# Fundamentals of Algorithms

## 1.1 What is an Algorithm?

An algorithm is a finite sequence of well-defined, unambiguous instructions that take a given input, produce an output, and terminate. Algorithms are specifically meant to solve computational problems.

There are several ways to define an algorithm: - **Definition 1:** A step-by-step instruction (procedure) to solve a problem. - **Definition 2:** An algorithm can be viewed as a mathematical function that maps an input to a desired output. In mathematics, a function from a set X to a set Y assigns exactly one element of Y to each element of X (y = f(x)). - **Definition 3:** An algorithm is a general description of a solution, strategy, or process that can be used to communicate your approach to others.

## 1.2 Characteristics of an Algorithm

A proper algorithm should have the following characteristics: - **Correctness:** It produces the correct output for every valid input. - **Deterministic:** It should be consistent across time; given the same input, it always produces the same output. - **Finite:** The algorithm must terminate after a finite number of steps. - **Efficient:** It should solve the problem using a reasonable amount of resources. - **Simple & Communicable:** It should be easily understandable.

## 1.3 Factors Determining Software/Program Performance

The performance of a program is determined by several factors: - **Major Factors:** - i) Algorithm and Data Structures (the solution). - **Platform-dependent Factors:** - ii) Hardware - iii) Programming Language - iv) Compiler - v) Operating System (OS)

The study of **Design and Analysis of Algorithms (DAA)** helps you classify and build an algorithm, allowing you to figure out the right design strategy for any given computational problem. It also helps you analyze an algorithm to determine how efficient it is in terms of time, space, power consumption, and cache locality.

---

# 2. Algorithm Analysis

## 2.1 Measuring Algorithm Speed

The central question is: **"How fast is my algorithm as a function of input size?"**

Finding the exact time an algorithm takes before running it is difficult.

- **Observation:** Performance patterns seem to emerge only for large inputs. For smaller inputs, the running time is almost indistinguishable.

Example: - For an O(n) algorithm where $n=10^4$ and an $O(n^2)$ algorithm where $n=10^4$, the difference is significant (e.g., 1 sec vs 1 nanosecond). - For $n=10^5$, an $O(n^2)$ algorithm might take $(10^5)^2 / 10^9 = 10$ seconds on a 1 GHz processor ($10^9$ cycles/sec, assuming an operation takes 10 cycles on average). - The difference in time will be difficult to notice for small inputs.

## 2.2 Step-Count Analysis

Instead of measuring the exact time, we can count the number of steps/operations an algorithm takes for a given input size. This is called **Step-Count Analysis**.

**Steps to find T(n) (the time function):** 1. Find all the 'operations' in our algorithm. 2. Calculate the time taken for each operation. Let the algorithm have K operations {0, 1, ..., K-1}, with times $t_i$. 3. Find $C_i(n)$: the execution count of operation i (how many times operation i is executed). 4. Total time: $T(n) = C_0(n)t_0 + C_1(n)t_1 + ... + C_{k-1}(n)t_{k-1}$.

**Limitations of Step-Count Analysis:** - i) We assume all operations take the same time, which is false (compare vs load/store cycles differ). - ii) We don't model memory and caching delays. - iii) We ignore lower-order terms (e.g., in $O(n^2)$, we ignore $O(n^{2-1})$).

## 2.3 Sensitivity Analysis

A key question: **"How sensitive is the running time to the input size?"**

This asks how much T(n) changes if we double the input size.

- Sensitivity measured by ratio: $\lim n\to\infty T(2n) / T(n)$.
- Independent of programming language, machine, etc.
- Helps group algorithms into complexity classes.

**Examples:** - Line of n dots: C_line(n) = n. Ratio → 2. - Square grid (n x n): $C_{grid}(n) = n^2$. Ratio → 4. - Triangle: C_tri(n) = n(n+1)/2. Ratio → 4.

Conclusion: For large n, a triangle behaves asymptotically like a square.

## 2.4 Asymptotic Analysis and Notation

Asymptotic analysis groups/clusters algorithms based on their sensitivity. We generally care about performance for large inputs.

Main idea: Measure efficiency without depending on machine-specific constants or implementation.

- **Step count is asymptotic:** Only count primary operations.

### Big-Oh (O) Notation – Asymptotic Upper Bound

- $f(n) \in O(g(n))$ if $\exists\ c > 0, N > 0$ s.t. $f(n) \leq c * g(n)$ for all $n \geq N$.

- Example: $9.7n^2 + 5n + 45 = 9.7n^2 + O(n)$.

### Big-Omega (Ω) Notation – Asymptotic Lower Bound

- $f(n) \in \Omega(g(n))$ if $\exists\ c > 0, N_0 > 0$ s.t. $f(n) \geq c * g(n)$ for all $n \geq N_0$.

