Algorithms Design & Analysis

Unit 1: Introduction & Divide and Conquer

Introduction to Algorithms

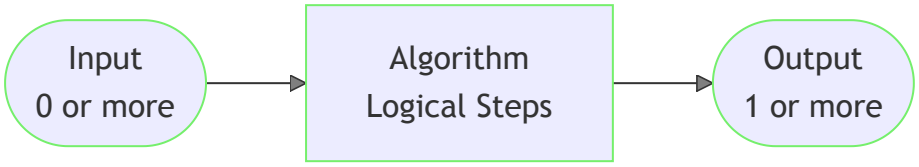
What is an Algorithm?

The term **"algorithm"** is historically derived from the name of the famous 9th-century Persian mathematician and astronomer, **Abu Ja'far Muhammad ibn Musa al-Khwarizmi** (whose Latinized name became *Algoritmi*).

In computer science, an **algorithm** is defined as:

> *A finite set of unambiguous, step-by-step instructions that, when executed in a specific order with a given set of inputs, solves a computational problem and produces a desired output in a finite amount of time.*

An algorithm acts as a blueprint or logical recipe for solving a problem, independent of any programming language or hardware platform.



Difference Between an Algorithm and a Program

While these terms are often used interchangeably in casual conversation, they have distinct technical differences:

Attribute	Algorithm	Program
Definition	A theoretical, language-independent design or logic to solve a problem.	A concrete implementation of an algorithm in a specific programming language.
Termination	Must **always** terminate after a finite number of steps.	May or may not terminate (e.g., an Operating System or web server runs in an infinite loop).
Execution	Cannot be directly executed by a computer; must be converted to code.	Directly executable by the computer CPU after compilation/interpretation.
Medium	Written in natural language, pseudocode, or flowcharts.	Written in formal code (C, C++, Java, Python, etc.).

A process that satisfies all characteristics of an algorithm *except* termination is called a **computational procedure** (e.g., operating systems, database management engines).

Core Characteristics / Criteria of an Algorithm

For a set of instructions to be classified as an algorithm, it must satisfy the following five fundamental criteria established by Donald Knuth:

1. **Input:**
   - It must have **zero or more** externally supplied values. Some algorithms do not require external inputs and generate values internally (e.g., generating first n Fibonacci numbers).
2. **Output:**
   - It must produce **at least one** output value. The output is the solution to the given problem.
3. **Definiteness (Unambiguity):**
   - Every step of the algorithm must be clear, precise, and unambiguous. There should be no doubt about what action is to be taken.
   - *Non-example:* Instructions like `"compute 5 / 0"` or `"add 6 or 7 to x"` are invalid because their results or actions are undefined or ambiguous.
4. **Finiteness (Termination):**
   - If we trace the steps of the algorithm, it must terminate after a **finite number of operations** for all possible input values.
5. **Effectiveness (Feasibility):**
   - Every instruction must be basic enough that it can, in principle, be executed by a human using only pencil and paper in a finite amount of time.
   - *Non-example:* Arithmetic using arbitrary real numbers is not always effective because some real numbers (like π or $1/3$) have infinite decimal expansions, making exact arithmetic impossible in finite time.

Performance Analysis

Overview

Performance analysis is the process of evaluating the resources (computation time and memory space) that an algorithm requires to execute. It allows us to compare different algorithms designed to solve the same problem and choose the most efficient one.

Performance evaluation is split into two distinct phases:

1. **A Priori Analysis (Theoretical Evaluation):**
   - An analysis performed **before** implementing the algorithm in code.
   - It determines the growth rate of the algorithm's running time and space requirements as the input size n increases.
   - It is independent of machine specifications, compiler version, and programming language.
   - Focuses on the **frequency count** of operations.
2. **A Posteriori Testing (Profiling / Empirical Evaluation):**
   - A practical evaluation performed **after** the algorithm is written, compiled, and executed.
   - It involves running the program on sample datasets of varying sizes and measuring actual execution time (in milliseconds/seconds) using system clocks and tracking memory usage in bytes.
   - Dependent on hardware, compiler, operating system, and system load.

Space Complexity

Definition

> **Space Complexity** of an algorithm is the total amount of memory space (RAM/storage) required by the algorithm to run to completion as a function of the input size n.

The total space requirements of any program P, denoted as $S(P)$, can be expressed as:

$$S(P) = C + S_P(I)$$

Where:

- C **(Fixed Space Component):**
  - The memory required that is independent of the input size.
  - This includes space for the machine code instructions, constant values, simple variables, and fixed-size structures.
- $S_P(I)$ **(Variable Space Component):**
  - The memory required that depends dynamically on the input size I (denoted by n).
  - This includes space for dynamic data structures (like arrays, linked lists, trees), recursion stack frames, and local variables inside active recursive calls.

Space Complexity Examples

Example 1: Iterative Sum of Elements

Consider an algorithm that sums the elements of an array:

```
Algorithm Sum(A, n)
// Input: An array A of size n
// Output: Sum of all elements in the array
{
    total := 0.0;
    for i := 1 to n do
        total := total + A[i];
    return total;
}
```

- **Fixed space:**
  - Variables `n` (size), `total` (accumulator), and `i` (loop counter) require constant space. Let this space be c_1.
  - Code instructions require constant space c_2.
- **Variable space:**
  - The array parameter `A` is passed by reference (usually a pointer of constant size). Thus, no extra local copies of the array are created.
  - The loop does not allocate dynamic memory.
- **Analysis:**
  - The space requirement is independent of the input size n.
  - Total space $S(\text{Sum}) = O(1)$ (Constant Space).

Example 2: Recursive Sum of Elements

Now, consider the recursive version of the same algorithm:

```
Algorithm RecSum(A, n)
// Input: An array A of size n
// Output: Sum of all elements in the array computed recursively
{
    if (n <= 0) then
        return 0.0;
    else
        return RecSum(A, n - 1) + A[n];
}
```

- **Analysis:**
  - In a recursive algorithm, each call pushes an activation record (stack frame) onto the call stack.

- A stack frame contains parameters (`A` pointer, `n`) and the return address.
- For input size n, the recursion goes to a depth of $n+1$ frames: $\text{RecSum}(A, n) \to \text{RecSum}(A, n-1) \to \cdots \to \text{RecSum}(A, 0)$.
- Each stack frame requires a fixed amount of space (say, k bytes).
- Total space required by the stack = $k \cdot (n+1)$ bytes.
- Therefore, the variable space component grows linearly with n.
- Total Space Complexity $S(\text{RecSum}) = \Theta(n)$ (Linear Space).

Time Complexity

Definition

> **Time Complexity** of an algorithm is the total time taken by the algorithm to execute and run to completion as a function of the input size n.

Because the absolute execution time depends on hardware, compiler, and OS, we analyze time complexity theoretically by calculating the **Step Count** or **Frequency Count** of basic operations.

A **program step** is defined as a segment of code that is executed conceptually as a single unit, independent of the exact number of assembly instructions it translates into.

Time Complexity Calculations (Step Count Method)

Example 1: Iterative Sum of Elements

Let's analyze the step count for the iterative array sum:

Line No.	Statement	Step Cost	Execution Frequency (Count)	Total Steps
1	`Algorithm Sum(A, n)`	0	0	0
2	`{`	0	0	0
3	`total := 0.0;`	1	1	1
4	`for i := 1 to n do`	1	$n+1$	$n+1$
5	`total := total + A[i];`	1	n	n
6	`return total;`	1	1	1
7	`}`	0	0	0
Total				$2n+3$

- **Explanation of loop bounds:** The loop header `for i := 1 to n` executes $n + 1$ times because it evaluates i from 1 up to n (which executes the loop body n times) plus a final check where $i = n + 1 > n$, which fails and exits the loop.
- **Resulting Function:** $T(n) = 2n + 3$.
- As $n \to \infty$, the dominant term is $2n$, meaning the time complexity is linear, denoted as $O(n)$.

