People’s Democratic Republic of Algeria
Ministry of higher education and scientific research
University of Mohamed Boudiaf - M’sila

Faculty of Mathematics and Computer Science
Department of Computer Science

2nd Year License (2L)

Data Structures and Algorithms 3 (DSA3)

Semester : 03
2025/2026

Directed by

Dr. DABBA ALI

➢ Contact us

alidabba@gmail.com

ali.dabba@univ-msila.dz

- ➢ If you have any problems or difficulties, please contact me or your tutorial/practical teacher.
- ➢ We are at your disposal to help you

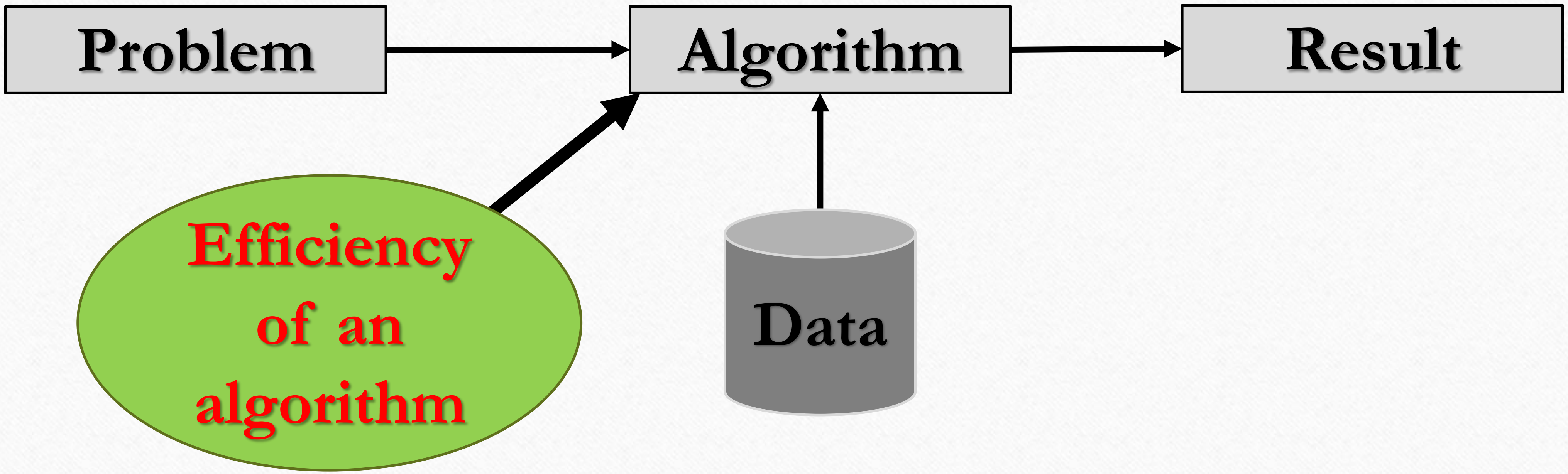
CHAPTER 1

Algorithm Analysis

Plan

I. Introduction
II. Data Structure
III. What is an Algorithm?
IV. Algorithm Analysis
V. Algorithm Complexity
VI. Mathematical Notation
VII. Running-Time Calculations

I. Introduction



Efficiency of an algorithm can be measured in terms of:

- Execution time (**time complexity**)
- The amount of memory required (**space complexity**)