### Big-Theta (Θ) Notation – Asymptotic Tight Bound

- $f(n) \in \Theta(g(n))$ if $\exists\ c_1, c_2 > 0, N_0 > 0$ s.t. $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq N_0$.
- Big-Theta = intersection of Big-Oh and Big-Omega.

### Little-oh (o) and Little-omega (ω)

- **o(g(n))**: Strictly smaller order.
- **ω(g(n))**: Strictly larger order.

### Limit Test

- $\lim f(n)/g(n) = 0 \rightarrow f(n) \in o(g(n))$.
- $\lim f(n)/g(n) = \text{constant} > 0 \rightarrow f(n) \in \Theta(g(n))$.
- $\lim f(n)/g(n) = \infty \rightarrow f(n) \in \omega(g(n))$.

---

# 3. Recurrence Relations

A recurrence relation expresses the $n^{th}$ term with reference to previous term(s). Goal: reduce to a **closed form** (non-recursive solution).

### Examples

- **Tower of Hanoi:** $T(n) = 2T(n-1) + 1 \rightarrow T(n) = 2^n - 1 = \Theta(2^n)$.
- **Binary Search:** $T(n) = T(n/2) + c \rightarrow O(\log n)$.
- **Merge Sort:** $T(n) = 2T(n/2) + n \rightarrow \Theta(n \log n)$.

## 3.1 Divide and Conquer Recurrences

Standard form: T(n) = aT(n/b) + f(n). - **a:** branching factor. - **b:** reduction factor. - **f(n):** overhead cost.

Number of leaves = a^(log_b n) = n^(log_b a). - If a > b → log_b a > 1 → recursion wider. - If a < b → log_b a < 1 → fewer leaves. - If a = b → log_b a = 1 → exactly n leaves.

Asymptotically, either subproblems or overhead dominate.

### 3.2 Master Theorem

For $T(n) = aT(n/b) + f(n)$:

1. **Case 1 (Overhead Dominates):** $f(n) = \Omega(n^{(\log_b a + \varepsilon)})$, with regularity condition $a f(n/b) \leq c f(n)$. Then $T(n) = \Theta(f(n))$.
2. **Case 2 (Balanced):** $f(n) = \Theta(n^{(\log_b a)})$. Then $T(n) = \Theta(n^{(\log_b a)} \log n)$.
3. **Case 3 (Leaves Dominate):** $f(n) = O(n^{(\log_b a - \varepsilon)})$. Then $T(n) = \Theta(n^{(\log_b a)})$.

---

# 4. Loop Invariants and Correctness

A loop invariant is a condition that remains true before and after each iteration of a loop.

To prove correctness: 1. **Initialization:** Invariant true before first iteration. 2. **Maintenance:** If true before iteration, stays true before next. 3. **Termination:** When loop ends, invariant gives useful property proving correctness.

**Example: Insertion Sort** - Loop Invariant: Subarray $A[1,\ldots,i-1]$ remains sorted.

---

# 5. Notes on Specific Algorithms

### Heapsort

- Time: $O(n \log n)$.
- Max-Heapify: $O(\log n)$.
- Build-Max-Heap: $O(n)$.

### Merge Sort

- Recurrence: $T(n) = 2T(n/2) + O(n)$.
- Solution: $O(n \log n)$.

### Quick Sort

- Partition: $O(n)$.
- Worst Case: $T(n) = T(n-1) + O(n) = O(n^2)$.

---

# 6. Greedy Method

## 6.1 Formal Setup

Optimization problem = (I, S, f, goal) - I: set of instances. - $S(I_i)$: set of feasible solutions. - f: objective function. - goal: maximize or minimize.

## 6.2 Greedy Algorithm Strategy

- Construct solution incrementally.
- Make locally optimal choices.
- Choices are irrevocable.
- Greedy fails when future consequences matter.

## 6.3 Correctness Conditions

1. **Greedy-Choice Property:** Optimal solution can be built from greedy choices.
2. **Optimal Substructure:** Optimal solution can be built from optimal subproblem solutions.

## 6.4 Example Problems

### 1. Coin Change Problem

- Strategy: Take largest coin.
- Works for canonical {quarter, dime, nickel, penny}.
- Fails for non-canonical {1, 3, 4}.

### 2. Container Loading Problem

- Strategy: Select least weight container.
- Proof: Induction on positions.

### 3. Fractional Knapsack Problem

- Strategy: Sort by profit/weight ratio, take fractions.
- Correct and optimal.

### 4. Job Scheduling with Deadlines

- Strategy: Sort jobs by decreasing profit, schedule feasible ones.
- Correct and optimal.