Example 2: Matrix Addition

Let's analyze two $m \times n$ matrices being added together:

```
Algorithm Add(A, B, C, m, n)
{
    for i := 1 to m do                      // m + 1 times
        for j := 1 to n do                  // m * (n + 1) times
            C[i, j] := A[i, j] + B[i, j];   // m * n times
}
```

Let's break down the execution count step-by-step:

1. Outer loop `for i := 1 to m`: runs $m + 1$ times.
2. Inner loop `for j := 1 to n`: runs $n + 1$ times for each iteration of the outer loop. Since the outer loop executes m times successfully, the inner loop header is evaluated $m \times (n + 1) = mn + m$ times.
3. Assignment statement `C[i, j] := A[i, j] + B[i, j]`: runs n times for each successful iteration of the outer loop. Total executions = $m \times n = mn$.

Total Step Count Calculation:

$$T(m, n) = (m + 1) + (mn + m) + mn$$

$$T(m, n) = 2mn + 2m + 1$$

If $m = n$ (square matrices of size $n \times n$):

$$T(n) = 2n^2 + 2n + 1$$

- The dominant term is $2n^2$. Thus, the Time Complexity is quadratic, denoted as $O(n^2)$.

Asymptotic Notations

Concept

When analyzing the running time of an algorithm, we want to know how the execution time scales as the input size n grows towards infinity ($n \to \infty$).

Asymptotic notations are mathematical tools used to:

1. Ignore constant factors (like compiler overhead, clock speed).
2. Ignore lower-order terms (which become insignificant as n grows very large).
3. Focus purely on the **order of growth** of the run time.

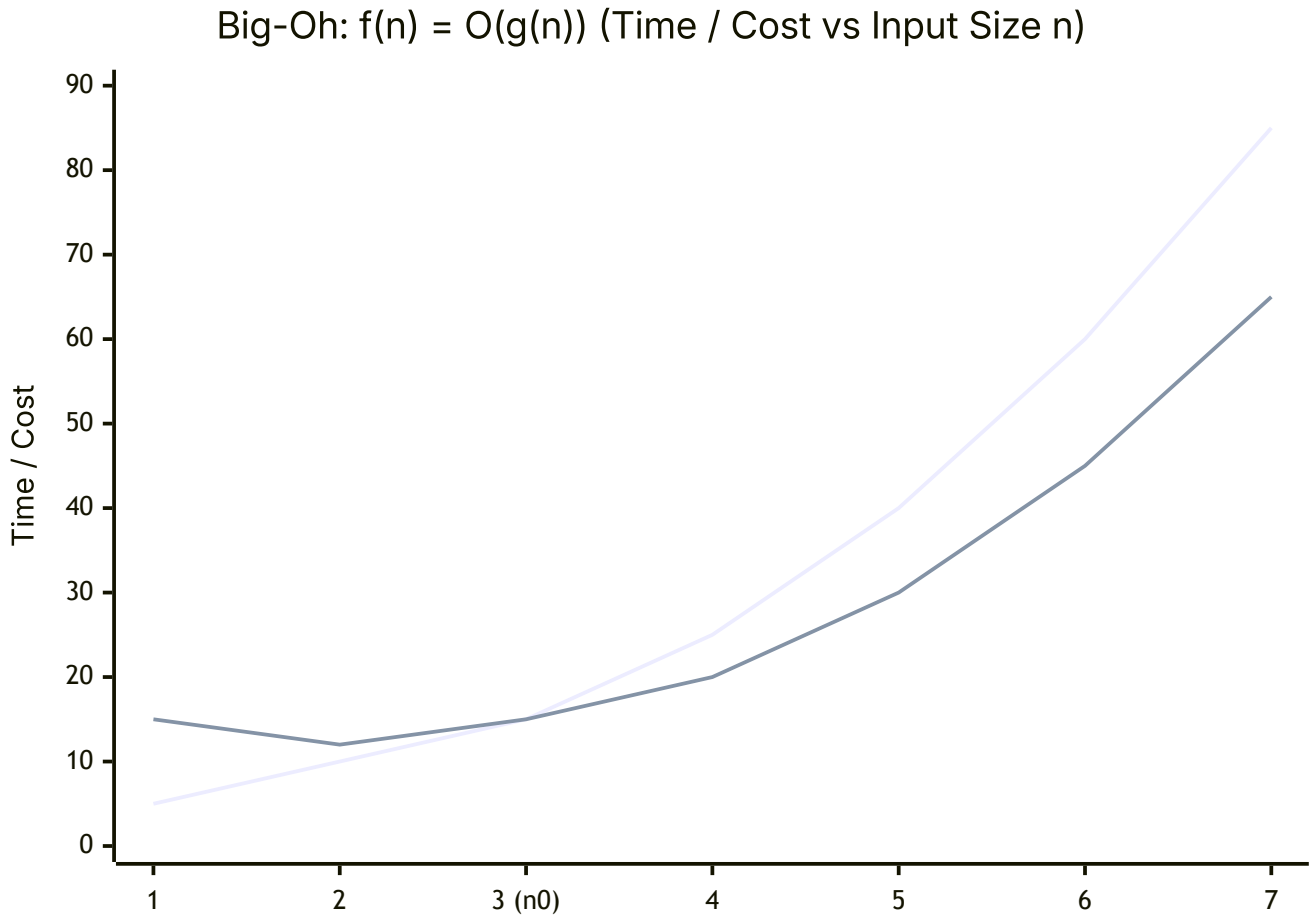
---

1. Big-Oh Notation (O) – Upper Bound

Formal Definition

We say $f(n) = O(g(n))$ (read as *"f of n is Big-Oh of g of n"*) if and only if there exist positive constants c and n_0 such that:

$$|f(n)| \leq c \cdot |g(n)| \quad \text{for all } n \geq n_0$$



> **Graph Legend:** Curve 1 (higher at $n \geq n_0$) is $c \cdot g(n)$ (Upper Bound) | Curve 2 is $f(n)$ (Actual Complexity). At $n \geq n_0$, $f(n) \leq c \cdot g(n)$ holds.

Explanation

Big-Oh notation provides an **asymptotic upper bound** for a function. It guarantees that the algorithm will never take more than $c \cdot g(n)$ time. It represents the **worst-case** performance.

Mathematical Proofs

Problem 1: Prove that $3n + 2 = O(n)$

- **Goal:** Find constants $c > 0$ and $n_0 > 0$ such that $3n + 2 \leq c \cdot n$ for all $n \geq n_0$.
- Let's analyze the equation:

$$3n + 2 \leq c \cdot n$$

$$\frac{3n+2}{n} \leq c \implies 3 + \frac{2}{n} \leq c$$

- If we choose $n_0 = 1$, then for all $n \geq 1$, the term $\frac{2}{n} \leq 2$.
- So, $3 + \frac{2}{n} \leq 3 + 2 = 5$.
- Therefore, we can choose $c = 5$ and $n_0 = 1$.
- **Verification:**
  For $n = 1$: $3(1) + 2 = 5 \leq 5(1) \implies 5 \leq 5$ (True).
  For $n = 2$: $3(2) + 2 = 8 \leq 5(2) \implies 8 \leq 10$ (True).
- Since $3n + 2 \leq 5n$ for all $n \geq 1$, we have successfully proven that $3n + 2 = O(n)$.