II. Data Structure

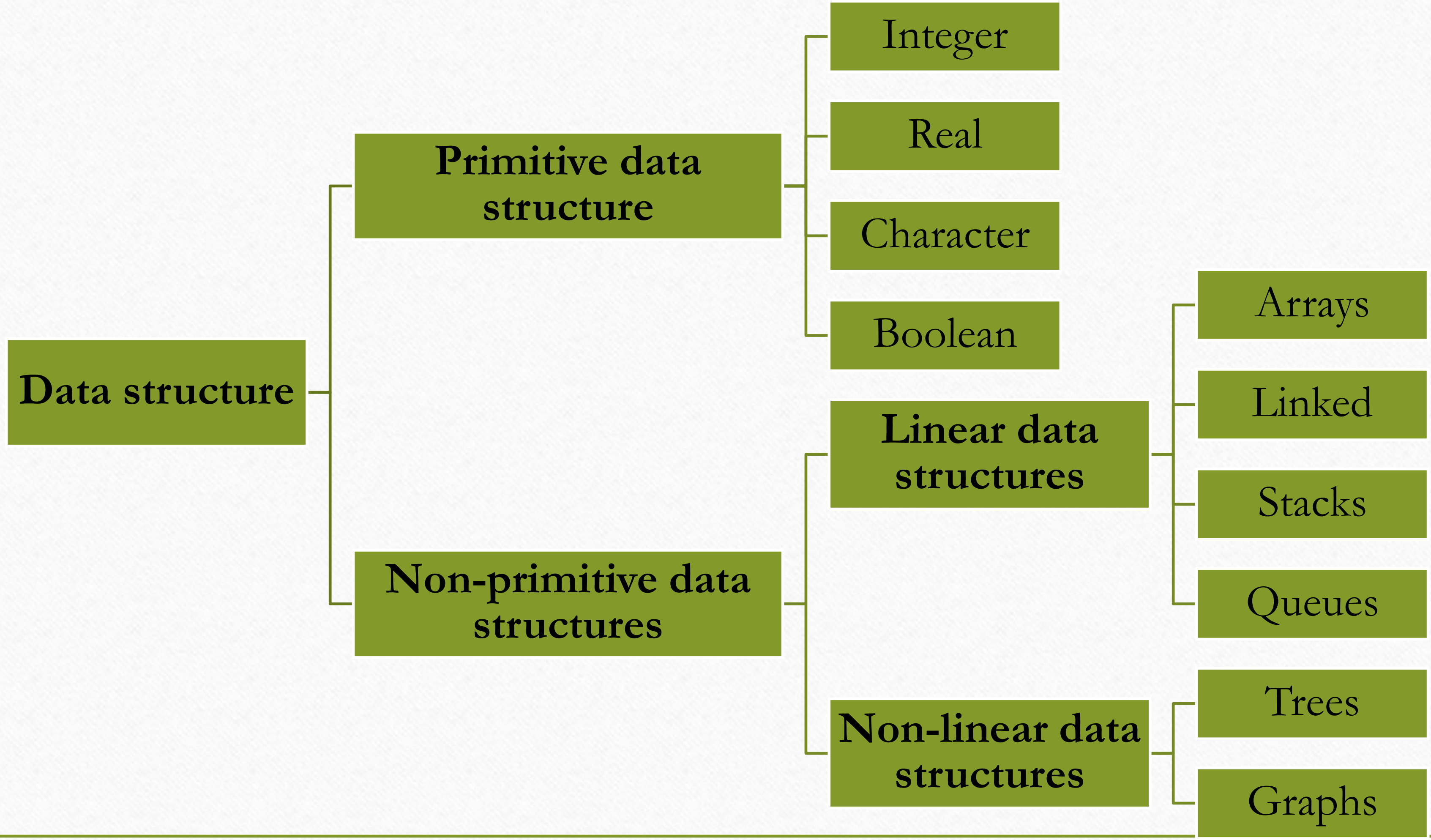
Linus Torvalds (creator of Linux)

" I will, in fact, claim that the difference between a bad programmer and a good one is whether he considers his code or his data structures more important. **Bad programmers worry about the code. Good programmers worry about data structures and their relationships.** "

II. Data Structure

- Data structure is a particular way of storing and organizing data in a computer so that it can be used efficiently.
- A data structure is a special format for organizing and storing data.
- General data structure types include arrays, files, linked lists, stacks, queues, trees, graphs, and so on.

II. Data Structure



III. What is an Algorithm?

Algorithm is **a step-by-step procedure to solve a problem**, where each step indicates an intermediate task. Algorithm contains **a finite number of instructions** to be executed in a certain order to get the desired output. Algorithms are **generally created independent of underlying languages**, i.e., an algorithm can be implemented in more than one programming language.

