complete a task.

**Characteristics of a Good Algorithm:**

1. **Input**:
   * It should take **zero or more inputs** from the user or external sources.
2. **Output**:
   * It should produce **at least one output** as a result.
3. **Definiteness**:
   * Each step in the algorithm must be **clear** and **unambiguous**, so there's no confusion.
4. **Finiteness**:
   * The algorithm must **end** after a limited number of steps. It should not run forever.
5. **Effectiveness**:
   * Every step should be **simple** enough to execute manually, like solving with just a pencil and paper.

**Important Aspects of Algorithm Study:**

1. **Designing an Algorithm**:
   * The process of creating a new algorithm.
2. **Expressing an Algorithm**:
   * Writing the steps of the algorithm clearly and in detail.
3. **Analyzing an Algorithm**:
   * Determining its **time complexity** (execution time) and **space complexity** (memory usage).
4. **Validating an Algorithm**:
   * Ensuring that it works correctly and solves the problem for all inputs.
5. **Testing an Algorithm**:
   * Running it on various inputs to check for errors and verify its correctness.

**1. How to devise algorithms:**

* **Creating algorithms is like art:** It’s not fully automatic; it requires creativity and knowledge of proven techniques.
* **Why study design strategies:** By learning different methods (like dynamic programming), you can create new, efficient algorithms.
* **Uses beyond computer science:** These methods are also valuable in fields like operations research and electrical engineering.

**2. How to validate algorithms:**

* **Proving correctness:** After creating an algorithm, you must ensure it works correctly for every possible input.
* **Steps in validation:**
  1. **Algorithm validation:** Prove the algorithm is logically correct, even before coding it.
  2. **Program verification:** Once the algorithm is written as code, ensure the program behaves as expected.
* **How to prove correctness:**
  1. Write clear rules about what the algorithm takes as input and what it outputs (this is called a specification).
  2. Use logic (predicate calculus) to check that the algorithm satisfies these rules.

**3. How to analyze algorithms:**

* **Why analyze algorithms:** To measure how efficiently they use a computer's resources like time and memory.
* **Key metrics:**
  + **Time complexity:** How much time the algorithm takes to run, depending on the size of the input.
  + **Space complexity:** How much memory the algorithm needs during execution.

**4. How to test a program:**

Testing ensures that the program behaves as expected. It has two parts:

1. **Debugging:**
   * Run the program with sample data to check for errors.
   * Fix any errors that occur. However, as Dijkstra said, debugging only proves errors exist; it doesn’t guarantee there are none.
2. **Profiling:**
   * Once the program is error-free, measure its performance.
   * This involves checking how much time and memory the program uses on different inputs.

Asymptotic notation is a way to describe how the time or space required by an algorithm grows as the size of the input increases. It focuses on the *big picture* of performance, ignoring small details like specific hardware or constant factors.

Here are the common notations explained simply:

1. **Big-O Notation (O):**
   * Describes the *worst-case* performance of an algorithm.
   * It shows the upper limit of how an algorithm's runtime grows with input size.
   * Example: If an algorithm is O(n2) its runtime grows no faster than n^2 as input size increases.
2. **Omega Notation (Ω):**
   * Describes the *best-case* performance of an algorithm.
   * It shows the lower limit of runtime growth.
   * Example: If an algorithm is Ω(n) its runtime is at least proportional to n in the best case.
3. **Theta Notation (Θ):**
   * Describes the *exact* growth rate of an algorithm.
   * It’s used when the runtime grows at the same rate in both the best and worst cases.
   * Example: If an algorithm is Θ(n) its runtime is proportional to n in all cases.

**Space Complexity**

This measures how much memory an algorithm needs to run. The memory usage is broken into two parts:

1. **Fixed Part:**
   * Memory needed for things that don’t change, regardless of the input size.
   * Example: Code (instructions), constants, and fixed-size variables like simple integers.
2. **Variable Part:**
   * Memory needed for things that depend on the input size.
   * Example: Arrays, dynamic data structures, and recursion stack space.

The total space required by an algorithm can be written as:  
**S(P) = c + Sp(instance characteristics)**  
Where:

* ccc = constant (fixed memory).
* SpSpSp = variable part (depends on the input).

A screenshot of a computer

Description automatically generatedA screenshot of a computer program

Description automatically generated

**Time Complexity**

This measures how much time an algorithm takes to complete, expressed as the number of steps.

1. **Compile Time:**
   * Time taken to compile the program.
   * It doesn’t depend on the input size, so we often ignore this.
2. **Run Time:**
   * Time taken to execute the program.
   * It depends on the input size and is analyzed in steps.