Problem 2: Prove that $10n^2 + 4n + 2 = O(n^2)$

- **Goal:** Find $c > 0, n_0 > 0$ such that $10n^2 + 4n + 2 \leq c \cdot n^2$ for all $n \geq n_0$.
- For $n \geq 1$, we know that:

$$4n \leq 4n^2$$

$$2 \leq 2n^2$$

- Substitute these bounds into our original expression:

$$10n^2 + 4n + 2 \leq 10n^2 + 4n^2 + 2n^2$$

$$10n^2 + 4n + 2 \leq 16n^2$$

- This inequality holds for all $n \geq 1$.
- Thus, we choose $c = 16$ and $n_0 = 1$.
- Since $10n^2 + 4n + 2 \leq 16n^2$ for all $n \geq 1$, the statement $10n^2 + 4n + 2 = O(n^2)$ is proven.

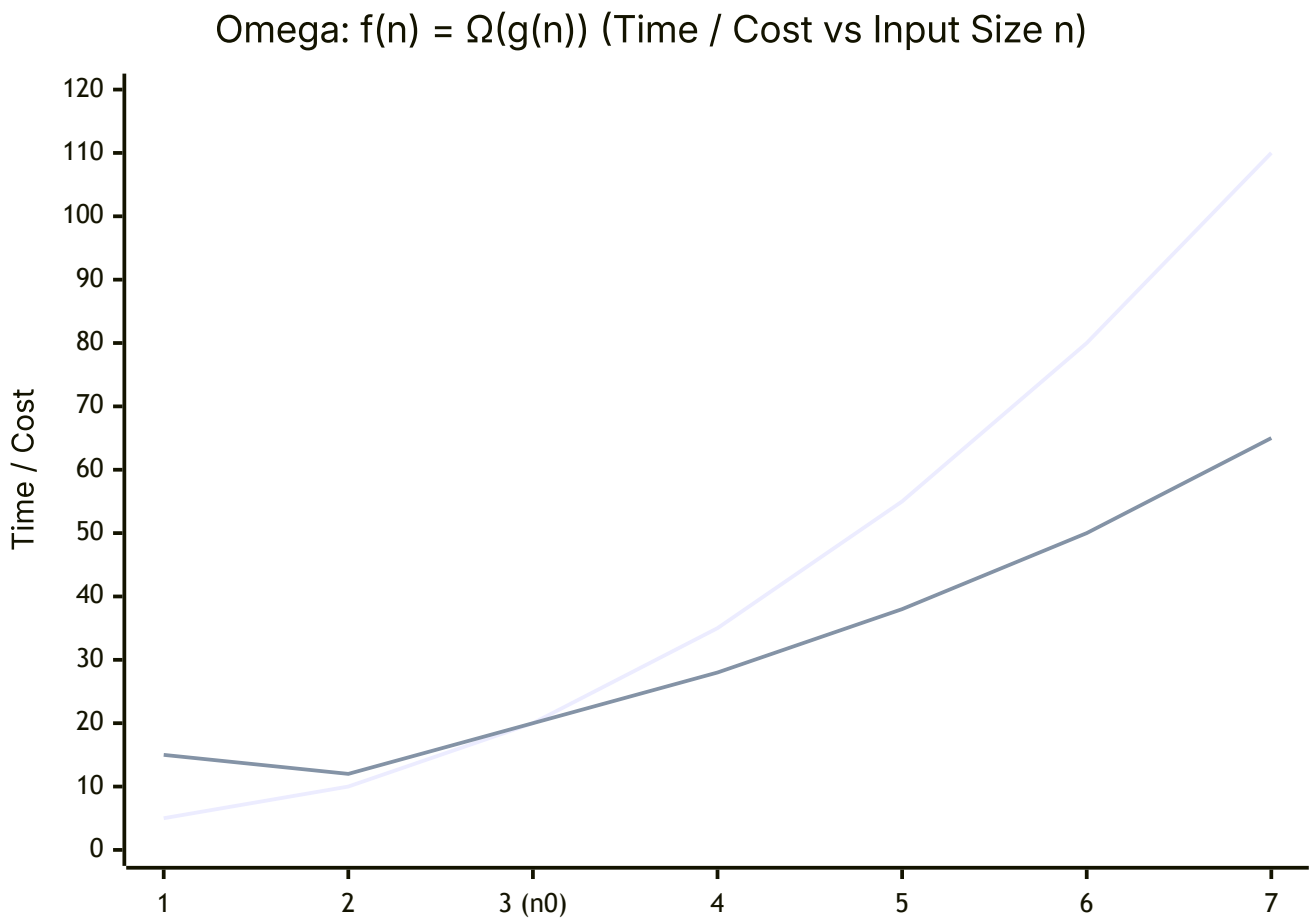
---

2. Omega Notation (Ω) – Lower Bound

Formal Definition

We say $f(n) = \Omega(g(n))$ (read as *"f of n is Omega of g of n"*) if and only if there exist positive constants c and n_0 such that:

$$|f(n)| \geq c \cdot |g(n)| \quad \text{for all } n \geq n_0$$



Graph Legend: Curve 1 (higher at $n \geq n_0$) is $f(n)$ (Actual Complexity) | Curve 2 is $c \cdot g(n)$ (Lower Bound). At $n \geq n_0$, $f(n) \geq c \cdot g(n)$ holds.

Explanation

Omega notation provides an **asymptotic lower bound** for a function. It guarantees that the algorithm will take at least $c \cdot g(n)$ time. It represents the **best-case** performance.

Mathematical Proofs

Problem: Prove that $3n + 2 = \Omega(n)$

- **Goal:** Find constants $c > 0$ and $n_0 > 0$ such that $3n + 2 \geq c \cdot n$ for all $n \geq n_0$.
- For $n \geq 1$:

$$3n + 2 \geq 3n$$

- Therefore, we can choose $c = 3$ and $n_0 = 1$.
- **Verification:**
  For $n = 1$: $3(1) + 2 = 5 \geq 3(1) \implies 5 \geq 3$ (True).
  For $n = 2$: $3(2) + 2 = 8 \geq 3(2) \implies 8 \geq 6$ (True).
- Since $3n + 2 \geq 3n$ for all $n \geq 1$, it is proven that $3n + 2 = \Omega(n)$.