III.1. Characteristics of Algorithms

The main Characteristics or features of Algorithms are:

- **Input**
- **Definiteness**
- **Feasible**
- **Language Independent**
- **Output**
- **Finiteness**
- **Effectiveness**

III.2. Advantages of Algorithms

- **Easy**: Algorithm is easy to understand.
- **Reusability**: Once developed, algorithms can be reused in different applications and scenarios, reducing the need to start from scratch each time.
- **Scalability**: Good algorithms can handle varying input sizes without a significant increase in resources.

III.3. Disadvantages of Algorithms

- **Complexity**: Designing and implementing algorithms can be complex, especially for intricate problems.
- **Branching and Looping** statements are difficult to show in Algorithms.

IV. Algorithm Analysis

The analysis is **a process of estimating the efficiency** of an algorithm and that is, trying to **know how good or how bad an algorithm could be**. There are two main parameters based on which we can analyze the algorithm:

- **Space complexity** can be understood as the amount of space required by an algorithm from its run to its completion
- **Time complexity** is a function of the input size $\boldsymbol{n}$ that indicates the amount of time required by the algorithm from its run to its completion.

IV.1. Execution Time / Runtime

- Usually, program execution time (program run-time) **is referred to as time complexity**, denoted by T_p (instance characteristics). This is the **sum of the execution times of all program instructions**.
- Accurately estimating the **execution time is a complex task**, since the **number of instructions executed depends on the input data**.
- What's more, the various instructions take varying amounts of time to execute. So, when estimating time complexity, **we only count the number of program steps.**

IV.1.1. Runtime Based on a Single Parameter

Let $\mathcal{A}$ be an algorithm and f the function such that $f(x)$ denotes the number of elementary operations performed by $\mathcal{A}$ on input x.

The execution time of $\mathcal{A}$ **in** the **worst case** is the function $T_{max} : \mathbb{N} \rightarrow \mathbb{N}$ such that:

$$T_{max}(n) = Max\{f(x)\text{: } input\ x\ of\ size\ n\}.$$

In other words, $T(n)$ indicates the largest number of operations performed among all inputs of size n.

IV.1.1. Runtime Based on a Single Parameter

Example:

The code below finds the sum of n numbers. **Determine the number of steps in this program.**

We consider the following operations to be elementary: **assignment**, **comparison**, **addition** and **access** to an element of a sequence.

IV.1.1. Runtime Based on a Single Parameter

Statement		Operations	Frequency	Total steps
float Sum(float a[], int n){	0	/	-	0
float s=0.0;	1	Assignment	1	1
int i=0;	1	Assignment	1	1
while(i<n){	1	comparison	n+1	n+1
s = s + a[i];	3	add, Ass, access	n	3n
i = i+1; }	2	add, Ass	n	2n
return s;}	0	/	-	0
			T(n)=	6n+3

Algorithm (function) works in **linear time**

IV.1.2. Runtime Based on Several Parameters

For some algorithms, the "size" of an input depends on several parameters, e.g., the number of rows $\boldsymbol{m}$ and columns $\boldsymbol{n}$ of a matrix; the number of elements $\boldsymbol{m}$ of a sequence and the number of bits $\boldsymbol{n}$ of its elements; the number of vertices $\boldsymbol{m}$ and edges $\boldsymbol{n}$ of a graph, and so on. For these algorithms, the notion of time is naturally extended:

$$\boldsymbol{T_{max}(n_1, \ldots, n_k) = Max\{f(x)\text{: } input\ x\ of\ size\ (n_1, \ldots, n_k)\}}.$$

IV.1.2. Runtime Based on Several Parameters

Example:

The code below **calculates the addition of two matrices** of size $\boldsymbol{m} \times \boldsymbol{n}$.

- **Determine the number of steps in this program.**
- We consider the following operations to be elementary: **assignment**, **comparison**, **addition** and **access** to an element of a matrix.