**Key Ideas:**

* Each type of statement contributes differently to the step count:
  + **Comments**: 000 steps.
  + **Simple statements (e.g., assignments)**: 111 step.
  + **Loops (e.g., for, while)**: Depends on how many iterations they execute.

**Methods to Count Steps:**

1. Add a **count** variable in the program to track the steps.
2. Increment **count** for each operation executed.

**Examples:**

A screenshot of a computer program

Description automatically generated

**Amortized Analysis (Simplified Explanation)**

Amortized analysis helps us find the **average time per operation** for an algorithm, even in the **worst-case scenario** of repeated operations. It’s useful for understanding the overall performance of algorithms that have occasional expensive operations but are efficient overall.

**Key Idea:**

Instead of looking at the cost of each operation individually, we spread (or "amortize") the high cost of expensive operations over many cheap ones. This gives a balanced or average cost for each operation.

**Example Scenario:**

Imagine we have a sequence of operations:  
Insert (I) and Delete (D):

* Inserts cost **1** step each.
* Deletes may have **higher costs**, e.g., **8** and **10** in some cases.

If we perform:  
I1, I2, D1, I3, I4, I5, I6, D2, I7, the **total cost** = 25 (adding individual costs).  
**Amortized cost** = Total cost ÷ Number of operations = **25 ÷ 9 = ~2.78 per operation**.  
This "averages out" the expensive operations across the whole sequence.

A screenshot of a black and white math problem

Description automatically generated

A screenshot of a black and white page

Description automatically generated

A screenshot of a black and white math problem

Description automatically generated

A screenshot of a computer

Description automatically generated

**Towers of Hanoi Algorithm in Simple Steps**

The **Towers of Hanoi** is a classic recursive problem. The goal is to move a stack of disks from one peg to another, following these rules:

1. Only one disk can be moved at a time.
2. A disk can only be placed on top of a larger disk or on an empty peg.
3. You must move all disks from the source peg to the target peg, using an auxiliary peg.

**Algorithm**

Let:

* n be the number of disks.
* source be the starting peg.
* target be the destination peg.
* auxiliary be the intermediate peg.

**Steps:**

1. Move the top n-1 disks from source to auxiliary, using target as a helper.
2. Move the largest disk (the nth disk) directly from source to target.
3. Move the n-1 disks from auxiliary to target, using source as a helper.

A screenshot of a computer program

Description automatically generated

**Example for 3 Disks**

Let A = source, B = auxiliary, and C = target.

1. Move 2 disks from A to B (using C as helper).
   * Move 1 disk from A to C.
   * Move 1 disk from A to B.
   * Move 1 disk from C to B.
2. Move the largest disk from A to C.
3. Move 2 disks from B to C (using A as helper).
   * Move 1 disk from B to A.
   * Move 1 disk from B to C.
   * Move 1 disk from A to C.

**Methods for Amortized Analysis:**

1. **Aggregate Method**:
   * Add up the **total cost of all operations**, then divide by the number of operations.
   * Formula: **Amortized cost = (Total cost of n operations) ÷ n**.
2. **Accounting Method**:
   * Assign a **fixed cost** (amortized cost) to each operation.
   * Overcharge cheaper operations to "save" extra cost for future expensive ones.
   * Example: Charging **$75 for each operation** even if actual costs vary.
3. **Potential Method**:
   * Define a **potential function** to represent the "extra cost saved" after each operation.
   * Use this function to calculate the amortized cost.
   * Formula: **Amortized Cost = Actual Cost + Change in Potential Function**.

**Example Using Monthly Costs:**

We pay:

* **$50 for regular months**.
* **$100 for March, June, September, December**.

1. **Aggregate Method**:  
   Calculate the average cost:
   * Total cost for 12 months ≤ **$75 per month**.
2. **Accounting Method**:  
   Assign **$75 for every month**. The extra cost in regular months covers expensive months.
3. **Potential Method**:  
   Use a potential function to track the saved cost across months:
   * Potential increases during $50 months and decreases during $100 months.

**Key Points to Remember:**

1. **Amortized cost** is about balancing the cost over time.
2. It does not ignore expensive operations; it spreads their impact over many cheaper ones.
3. Methods like **aggregate**, **accounting**, and **potential** provide different ways to analyze amortized costs.

**Probabilistic Analysis (Quick Overview):**

* In **probabilistic analysis**, we use **probability** to analyze the performance of an algorithm based on the likelihood of different inputs.
* Example: A company ranks **n candidates** for hiring and evaluates them based on probabilities of getting the best outcome.

In simpler terms, it predicts how efficient the algorithm will be for **random inputs** instead of worst-case scenarios.