Introduction to Algorithms and Asymptotic Analysis

Prepared for DSC 314 course at IISER-TVM by Dr. Dhanyamol Antony.

Algorithm: Design

An algorithm is a finite sequence of logically related instructions to solve a computational problem. Any algorithm must have an input provided by the user and must produce an output. Computational instructions in the algorithm must involve only basic algebraic (arithmetic) operations and it should terminate after a finite number of steps. Finally, any algorithm must involve unambiguous instructions and produce the desired output.

Note: Although the phrase algorithm is associated with computer science, the notion of computation (algorithm) did exist for many centuries.

The definition of algorithm sparks natural fundamental questions:

- How to design an algorithm for a given problem?
- Is every problem algorithmically solvable? If so, how many algorithms can a problem have and how to find the efficient one?

We shall address these questions in this lecture. Let us consider an example of finding a maximum element in an array of size n.

Example: Finding a maximum element in an array

Algo Max-array(A, n)

```
Max = A[1];
for i = 2 to n do
    if (A[i] > Max) then
        Max = A[i];
return Max;
```

Note: The maximum can also be computed by sorting the array in an increasing order (decreasing order) and picking the last element (first element). There are at least five different algorithms to find a maximum element and therefore, it is natural ask for an efficient algorithm. This calls for the study of analysis of algorithms.

Types of Algorithm

There are two ways to write an algorithm, namely,

- Iterative algorithm
- Recursive algorithm

Iterative Algorithm

Fact(n)

```
for i = 1 to n
    fact = fact * i;
return fact;
```

Here the factorial is calculated as $1 \times 2 \times 3 \times \cdots \times n$.

Recursive Algorithm

Fact(n)

```
if n = 1
    return 1;
else
    return n * Fact(n-1);
```

Here the factorial is calculated as $n \times (n-1) \times \cdots \times 1$.

Algorithm: Analysis

Designing an algorithm is not just about solving a problem correctly, but doing so efficiently. The efficiency of an algorithm is measured in terms of its **time** and **space** requirements.

- **Time Complexity:** How long an algorithm takes to run, as a function of input size.
- **Space Complexity:** How much memory it consumes.

In practice, we focus more on time than space.

Measuring Efficiency

Algorithm analysis depends critically on the **computational model** used. For example, the statement that an algorithm runs in $O(n \log n)$ time is only valid under a specific model of computation. The validity of this statement may not hold if the underlying model changes. We analyze algorithms independent of hardware by counting the number of **basic operations**, such as:

- Comparisons (e.g., `if A[i] < A[j]`)
- Assignments (e.g., `x = y`)
- Swaps or arithmetic operations

Computational Model and Analysis

Before formalizing what we mean by a computational model, let us first explore a concrete example: the problem of computing Fibonacci numbers. This will help us to analyse how different models can significantly influence the perceived efficiency of an algorithm.

1 Computing Fibonacci Numbers

The Fibonacci sequence is defined by the recurrence:

$$F_i = \begin{cases} 0 & i = 0 \\ 1 & i = 1 \\ F_{i-1} + F_{i-2} & i \geq 2 \end{cases}$$

Method 1: Naive Recursion A straightforward implementation using the recursive definition of Fibonacci numbers leads to an exponential number of operations.

The Fibonacci numbers are defined by the recurrence relation:

$$F_n = F_{n-1} + F_{n-2}, \quad \text{for } n \geq 2$$

with initial conditions:

$$F_0 = 0, \quad F_1 = 1$$

Assume a solution of the form $F_n = r^n$. Substituting into the recurrence:

$$r^n = r^{n-1} + r^{n-2} \Rightarrow r^n - r^{n-1} - r^{n-2} = 0$$

Divide both sides by r^{n-2} (assuming $r \neq 0$):

$$r^2 - r - 1 = 0$$

Solve the quadratic equation:

$$r^2 - r - 1 = 0 \Rightarrow r = \frac{1 \pm \sqrt{5}}{2}$$

Let

$$\phi = \frac{1 + \sqrt{5}}{2} \quad \text{and} \quad \psi = \frac{1 - \sqrt{5}}{2}$$