IV.1.2. Runtime Based on Several Parameters

Statement		Operations	Frequency	Total steps
void Add(Type a, b, m, n){	0	/	-	0
int i=0;	1	Assignment	1	1
while (i < m){	1	comparison	m+1	m+1
int j =0;	1	Assignment	m	m
while (i < n){	1	comparison	m(n+1)	m.n+m
c[i][j]=a[i][j]+b[i][j];	5	add, Ass, acc	m.n	5(m.n)
j = j+1;}	2	add, Ass	m.n	2(m.n)
i = i+1; }}	2	add, Ass	m	2m
Total			T(m, n)=	8mn+5m+2

IV.1.3. Types of Time Complexity Analysis

Example: Find a given value (if it exists!) in this array

0	1	2	3	4	5	6	7	8	9
15	25	3	7	33	28	7	9	10	11

15

OK

$$T(n) = 1 \;\Rightarrow\; \textbf{Best-case}$$

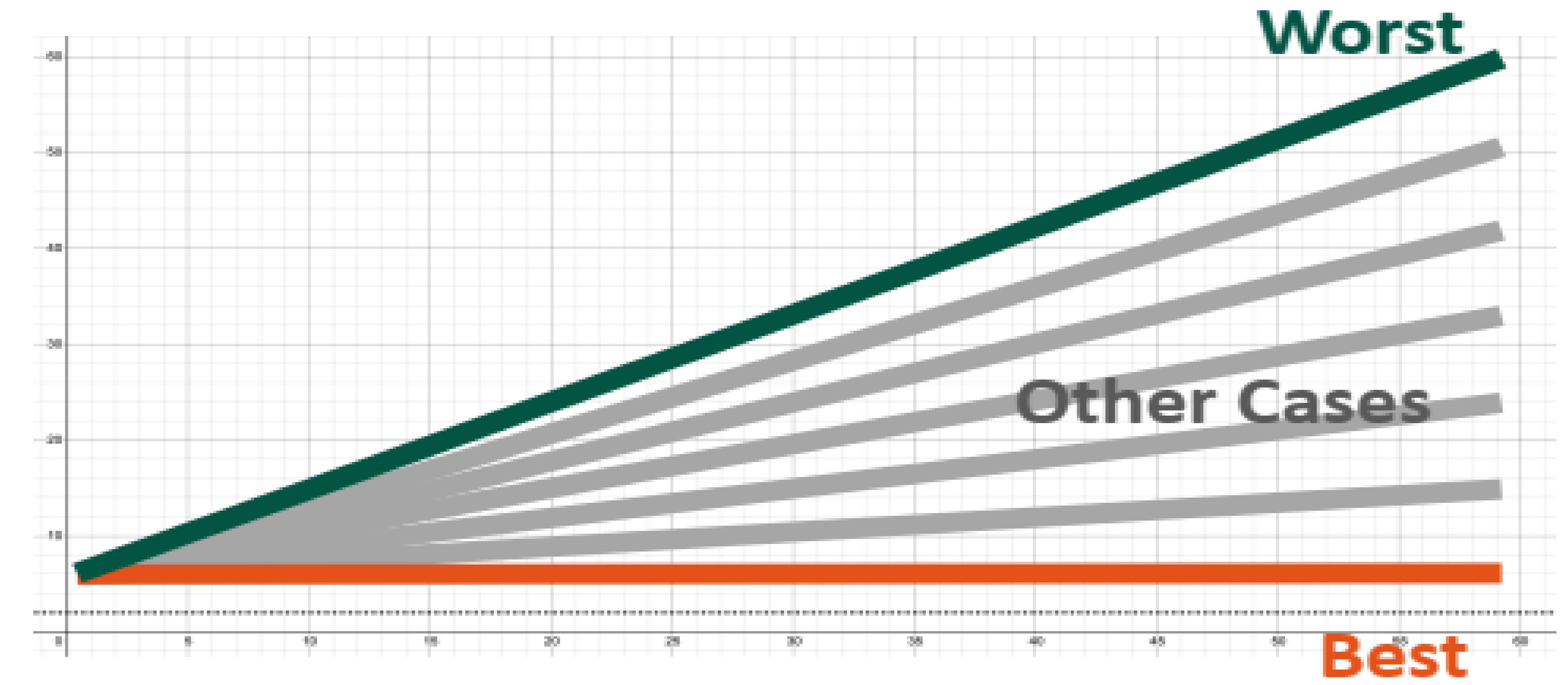
IV.1.3. Types of Time Complexity Analysis