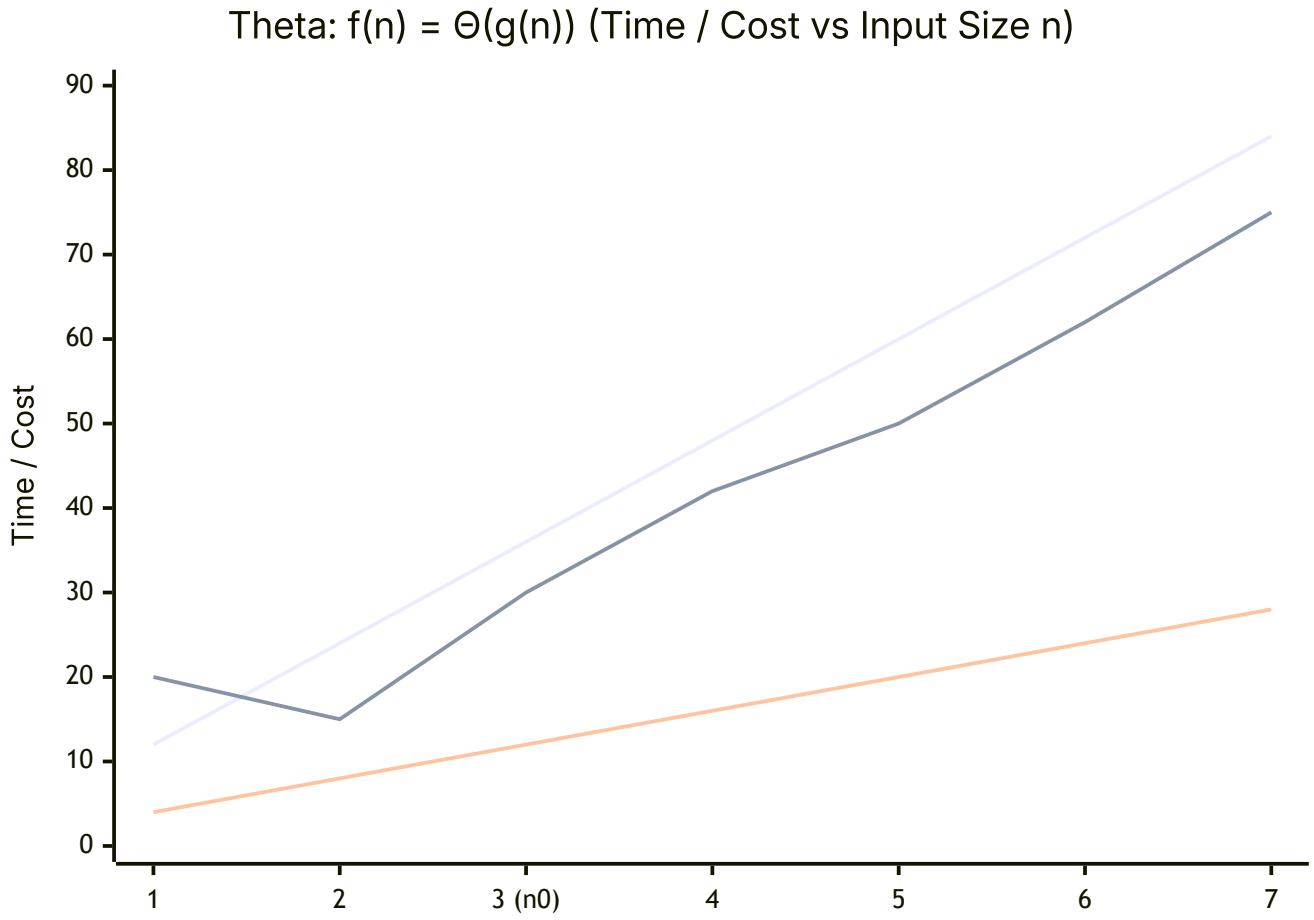
---

3. Theta Notation (Θ) – Tight Bound

Formal Definition

We say $f(n) = \Theta(g(n))$ (read as *"f of n is Theta of g of n"*) if and only if there exist positive constants c_1, c_2, and n_0 such that:

$$c_1 \cdot |g(n)| \leq |f(n)| \leq c_2 \cdot |g(n)| \quad \text{for all } n \geq n_0$$



> **Graph Legend:** Curve 1 (highest) is $c_2 \cdot g(n)$ (Upper Bound) | Curve 2 (middle) is $f(n)$ (Actual Complexity) | Curve 3 (lowest) is $c_1 \cdot g(n)$ (Lower Bound). At $n \geq n_0$, $c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$ holds.

Explanation

Theta notation provides a **tight bound** (exact rate of growth). It implies that the actual function $f(n)$ grows exactly like $g(n)$ up to constant factors.
An algorithm has $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$.

Mathematical Proofs

Problem: Prove that $3n + 2 = \Theta(n)$

- **Goal:** Find positive constants c_1, c_2, n_0 such that $c_1 \cdot n \leq 3n + 2 \leq c_2 \cdot n$ for all $n \geq n_0$.
- From our previous proofs:

1. We found $3n + 2 \leq 5n$ for all $n \geq 1$ (Upper Bound: $c_2 = 5, n_0 = 1$).
2. We found $3n + 2 \geq 3n$ for all $n \geq 1$ (Lower Bound: $c_1 = 3, n_0 = 1$).

- Combining these two inequalities:

$$3n \leq 3n + 2 \leq 5n \quad \text{for all } n \geq 1$$

- This satisfies the definition with $c_1 = 3$, $c_2 = 5$, and $n_0 = 1$.
- Therefore, $3n + 2 = \Theta(n)$.

---

4. Little-oh Notation (o) – Strict Upper Bound

Formal Definition

We say $f(n) = o(g(n))$ (read as *"f of n is little-oh of g of n"*) if and only if for **every** positive constant $c > 0$, there exists a positive constant $n_0 > 0$ such that:

$$|f(n)| < c \cdot |g(n)| \quad \text{for all } n \geq n_0$$

An equivalent and highly practical definition uses limits:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Explanation

While Big-Oh represents a loose upper bound (analogous to $\leq$), Little-oh represents a **strict upper bound** (analogous to $<$). It means $f(n)$ becomes completely insignificant compared to $g(n)$ as n grows.

Mathematical Proofs

Problem: Prove that $3n + 2 = o(n^2)$

- Let's calculate the limit of the ratio of the two functions as $n \to \infty$:

$$\lim_{n \to \infty} \frac{3n + 2}{n^2}$$

$$\lim_{n \to \infty} \left(\frac{3n}{n^2} + \frac{2}{n^2} \right) = \lim_{n \to \infty} \left(\frac{3}{n} + \frac{2}{n^2} \right)$$

- As $n \to \infty$, both terms approach zero:

$$0 + 0 = 0$$

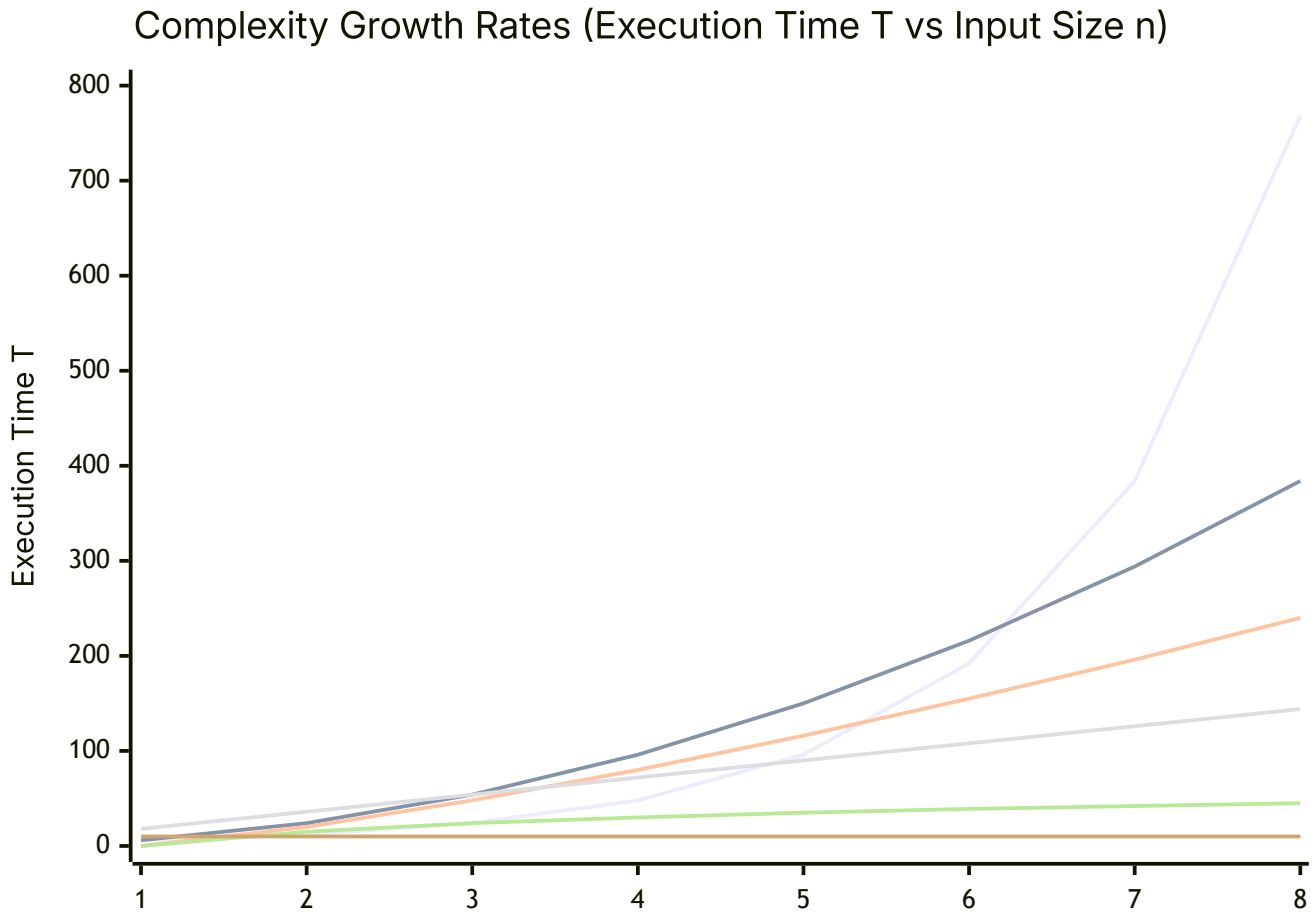
- Since the limit is exactly 0, it is proven that $3n + 2 = o(n^2)$.
- *Note:* $3n + 2 \neq o(n)$ because $\lim_{n \to \infty} \frac{3n+2}{n} = 3 \neq 0$.