The general solution to the recurrence is:

$$F_n = A\phi^n + B\psi^n$$

Use the initial conditions to solve for A and B:

$$F_0 = 0 = A + B \Rightarrow B = -A$$
$$F_1 = 1 = A\phi + B\psi = A(\phi - \psi)$$

$$A = \frac{1}{\phi - \psi}, \quad \phi - \psi = \sqrt{5} \Rightarrow A = \frac{1}{\sqrt{5}}, \quad B = -\frac{1}{\sqrt{5}}$$

$$\boxed{F_n = \frac{1}{\sqrt{5}} \left(\left(\frac{1+\sqrt{5}}{2} \right)^n - \left(\frac{1-\sqrt{5}}{2} \right)^n \right)}$$

The total number of additions grows proportionally to the value of F_n, resulting in an exponential time algorithm, which is highly inefficient and impractical for large values of n.

Method 2: Iterative (Dynamic Programming) Notice that to compute each new term in the Fibonacci sequence, we only require the previous two terms. By applying the principle of **dynamic programming**, we can compute F_i iteratively starting from the base cases $F_0 = 0$ and $F_1 = 1$, and using the relation $F_i = F_{i-1} + F_{i-2}$ for $i \geq 2$.

This approach performs approximately n additions. Compute $F_0, F_1, F_2, \ldots, F_n$ in order, forming each number by summing the two previous. Running time: $\theta(n)$. Given that the n-th Fibonacci number has at most n bits, it is natural to ask whether a faster algorithm exists.

Method 3: Matrix Exponentiation (recursive squaring)

Define the transformation matrix:

$$T = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}$$

Then the recurrence can be rewritten in matrix form as:

$$\begin{bmatrix} F_n \\ F_{n-1} \end{bmatrix} = T^{n-1} \begin{bmatrix} F_1 \\ F_0 \end{bmatrix} = T^{n-1} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

Then:

$$T^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \quad \text{for all } n \geq 1.$$

Base Case: $n = 1$

$$T^1 = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_2 & F_1 \\ F_1 & F_0 \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \quad \checkmark$$

Inductive Step

Assume the formula holds for n. That is,

$$T^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

Then,

$$T^{n+1} = T^n \cdot T = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix} \cdot \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} = \begin{bmatrix} F_{n+1} + F_n & F_{n+1} \\ F_n + F_{n-1} & F_n \end{bmatrix} = \begin{bmatrix} F_{n+2} & F_{n+1} \\ F_{n+1} & F_n \end{bmatrix} \quad \checkmark$$

Hence, by induction, the formula holds for all $n \geq 1$.

Let us now analyze the time complexity of computing F_n using matrix exponentiation.

To compute T^n, we use binary exponentiation (also called exponentiation by squaring), which works as follows:

- If $n = 0$, return the identity matrix.
- If n is even, compute $T^n = (T^{n/2})^2$
- If n is odd, compute $T^n = T \cdot T^{n-1}$

Each recursive step reduces the exponent by half, leading to $\log_2 n$ steps. Each step involves multiplying two 2×2 matrices, which takes constant time: $O(1)$. Therefore, the total time complexity is: $\mathcal{O}(\log n)$. This is a significant improvement over the naive recursive algorithm or even iterative $O(n)$ solutions.

Although computers come in thousands of variations with differing architectures and internal organizations, it is most useful to think of them at the level of **assembly language**. Despite these architectural differences, assembly-level support is typically quite similar. The main variations are in the **number of registers** and the **word length** of the machine. However, these differences are generally within a small constant factor—usually just a factor of two—and are thus **asymptotically equivalent**.

In summary, we can abstractly model any computer as a machine that supports a **basic instruction set** consisting of arithmetic operations, logical operations, and memory access (including indirect addressing). To keep our discussion general and practical, we avoid the low-level details of specific instruction sets and assume that any instruction on one machine can be simulated using a **constant number of instructions** on another. Since algorithm analysis typically counts the number of operations, rather than their precise execution times (which may vary by a constant factor), this abstraction is well justified.

In the **uniform cost model**, each arithmetic or logical operation is assumed to take constant time, regardless of the size of the operands. This model is commonly used for problems such as **sorting, selection, merging**, and many **data structure operations**, where we often

count only the number of **comparisons**, assuming each takes $O(1)$ time. This is generally acceptable, as the size of elements in such problems typically does not grow during the execution of the algorithm.

However, this assumption breaks down for problems where the size of numbers grows rapidly. For example, consider repeatedly squaring the number 2, n times. This produces 2^{2^n}, a number requiring 2^n bits to represent. Clearly, it is unrealistic to assume such a number can be written or stored in $O(n)$ time. Thus, for such problems, the uniform cost model **does not realistically capture the true computational cost**.

At the other extreme, we have the **logarithmic cost model**, where the cost of an operation is proportional to the **length (in bits) of the operands**. This model is aligned with physical computation and closely resembles the **Turing Machine model**, which is widely used in **computational complexity theory**.

The most commonly used model in algorithm analysis, however, is a **hybrid approach**. We assume that for an input of size n, any operation on operands of size $O(\log n)$ takes **constant time**, i.e., $O(1)$. This is justified as follows: modern processors are built to perform arithmetic operations (such as addition, multiplication, division) in a **fixed number of clock cycles**, provided the operands **fit into a word**. Since addressing n memory locations requires $\log n$ bits, it is natural to treat $\log n$-bit words as the unit of computation.

Modern processors typically have **64-bit word sizes**, which support addressing up to 2^{64} memory locations—well beyond the needs of most algorithms. Thus, we will generally use the **Random Access Machine (RAM) model**, where memory can be accessed in constant time and arithmetic on word-sized operands takes constant time. However, for problems involving **arbitrary-precision numbers** (like the repeated squaring example or large integer multiplication), we will resort to the **logarithmic cost model** to reflect the true computational complexity.

As a good practice, when designing an algorithm, you should **estimate the maximum size of numbers** involved. If all operands remain within $O(\log n)$ bits, it is safe to use the RAM model.

Input Size

The running time $T(n)$ of an algorithm for a problem with input size n depends on n. $T(n)$ represents the maximum amount of time (worst case) the algorithm takes for any input of size n. This gives a guarantee that the algorithm will never take more time than $T(n)$, regardless of the input. But, how do we fix the input size? A natural parameter for:

- Sorting and other problems on arrays: n is the number of elements in the array - Larger arrays will take longer to sort.
- Graph algorithms: n = number of vertices, m = number of edges
- Numeric problems: input size = number of digits $\approx \log_b n$

- Arithmetic operations such as Addition (with carry), Subtraction (with borrow), Multiplication, Long division, are performed **digit by digit**, similar to how we do them manually.

- Therefore, the **time complexity** of such operations depends on the **number of digits** in the input, not the numeric value itself.

- A number n written in base b has approximately $\log_b n$ digits.

Order of growth

When comparing time complexity functions $T(n)$ across different problems, it is important to focus on the **order of growth** (asymptotics) and **ignore constant factors**. For example, consider:

$$f(n) = n^3 \quad \text{and} \quad g(n) = 5000n^2$$

Although $g(n)$ has a large constant, $f(n)$ has a higher-order term. For small values of n, we may observe:

$$f(n) < g(n)$$

However, eventually, $f(n)$ will grow faster than $g(n)$. To find the crossover point:

$$n^3 = 5000n^2 \Rightarrow n = 5000$$

Thus, for $n > 5000$, we have $f(n) > g(n)$.

What happens in the limit, as n increases? When comparing functions $T(n)$, we focus on their **asymptotic behavior** as $n \to \infty$. Asymptotic complexity describes the behavior of an algorithm's running time (or space usage) as the size of the input grows toward infinity. It gives an upper bound, lower bound, or tight bound on performance, abstracting away constant factors and lower-order terms. When comparing algorithms, we focus on their **asymptotic behavior** — how the runtime function $T(n)$ grows as n becomes large.

There is flexibility in defining what counts as a "basic operation". Example: Swapping two variables involves three assignments:

$$\texttt{tmp} \leftarrow x, \quad x \leftarrow y, \quad y \leftarrow \texttt{tmp}$$

Thus, number of swaps is 3 times number of assignments. If we count each assignment separately, the total count changes by a constant factor. As long as we focus on asymptotic complexity, the exact choice doesn't affect the overall order of growth.