Comparison of Growth Rates

Asymptotic complexity classes can be ordered by their rate of growth. A lower growth rate means a more efficient algorithm for large values of n.

$$\text{Constant} < \text{Logarithmic} < \text{Linear} < \text{Linearithmic} < \text{Quadratic} < \text{Cubic} < \text{Exponential} < \text{Factorial}$$

$$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$$



Key Algebraic Properties of Asymptotic Notations

Let $f(n)$ and $g(n)$ be positive functions.

1. **Transitivity:**
   - If $f(n) = O(g(n))$ and $g(n) = O(h(n))$, then $f(n) = O(h(n))$.
   - If $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n))$, then $f(n) = \Theta(h(n))$.
2. **Reflexivity:**
   - $f(n) = O(f(n))$
   - $f(n) = \Omega(f(n))$
   - $f(n) = \Theta(f(n))$
3. **Symmetry:**
   - $f(n) = \Theta(g(n))$ if and only if $g(n) = \Theta(f(n))$.
4. **Transpose Symmetry:**

- $f(n) = O(g(n))$ if and only if $g(n) = \Omega(f(n))$.
- $f(n) = o(g(n))$ if and only if $g(n) = \omega(f(n))$ (where ω is little-omega).

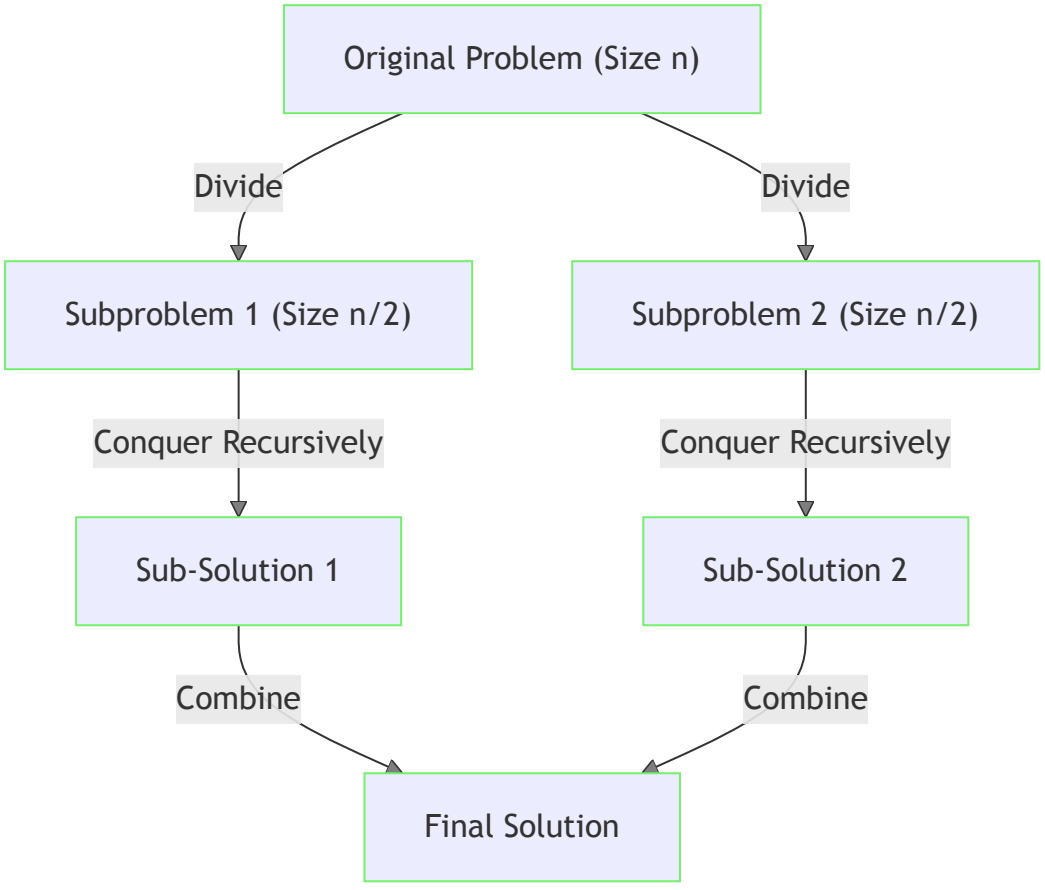
General Divide and Conquer Method

Core Concept

Divide and Conquer is a powerful algorithm design paradigm that operates by breaking down a large, complex problem into smaller, simpler subproblems of the same type, solving these subproblems, and then combining their individual solutions to form the solution to the original problem.

The strategy involves three main phases:

1. **Divide:** Partition the problem P of size n into k smaller, disjoint subproblems $P_1, P_2, \ldots, P_k$ ($1 < k \leq n$). Typically, the division yields subproblems of approximately equal size ($n/2$ or n/k).
2. **Conquer:** Solve each subproblem. If the subproblem sizes are small enough (base cases), solve them directly (using a base case solver). Otherwise, solve them recursively by applying the Divide and Conquer strategy.
3. **Combine:** Merge the solutions of the subproblems $S_1, S_2, \ldots, S_k$ to obtain the solution S for the original problem P.



Control Abstraction

A control abstraction is a formula or pseudocode that outlines the flow of control in a design strategy. The control abstraction for Divide and Conquer is expressed recursively as follows:

```
Algorithm DAndC(P)
// P is the problem to be solved
{
    if Small(P) then
        return Solve(P); // Base case: solve directly
    else
    {
        // Divide P into smaller subproblems P1, P2, ..., Pk
        Divide P into P1, P2, ..., Pk;

        // Conquer recursively
        S1 := DAndC(P1);
        S2 := DAndC(P2);
        ...
        Sk := DAndC(Pk);

        // Combine sub-solutions into final solution
        return Combine(S1, S2, ..., Sk);
    }
}
```

General Recurrence Relation

The running time $T(n)$ of a Divide and Conquer algorithm is modeled using a recurrence relation:

$$T(n) = \begin{cases} g(n) & \text{for small } n \\ aT(n/b) + f(n) & \text{for large } n \end{cases}$$

Where:

- a **(**$a \geq 1$**):** The number of subproblems generated in each split.
- n/b **(**$b > 1$**):** The size of each subproblem (assuming all subproblems are of equal size).
- $f(n)$**:** The time required to divide the input and combine the subproblem solutions.
- $g(n)$**:** The time required to solve the base case directly (typically $O(1)$).

The Master Theorem for Solving Recurrences

The Master Theorem provides a cookbook method for solving recurrence relations of the form $T(n) = aT(n/b) + f(n)$, where $a \geq 1$ and $b > 1$ are constants, and $f(n)$ is an asymptotically positive function.

We compare $f(n)$ with $n^{\log_b a}$ (the "boundary" function):

Case 1: $f(n) = O(n^c)$ where $c < \log_b a$

- If $f(n)$ grows asymptotically slower than $n^{\log_b a}$, then the work done at the leaves dominates the total running time.
- **Solution:** $T(n) = \Theta(n^{\log_b a})$.

Case 2: $f(n) = \Theta(n^c \log^k n)$ where $c = \log_b a$ and $k \geq 0$

- If $f(n)$ and $n^{\log_b a}$ grow at the same rate, then the work is distributed evenly across all levels of the recursion tree.
- **Solution:** $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$.
- *Special Case ($k = 0$):* If $f(n) = \Theta(n^{\log_b a})$, then $T(n) = \Theta(n^{\log_b a} \log n)$.

Case 3: $f(n) = \Omega(n^c)$ where $c > \log_b a$

- If $f(n)$ grows asymptotically faster than $n^{\log_b a}$, then the work done at the root (dividing and combining) dominates the running time.
- **Condition:** We must also satisfy the *regularity condition*: $a \cdot f(n/b) \leq d \cdot f(n)$ for some constant $d < 1$ and all sufficiently large n.
- **Solution:** $T(n) = \Theta(f(n))$.

Binary Search

Concepts & Pre-conditions

Binary Search is an efficient search algorithm used to find the position of a target element x within a **sorted array** (arranged in non-decreasing order).

- **Core Idea:** Instead of scanning elements sequentially (which takes $O(n)$ time), Binary Search repeatedly divides the search interval in half.
- **Pre-condition:** The input list must be pre-sorted. If not sorted, Binary Search will fail to return correct results.

Pseudocode

1. Iterative Binary Search

```
Algorithm BinarySearch(A, n, x)
// A is a sorted array of size n (1-indexed)
// x is the target value to find
{
    low := 1;
    high := n;
    while (low <= high) do
    {
        mid := floor((low + high) / 2);
        if (x = A[mid]) then
            return mid; // Target found
        else if (x < A[mid]) then
            high := mid - 1; // Discard right half
        else
            low := mid + 1; // Discard left half
    }
    return 0; // Target not found
}
```

2. Recursive Binary Search

```
Algorithm RecBinarySearch(A, low, high, x)
{
    if (low > high) then
        return 0; // Base case: not found

    mid := floor((low + high) / 2);
    if (x = A[mid]) then
        return mid; // Target found
    else if (x < A[mid]) then
        return RecBinarySearch(A, low, mid - 1, x); // Search left
    else
        return RecBinarySearch(A, mid + 1, high, x); // Search right
}
```

Step-by-Step Trace Example

Let's trace the search for target $x = 35$ in the sorted array:

$$A = [10, 14, 19, 26, 31, 35, 42, 44] \quad (\text{Size } n = 8)$$

Trace Table for $x = 35$:

Iteration	`low`	`high`	`mid`	`A[mid]`	Comparison / Action
1	1	8	$\lfloor (1+8)/2 \rfloor = 4$	$A[4] = 26$	$35 > 26 \implies$ Search right half. Set `low := mid + 1 = 5`
2	5	8	$\lfloor (5+8)/2 \rfloor = 6$	$A[6] = 35$	$35 = 35 \implies$ Element found! Return index 6.

Result: Element found at index 6.

Trace Table for a Missing Element $x = 40$:

Iteration	`low`	`high`	`mid`	`A[mid]`	Comparison / Action
1	1	8	4	$A[4] = 26$	$40 > 26 \implies$ Set `low := mid + 1 = 5`
2	5	8	6	$A[6] = 35$	$40 > 35 \implies$ Set `low := mid + 1 = 7`
3	7	8	7	$A[7] = 42$	$40 < 42 \implies$ Set `high := mid - 1 = 6`
Exit	7	6	-	-	Loop terminates because `low > high` ($7 > 6$). Return 0.

Complexity Analysis

Time Complexity

At each step of the algorithm, the search space is cut in half. The recurrence relation representing the worst-case number of comparisons $T(n)$ for an array of size n is:

$$T(n) = T(n/2) + \Theta(1)$$

Using Case 2 of the Master Theorem ($a = 1, b = 2, f(n) = \Theta(1)$):

- $\log_b a = \log_2 1 = 0$.
- Since $f(n) = \Theta(1) = \Theta(n^0)$, we have $c = 0 = \log_b a$.
- Therefore, $T(n) = \Theta(n^0 \log n) = \Theta(\log n)$.
- **Best-Case:** $\Theta(1)$ (when the target is found at the first `mid` check).
- **Worst-Case:** $\Theta(\log n)$ (when the target is at the ends or not in the array).
- **Average-Case:** $\Theta(\log n)$.

Space Complexity

- **Iterative Version:** $O(1)$ auxiliary space since we only use a few variables (`low`, `high`, `mid`).

- **Recursive Version:** $O(\log n)$ auxiliary space because of the recursion stack frames. The maximum stack depth is equal to the height of the decision tree, which is $\lfloor \log_2 n \rfloor + 1$.

Merge Sort

Concepts

Merge Sort is an external, stable sorting algorithm based on the Divide and Conquer strategy.

- **Core Idea:** It divides the unsorted list of size n into two halves of size $n/2$, sorts each half recursively, and then merges the two sorted halves back into a single sorted list.
- **Stability:** It maintains the relative order of equal elements, making it a stable sort.

Pseudocode

1. MergeSort Algorithm

```
Algorithm MergeSort(A, low, high)
// Sorts array A from index low to high
{
    if (low < high) then
    {
        mid := floor((low + high) / 2);
        MergeSort(A, low, mid);       // Sort left sub-array
        MergeSort(A, mid + 1, high);   // Sort right sub-array
        Merge(A, low, mid, high);     // Merge sorted halves
    }
}
```