Worst-Case Complexity

- For each input size n, the worst-case input yields the maximum runtime.

- Example: Search for K in an unsorted array A:

```
i ← 0 while i < n and A[i] ≠ K do
    i ← i + 1
if i < n return i
else return -1
```

- Worst-case: K is not in the array $\Rightarrow$ scan all n elements.
- Worst-case time complexity is $\mathcal{O}(n)$.
- Upper bound for the overall running time. Here worst case is proportional to n for array size n

Average-Case Complexity

- Worst-case complexity can be too pessimistic.
- Average-case complexity considers expected time over all possible inputs.
- Hard to compute in practice:
  - What is the distribution over inputs?
  - Are all inputs equally likely?
  - Need probability distribution over inputs

Worst-Case vs Average-Case

- Worst-case complexity may not reflect typical performance.
- Average-case complexity is often more informative but also harder to compute.
- A good worst-case upper bound is often practically useful.
- A bad worst case upper bound may be less informative. Here try to classify worst case inputs to find simpler subclasses that are easier to analyze.

Asymptotic complexity analysis

- We measure time complexity **up to an order of magnitude**.
- Constant factors are ignored when comparing growth rates.
- We use asymptotic notation to describe how functions grow as input size n increases.

1. Big $\mathcal{O}$ Notation ($\mathcal{O}(g(n))$) – Upper Bound

Definition: A function $t(n)$ is said to be $\mathcal{O}(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that

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

- This provides an upper bound on growth.
- It describes the worst-case time complexity.

Example 1:

Consider the function:

$$t(n) = 100n + 5$$

$\Longrightarrow$

$$\begin{aligned} 100n + 5 &\leq 100n + n, \quad \text{for } n \geq 5 \\ &= 101n \leq 101n^2 \\ \Rightarrow \quad t(n) &\leq 101n^2 \text{ for } n \geq 5 \end{aligned}$$

So we can choose $n_0 = 5$, $c = 101$.

Alternatively:

$$\begin{aligned} 100n + 5 \leq 100n + 5n = 105n \leq 105n^2, \quad \text{for } n \geq 1 \\ \Rightarrow \quad t(n) \leq 105n^2 \text{ for } n \geq 1 \end{aligned}$$

Here, $n_0 = 1$, $c = 105$ also works.

The constants n_0 and c are **not unique**. Big-O describes an asymptotic upper bound, so any function that eventually grows faster than $t(n)$ qualifies. Hence, by similar logic, we can also say:

$$100n + 5 \in \mathcal{O}(n)$$

since:

$$100n + 5 \leq 105n, \quad \text{for } n \geq 1$$

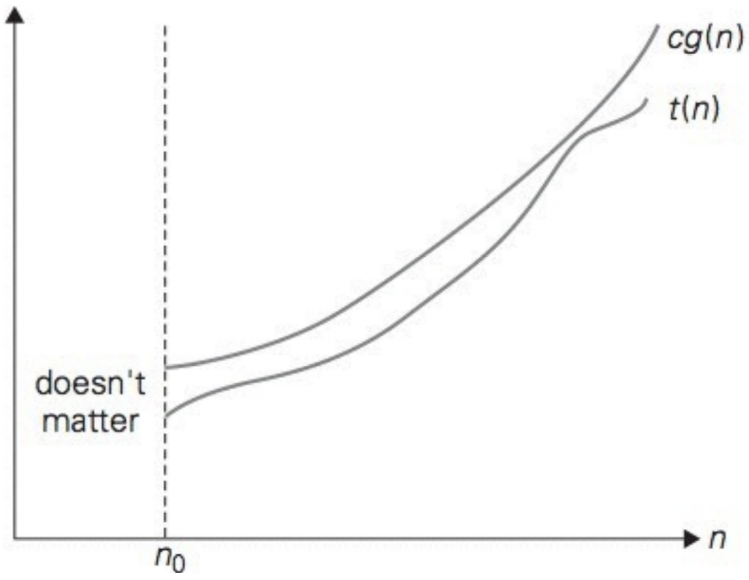

Figure 1: Big $\mathcal{O}$ Notation : $t(n) \leq c \cdot g(n)$ for $n \geq n_0$

Example 2: Consider the function:

$$t(n) = 100n^2 + 20n + 5$$

To prove $t(n) \in \mathcal{O}(n^2)$, we find constants c and n_0 such that:

$$t(n) \leq c \cdot n^2 \quad \text{for all } n \geq n_0$$

$\Longrightarrow$

$$\begin{aligned} 100n^2 + 20n + 5 &\leq 100n^2 + 20n^2 + 5n^2, \quad \text{for } n \geq 1 \\ &= (100 + 20 + 5)n^2 = 125n^2 \end{aligned}$$

Thus,

$$t(n) \leq 125n^2 \quad \text{for } n \geq 1 \Rightarrow t(n) \in \mathcal{O}(n^2)$$

with constants $c = 125$, $n_0 = 1$.

- What matters in asymptotic analysis is the **dominant term**.
- In this case, $100n^2$ dominates the lower-order terms $20n$ and the constant 5.
- So even though the full expression includes smaller terms, the function grows like n^2 in the limit.

Example 3: Suppose we want to test whether:

$$f(n) = n^3 \in \mathcal{O}(n^2)$$

By definition, for $f(n) \in \mathcal{O}(n^2)$, there must exist constants $c > 0$ and $n_0 \geq 0$ such that:

$$n^3 \leq c \cdot n^2 \quad \text{for all } n \geq n_0$$

- Divide both sides by n^2 (valid for $n > 0$):

$$\frac{n^3}{n^2} = n \leq c$$

- But this implies $n \leq c$ for all large n, which is a contradiction.
- No matter how large c is, n will eventually exceed c as $n \to \infty$.

$$\implies \qquad n^3 \notin \mathcal{O}(n^2)$$

That is, n^3 grows strictly faster than n^2, and cannot be upper bounded by cn^2 asymptotically.

Combining Asymptotic Complexities

Suppose:

$$f_1(n) \in \mathcal{O}(g_1(n)) \quad \text{and} \quad f_2(n) \in \mathcal{O}(g_2(n))$$

Then:

$$f_1(n) + f_2(n) \in \mathcal{O}(\max(g_1(n), g_2(n)))$$

Proof

By the definition of Big-O, there exist constants c_1, n_1 such that:

$$f_1(n) \leq c_1 \cdot g_1(n), \quad \text{for all } n \geq n_1$$

Similarly, there exist constants c_2, n_2 such that:

$$f_2(n) \leq c_2 \cdot g_2(n), \quad \text{for all } n \geq n_2$$

Let $n_0 = \max(n_1, n_2)$. Then for all $n \geq n_0$, we have:

$$f_1(n) + f_2(n) \leq c_1 g_1(n) + c_2 g_2(n)$$

Observe that:

$$c_1 g_1(n) + c_2 g_2(n) \leq (c_1 + c_2) \cdot \max(g_1(n), g_2(n))$$

Hence,

$$f_1(n) + f_2(n) \leq C \cdot \max(g_1(n), g_2(n)), \quad \text{for } C = c_1 + c_2, \text{ and } n \geq n_0$$

Therefore:

$$f_1(n) + f_2(n) \in \mathcal{O}(\max(g_1(n), g_2(n)))$$

- Many algorithms consist of multiple phases.
- Suppose:

- Phase A takes $\mathcal{O}(g_A(n))$ time
- Phase B takes $\mathcal{O}(g_B(n))$ time

- Then the total time is:
$$\mathcal{O}(\max(g_A(n), g_B(n)))$$
- If an algorithm has multiple phases with different time complexities, the overall complexity is dominated by the slowest (i.e., least efficient) phase.

2. Big Omega Notation ($\Omega(g(n))$) – Lower Bound

Definition: A function $t(n)$ is $\Omega(g(n))$ if there exist constants $c > 0$ and $n_0 \geq 0$ such that:

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

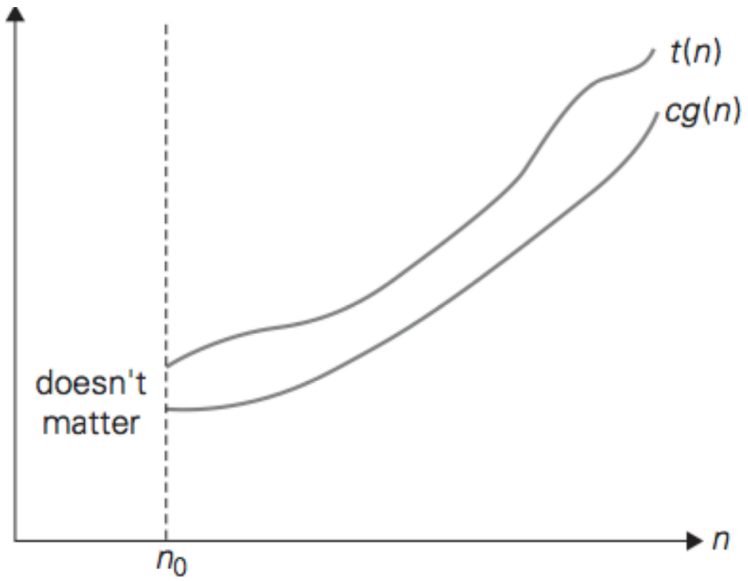

Figure 2: Big Omega Notation: $t(n) \geq c \cdot g(n)$ for $n \geq n_0$

Example:

$$n^3 \geq n^2, \quad \forall n \geq 1 \Rightarrow n^3 \in \Omega(n^2)$$

We know:

$$n^3 \geq n^2, \quad \text{for all } n \geq 0$$

So we can choose:

$$c = 1, \quad n_0 = 0 \Rightarrow n^3 \in \Omega(n^2)$$

- We establish lower bounds for problems as a whole, not for individual algorithms
- Example:

Any comparison-based sorting algorithm requires at least $\Omega(n \log n)$ comparisons in the worst case.

- This means that no matter how clever the algorithm, we cannot beat $n \log n$ comparisons in the general case.

3. Big Theta Notation ($\Theta(g(n))$) – Tight Bound

Definition: A function $t(n)$ is $\Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 \geq 0$ such that:

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

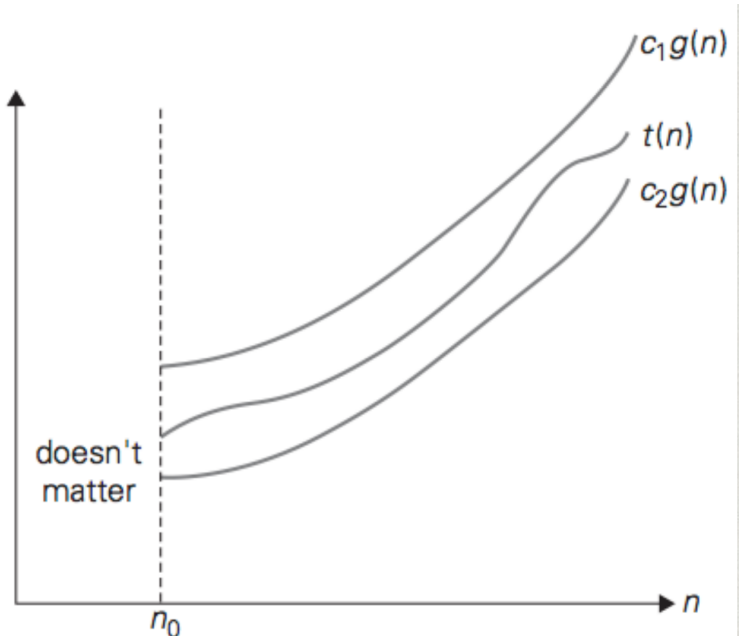

Figure 3: Big Omega Notation: $c_1 g(n) t(n) \geq c_2 g(n)$ for $n \geq n_0$

Example: Consider the function:

$$f(n) = \frac{n(n-1)}{2}$$

$$\begin{aligned} f(n) &= \frac{n(n-1)}{2} \\ &= \frac{n^2 - n}{2} \\ &\leq \frac{n^2}{2}, \quad \text{for all } n \geq 0 \end{aligned}$$