2. Merge Algorithm (Combining Step)

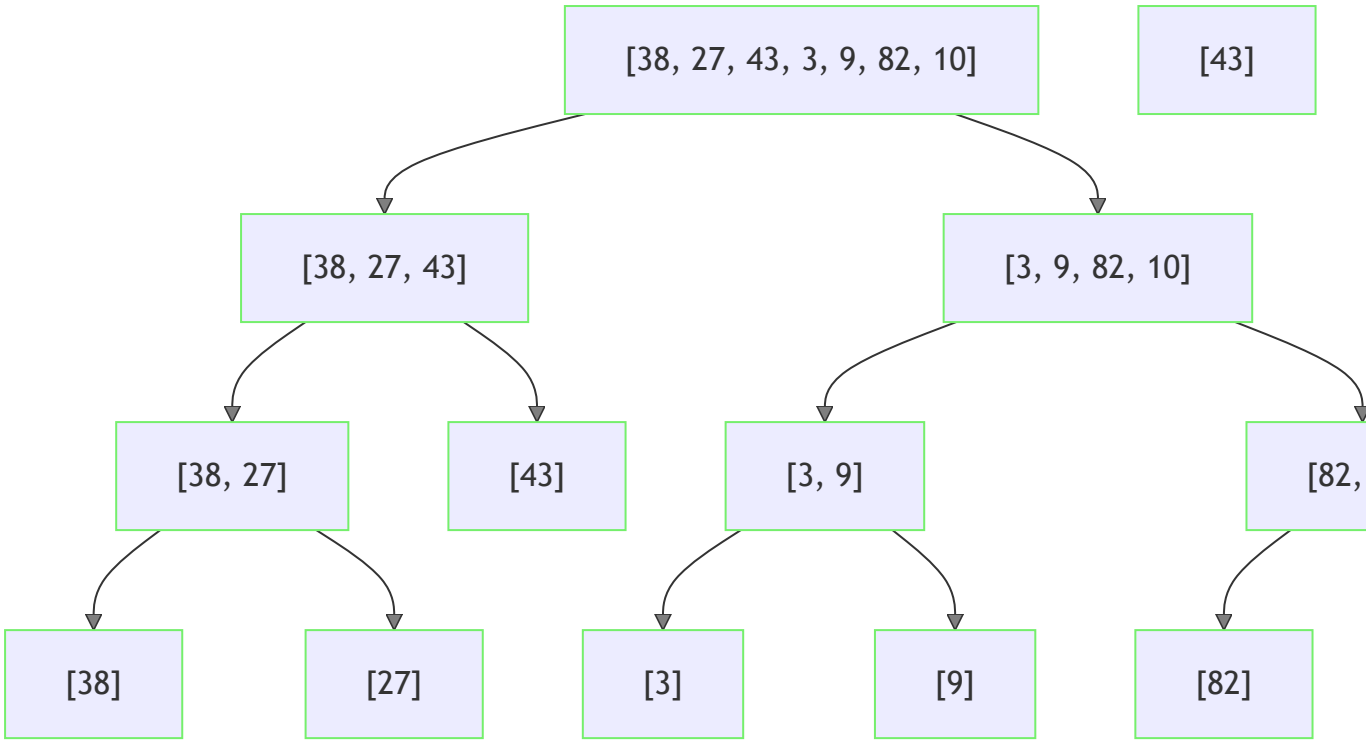
```
Algorithm Merge(A, low, mid, high)
// Merges two sorted sub-arrays: A[low..mid] and A[mid+1..high]
{
    h := low;   // Pointer for left sub-array
    i := low;   // Pointer for auxiliary array B
    j := mid + 1; // Pointer for right sub-array

    // Copy elements to auxiliary array B in sorted order
    while (h <= mid and j <= high) do
    {
        if (A[h] <= A[j]) then
        {
            B[i] := A[h];
            h := h + 1;
        }
        else
        {
            B[i] := A[j];
            j := j + 1;
        }
        i := i + 1;
    }

    // Copy any remaining elements of the left sub-array
    if (h > mid) then
        for k := j to high do
        {
            B[i] := A[k];
            i := i + 1;
        }
    else
        for k := h to mid do
        {
            B[i] := A[k];
            i := i + 1;
        }

    // Copy elements back from B to original array A
    for k := low to high do
        A[k] := B[k];
}
```

Step-by-Step Trace Example

Let's trace Merge Sort on array:

$$A = [38, 27, 43, 3, 9, 82, 10]$$

Division Phase (Recursive Splits):



Merge Phase (Combining Steps):

1. Merge `[38]` and `[27]` → `[27, 38]`
2. Merge `[27, 38]` and `[43]` → `[27, 38, 43]`
3. Merge `[3]` and `[9]` → `[3, 9]`
4. Merge `[82]` and `[10]` → `[10, 82]`
5. Merge `[3, 9]` and `[10, 82]` → `[3, 9, 10, 82]`
6. Merge `[27, 38, 43]` and `[3, 9, 10, 82]` → `[3, 9, 10, 27, 38, 42, 82]` (Final Sorted Array: `[3, 9, 10, 27, 38, 43, 82]`).

Complexity Analysis

Time Complexity

The recurrence relation for Merge Sort is:

$$T(n) = \begin{cases} a & \text{if } n = 1 \\ 2T(n/2) + cn & \text{if } n > 1 \end{cases}$$

Where cn represents the time taken by the `Merge` process to linearly scan and combine elements.
Using the Master Theorem ($a = 2, b = 2, f(n) = \Theta(n)$):

- $\log_b a = \log_2 2 = 1$.

- Since $f(n) = \Theta(n) = \Theta(n^1)$, we have $c = 1 = \log_b a$.
- Case 2 applies with $k = 0$:

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(n \log n)$$

- **Best, Worst, and Average Cases:** All are $\Theta(n \log n)$ because the algorithm divides the array and merges it regardless of the initial arrangement of elements.

Space Complexity

- **Auxiliary Space:** $O(n)$ is required to hold the auxiliary array `B` during the merging step.
- **Recursion Stack:** $O(\log n)$ to handle recursive frames.
- **Total Space Complexity:** $O(n)$ (dominated by the auxiliary array).

Quick Sort (Partition Exchange Sort)

Concepts

Quick Sort is an in-place, unstable sorting algorithm based on Divide and Conquer.

- **Core Idea:** Unlike Merge Sort, which divides the array at the exact midpoint, Quick Sort divides the array dynamically using a **pivot** element.
- **Partitioning:** The array is rearranged so that all elements smaller than or equal to the pivot are placed to its left, and all elements larger than or equal to the pivot are placed to its right. The sub-arrays are then sorted independently. No merge step is needed at the end because the elements are already in their correct partitions.

Pseudocode

1. QuickSort Algorithm

```
Algorithm QuickSort(A, low, high)
{
    if (low < high) then
    {
        // j is the correct index of the pivot element
        j := Partition(A, low, high + 1); // high + 1 acts as bound
        QuickSort(A, low, j - 1);         // Sort left partition
        QuickSort(A, j + 1, high);        // Sort right partition
    }
}
```

2. Partition Algorithm (Hoare-like standard)

To implement the partition, we append a sentinel value ∞ at the end of the array to prevent pointer i from running off the bounds of the array.

```
Algorithm Partition(A, m, p)
// m is low, p is high + 1. Pivot is A[m].
{
    pivot := A[m];
    i := m;
    j := p; // Started at high + 1

    repeat
    {
        // Move i right as long as elements are smaller than pivot
        repeat
            i := i + 1;
        until (A[i] >= pivot);

        // Move j left as long as elements are larger than pivot
        repeat
            j := j - 1;
        until (A[j] <= pivot);

        if (i < j) then
            Swap(A[i], A[j]);
    } until (i >= j);

    // Swap pivot to its final correct position
    Swap(A[m], A[j]);
    return j; // Return pivot position
}
```

Step-by-Step Trace Example

Let's trace the partition of the array:

$$A = [24, 9, 29, 14, 19, 27, 11]$$

We set a sentinel value $A[8] = \infty$ at the end. Thus:

$$A = [24, 9, 29, 14, 19, 27, 11, \infty]$$

- **Initial indices:** $m = 1$, $p = 8$.
- **Pivot:** $A[m] = A[1] = 24$.
- **Pointers:** $i = 1$, $j = 8$.

Partitioning Execution:

1. **Loop 1:**
   - Move i right: $A[2] = 9 < 24$, $A[3] = 29 \geq 24 \implies$ Stops at $i = 3$.
   - Move j left: $A[7] = 11 \leq 24 \implies$ Stops at $j = 7$.
   - Since $i < j$ $(3 < 7)$, Swap $A[3]$ and $A[7]$.
   - Array becomes: $A = [24, 9, \mathbf{11}, 14, 19, 27, \mathbf{29}, \infty]$
2. **Loop 2:**
   - Move i right: $A[4] = 14 < 24$, $A[5] = 19 < 24$, $A[6] = 27 \geq 24 \implies$ Stops at $i = 6$.
   - Move j left: $A[6] = 27 > 24$, $A[5] = 19 \leq 24 \implies$ Stops at $j = 5$.
   - Since $i \geq j$ $(6 \geq 5)$, exit the loop.
3. **Swap Pivot:**
   - Swap pivot $A[1]$ with $A[j] = A[5]$.
   - Array becomes: $[\mathbf{19}, 9, 11, 14, \mathbf{24}, 27, 29]$
   - Pivot 24 is now at index 5 (its final correct position).