So: $f(n) \leq \frac{1}{2}n^2$ for $n \geq 0$

$$\Rightarrow f(n) \in \mathcal{O}(n^2), \quad \text{with } c_1 = \frac{1}{2},\ n_0 = 0, \textbf{(Upper bound Big-}O\textbf{)}$$

$$\begin{aligned} f(n) &= \frac{n(n-1)}{2} = \frac{n^2 - n}{2} \\ &= \frac{n^2}{2} - \frac{n}{2} \\ &\geq \frac{n^2}{2} - \left(\frac{n}{2} \cdot \frac{n}{2}\right) = \frac{n^2}{2} - \frac{n^2}{4} = \frac{n^2}{4}, \quad \text{for } n \geq 2 \end{aligned}$$

So: $f(n) \geq \frac{1}{4}n^2$ for $n \geq 2$

$$\Rightarrow f(n) \in \Omega(n^2), \quad \text{with } c_2 = \frac{1}{4},\ n_0 = 2, \textbf{(Lower Bound (Big-}\Omega\textbf{))}$$

Since:

$$\frac{1}{4}n^2 \leq f(n) \leq \frac{1}{2}n^2, \quad \text{for all } n \geq 2$$

We conclude:

$$f(n) = \frac{n(n-1)}{2} \in \Theta(n^2)$$

Constants used: $c_1 = \frac{1}{2},\ c_2 = \frac{1}{4},\ n_0 = 2$

In summary:

- $t(n) = \mathcal{O}(g(n))$: Upper bound — algorithm will not take more time than $g(n)$ asymptotically.
- $t(n) = \Omega(g(n))$: Lower bound — problem requires at least $g(n)$ time.
- $t(n) = \Theta(g(n))$: Tight bound — the algorithm is optimal up to constant factors. i.e., it is matching with the upper and lower bounds and hence best possible algorithm has been found.

Examples of Efficiency

Example 1: Sorting a Large Dataset

- Naive algorithm (e.g., Bubble Sort): $O(n^2)$
- Efficient algorithm (e.g., Merge Sort): $O(n \log n)$

- For $n = 10^9$:
  - $n^2 = 10^{18}$ operations $\Rightarrow$ 300 years
  - $n \log n \approx 3 \times 10^{10}$ ops $\Rightarrow$ 5 minutes

Example 2: Real-Time Video Game

- Problem: Find the closest pair among $n = 5 \times 10^5$ objects
- Naive algorithm: $O(n^2) \Rightarrow$ 42 minutes
- Efficient algorithm: $O(n \log n) \Rightarrow$ fractions of a second

Typical Complexity Classes

- $\log n$ — logarithmic
- n — linear
- $n \log n$ — linearithmic
- n^2, n^3 — polynomial
- 2^n, $n!$ — exponential/factorial

Summary

- Analyzing efficiency is critical for algorithm design.
- Input size directly impacts running time.
- Worst-case analysis gives us a strong performance guarantee.
- Average-case complexity is valuable when the input distribution is known.
- $f(n) = O(g(n))$ means $g(n)$ is an upper bound — useful for worst-case analysis.
- $f(n) = \Omega(g(n))$ means $g(n)$ is a lower bound — often used for problem classes.
- $f(n) = \Theta(g(n))$ means a tight bound — best possible matching upper and lower bounds.

1. **f(n) = 2n + 3**

   Analyze the asymptotic upper bound of the function $f(n) = 2n + 3$.

   Suppose $f(n) = O(n)$

We aim to find constants $c > 0$ and $n_0 \geq 1$ such that:

$$f(n) = 2n + 3 \leq c \cdot n \quad \forall n \geq n_0$$

Observe:

$$2n + 3 \leq 10n \quad \text{for all } n \geq 1$$

Hence, choosing $c = 10$ and $n_0 = 1$, we conclude:

$$f(n) = O(n)$$

Also: $f(n) = O(n^2)$
Since $n \leq n^2$ for $n \geq 1$, it follows:

$$2n + 3 \leq 2n^2 + 3n^2 = 5n^2 \Rightarrow f(n) = O(n^2)$$

Although $f(n) = O(n^2)$ is mathematically correct, it is not the tightest bound. The tightest upper bound remains $O(n)$, which best reflects the growth rate of $f(n)$.

Hierarchy of Common Asymptotic Classes

$$O(1) < O(\log n) < O(\sqrt{n}) < O(n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(3^n) < O(n^n)$$

Step-count Method and Asymptotic Notation

In this section, we shall look at analysis of algorithms using step count method. Time and space complexity are calculated using step count method. Some basic assumptions are:

- There is no count for { and }.
- Each basic statement like *assignment* and *return* has a count of 1.
- If a basic statement is iterated, then multiply by the number of times the loop is run.
- If a loop statement is iterated n times, it has a count of $(n + 1)$. Here the loop runs n times for the true case and a check is performed for the loop exit (the false condition), hence the additional 1 in the count.

Examples for Step-Count Calculation

1. Sum of elements in an array

Algorithm Sum(a, n)

```
{
sum = 0;
for i = 1 to n do
    sum = sum + a[i];
return sum;
}
```

Step-count (Time Complexity)

- `sum = 0;` 1
- `for i = 1 to n do` $n + 1$
- `sum = sum + a[i];` n
- `return sum;` 1

$$\text{Total Time Complexity} = 2n + 3$$

Step-count (Space Complexity)

- 1 word for `sum`
- 1 word each for `i` and `n`
- n words for array `a[]`

$$\text{Total Space} = (n + 3) \text{ words}$$

2. Adding two matrices of order m and n

Algorithm Add(a, b, c, m, n)

```
{
for i = 1 to m do
    for j = 1 to n do
        c[i,j] = a[i,j] + b[i,j];
}
```

Step Count

- Outer loop `for i = 1 to m` $m+1$
- Inner loop `for j = 1 to n` $m(n+1)$
- Assignment `c[i,j] = a[i,j] + b[i,j]` mn

$$\text{Total number of steps} = 2mn + 2m + 2$$

Note: The first `for` loop is executed $m+1$ times. The first m calls are true calls during which the inner loop is executed, and the $(m+1)$-th call is a false call.

3. Fibonacci series

Algorithm Fibonacci(n)

```
{
if n <= 1 then
    output n
else
    f2 = 0;
    f1 = 1;
    for i = 2 to n do
    {
        f = f1 + f2;
        f2 = f1;
        f1 = f;
    }
    output f
}
```

Step Count

- `if n <= 1` 1
- `f2 = 0;` 1
- `f1 = 1;` 1
- `for i = 2 to n` n
- `f = f1 + f2;` $n-1$
- `f2 = f1;` $n-1$

- `f1 = f;` $n-1$
- `output f` 1

Total number of steps $= 4n + 1$

Note: If $n \leq 1$, then the step count is just 2, and it is $4n + 1$ otherwise.

4. Recursive sum of elements in an array

Algorithm RecursiveSum(a, n)

```
{
if n <= 0 then
    return 0;
else
    return RecursiveSum(a, n-1) + a[n];
}
```

The step count of two is added at each recursive call: one count for making a recursive call with $(n-1)$ size input and the other count is for addition when the recursion bottoms out.

Let the step count of an array of size n be denoted as $T(n)$. Then,

$$T(n) = 3 + T(n-1), \quad n > 0$$
$$T(n) = 2, \quad n \leq 0$$

Solving the above recurrence relation yields:

$$T(n) = 3n + 2$$

Order of Growth

Order of growth gives a simple characterization of the algorithm's efficiency by identifying the relatively significant term in the step count.

For example, for $2n^2 + 3n + 1$, the order of growth depends on $2n^2$.

Properties of Asymptotic Notation

Reflexivity

$$f(n) = O(f(n)), \quad f(n) = \Omega(f(n)), \quad f(n) = \Theta(f(n))$$

Symmetry

$$f(n) = \Theta(g(n)) \iff g(n) = \Theta(f(n))$$

Transitivity

$$f(n) = O(g(n)) \wedge g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$$

Transpose Symmetry

$$f(n) = O(g(n)) \iff g(n) = \Omega(f(n))$$