Resulting Subproblems:

- Left partition: $[19, 9, 11, 14]$
- Right partition: $[27, 29]$

Complexity Analysis

Time Complexity

1. Worst-Case ($O(n^2)$)

- **Occurs when:** The input array is already sorted, reverse sorted, or contains all identical elements. In these cases, the partition element is always the minimum or maximum element, dividing the problem into subproblems of size 0 and $n - 1$.
- **Recurrence:**

$$T(n) = T(n-1) + \Theta(n)$$

- Solving by substitution:

$$T(n) = T(n-2) + (n-1) + n = \sum_{i=1}^{n} i = \Theta(n^2)$$

2. Best-Case ($\Theta(n \log n)$)

- **Occurs when:** The pivot always splits the array into two equal halves of size $n/2$.
- **Recurrence:**

$$T(n) = 2T(n/2) + \Theta(n)$$

- Solving via Master Theorem Case 2: $T(n) = \Theta(n \log n)$.

3. Average-Case ($\Theta(n \log n)$)

- Assuming random elements, the average split is reasonably balanced (e.g., $1:9$), which still yields a recursion depth of $O(\log n)$ and total time of $O(n \log n)$.

Space Complexity

- **Worst-Case:** $O(n)$ stack space due to unbalanced recursive calls.
- **Best-Case / Average-Case:** $O(\log n)$ stack space.

Strassen's Matrix Multiplication

Concepts

Let A and B be two $n \times n$ matrices. We want to calculate the product matrix $C = A \times B$.

Conventional Method ($O(n^3)$)

The standard algorithm uses three nested loops to multiply matrices:

```
for i := 1 to n do
    for j := 1 to n do
    {
        C[i, j] := 0;
        for k := 1 to n do
            C[i, j] := C[i, j] + A[i, k] * B[k, j];
    }
```

- This method does exactly n^3 multiplications and $n^3 - n^2$ additions, leading to a complexity of $\Theta(n^3)$.

Naive Divide & Conquer Approach

We can partition $n \times n$ matrices into four submatrices of size $n/2 \times n/2$:

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, \quad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Their product matrix C is defined as:

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

Where:

- $C_{11} = A_{11}B_{11} + A_{12}B_{21}$
- $C_{12} = A_{11}B_{12} + A_{12}B_{22}$
- $C_{21} = A_{21}B_{11} + A_{22}B_{21}$
- $C_{22} = A_{21}B_{12} + A_{22}B_{22}$
- **Analysis:** This formulation requires **8 multiplications** of submatrices and **4 additions**.
- **Recurrence:** $T(n) = 8T(n/2) + \Theta(n^2)$ (where n^2 is the cost of matrix additions).
- By Master Theorem Case 1 ($\log_2 8 = 3 > 2$): $T(n) = \Theta(n^3)$. This does not improve on the conventional method.

Strassen's Formulas

Volker Strassen discovered a way to compute the submatrices of C using only **7 multiplications** (instead of 8) and **18 additions/subtractions**:

We define 7 products P_1 to P_7:

$$P_1 = (A_{11} + A_{22})(B_{11} + B_{22})$$

$$P_2 = (A_{21} + A_{22})B_{11}$$

$$P_3 = A_{11}(B_{12} - B_{22})$$

$$P_4 = A_{22}(B_{21} - B_{11})$$

$$P_5 = (A_{11} + A_{12})B_{22}$$

$$P_6 = (A_{21} - A_{11})(B_{11} + B_{12})$$

$$P_7 = (A_{12} - A_{22})(B_{21} + B_{22})$$

Using these products, the submatrices of C are computed as:

$$C_{11} = P_1 + P_4 - P_5 + P_7$$

$$C_{12} = P_3 + P_5$$

$$C_{21} = P_2 + P_4$$

$$C_{22} = P_1 + P_3 - P_2 + P_6$$

Step-by-Step Numerical Trace Example

Let's multiply the two matrices:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, \quad B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}$$

Submatrix Blocks (since $n = 2$, elements are 1×1 matrices):

- $A_{11} = 1, A_{12} = 2, A_{21} = 3, A_{22} = 4$
- $B_{11} = 5, B_{12} = 6, B_{21} = 7, B_{22} = 8$

Step 1: Compute P_1 to P_7

- $P_1 = (A_{11} + A_{22})(B_{11} + B_{22}) = (1 + 4)(5 + 8) = 5 \times 13 = 65$
- $P_2 = (A_{21} + A_{22})B_{11} = (3 + 4) \times 5 = 7 \times 5 = 35$
- $P_3 = A_{11}(B_{12} - B_{22}) = 1 \times (6 - 8) = 1 \times (-2) = -2$
- $P_4 = A_{22}(B_{21} - B_{11}) = 4 \times (7 - 5) = 4 \times 2 = 8$
- $P_5 = (A_{11} + A_{12})B_{22} = (1 + 2) \times 8 = 3 \times 8 = 24$
- $P_6 = (A_{21} - A_{11})(B_{11} + B_{12}) = (3 - 1)(5 + 6) = 2 \times 11 = 22$
- $P_7 = (A_{12} - A_{22})(B_{21} + B_{22}) = (2 - 4)(7 + 8) = (-2) \times 15 = -30$

Step 2: Combine to form C

- $C_{11} = P_1 + P_4 - P_5 + P_7 = 65 + 8 - 24 + (-30) = 19$
- $C_{12} = P_3 + P_5 = -2 + 24 = 22$
- $C_{21} = P_2 + P_4 = 35 + 8 = 43$
- $C_{22} = P_1 + P_3 - P_2 + P_6 = 65 + (-2) - 35 + 22 = 50$

Result Matrix:

$$C = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix}$$

(Which is exactly correct as $1 \times 5 + 2 \times 7 = 19$, $1 \times 6 + 2 \times 8 = 22$, etc.)

Complexity Analysis

Time Complexity

The recurrence relation for Strassen's matrix multiplication is:

$$T(n) = 7T(n/2) + \Theta(n^2)$$

Using the Master Theorem ($a = 7, b = 2, f(n) = \Theta(n^2)$):

- $\log_b a = \log_2 7 \approx 2.807$.
- Since $f(n) = \Theta(n^2) = \Theta(n^2)$, we have $c = 2 < \log_b a$.
- Case 1 applies:

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^{\log_2 7}) \approx \Theta(n^{2.81})$$

Thus, Strassen's algorithm reduces matrix multiplication complexity from $\Theta(n^3)$ to $O(n^{2.81})$.

Limitations of Strassen's Algorithm

Despite its superior asymptotic complexity, Strassen's algorithm has several practical issues:

1. **Space Overhead:** It requires allocating many temporary submatrices during recursion, increasing the spatial constant factor.
2. **Crossover Point:** For small matrices (typically $n < 64$ or 128), the constant factor of additions outweighs the savings in multiplications. Standard multiplication is faster for small matrices.
3. **Numerical Stability:** The subtraction operations in Strassen's algorithm introduce larger numerical precision errors than conventional methods.
4. **Hardware Optimization:** Standard $O(n^3)$ multiplication can be easily parallelized and fits better into cache hierarchies (blocked algorithms).

> Website: https://r25.jntuh.harshrb.in
> YouTube: https://www.youtube.com/@QueryTheConcept