Example: Find a given value (if it exists!) in this array

0	1	2	3	4	5	6	7	8	9
15	25	3	7	33	28	7	9	10	11
↑	↑	↑	↑	↑	↑	↑	↑	↑	↑
11	11	11	11	11	11	11	11	11	11

$$T(n) = 10 \;\; \Rightarrow \;\; \textbf{Worst-case}$$

IV.1.3. Types of Time Complexity Analysis



IV.1.3. Types of Time Complexity Analysis

We have three types of analysis related to time complexity:

- **Worst-case time complexity:** For '**n**' input size, the worst-case time complexity can be defined as the **maximum amount of time needed by an algorithm to complete its execution**. Thus, it is nothing but a function defined by the maximum number of steps performed on an instance having an input size of **n**. **Computer Scientists are more interested in this.**

  - **Big-O Notation**

IV.1.3. Types of Time Complexity Analysis

We have three types of analysis related to time complexity:

- **Average case time complexity:** For '**n**' input size, the average case time complexity can be defined as **the average amount of time needed by an algorithm to complete its execution**. Thus, it is nothing but a function defined by the average number of steps performed on an instance having an input size of **n**.

- **Big-Theta (Θ) Notation**

IV.1.3. Types of Time Complexity Analysis

We have three types of analysis related to time complexity:

- **Best-case time complexity:** For '**n**' input size, the best-case time complexity can be defined **as the minimum amount of time needed by an algorithm to complete its execution.** Thus, it is nothing but a function defined by the minimum number of steps performed on an instance having an input size of **n**.

  - **Big-Omega (Ω) Notation**

V. Algorithm Complexity

- The term algorithm complexity measures **how many steps are required by the algorithm to solve the given problem**.
- It evaluates the **order of count of operations executed** by an algorithm as a function of input data size.
- To assess the complexity, **the order (approximation) of the count of operations** is always considered instead of counting the exact steps.

V. Algorithm Complexity

- **O(f)** notation represents the complexity of an algorithm, which is also termed an **Asymptotic notation or "Big O" notation**.
- The complexity can be found in any form such as **constant, logarithmic, linear,** $\boldsymbol{n * log(n)}$ **, quadratic, cubic, exponential, factorial** etc.
- To make it even more precise, we often call the complexity of an algorithm "**running time**".

VI. Mathematical Notation

- Algorithms are widely used in various fields of study.
- We can solve different problems using the same algorithm. Consequently, **all algorithms must respect a standard**.
- **Mathematical notations use symbols or symbolic expressions that have a precise semantic meaning.**

VI.1. Asymptotic Notations

To select the best algorithm, it is necessary to check the efficiency of each algorithm. **The efficiency of each algorithm can be verified by calculating its time complexity.** Asymptotic notations are used to represent time complexity in an abbreviated form.

- **Big-O Notation :** Worst-case time complexity.
- **Big-Theta (Θ) Notation :** Average case time complexity.
- **Big-Ome** ity.

In this course, we'll focus on the **Big-O notation.**

VI.1.1. Big-O Notation

- **Big-O notation** is a mathematical notation used to describe the **upper bound or worst-case scenario of the time complexity or space complexity** of an algorithm in computer science.
- It provides a way to **classify algorithms** based on how their runtime or space requirements grow as the input size increases,
- Generally, it is represented as **$f(n) = O(g(n))$**. That means, at larger values of **n**, the upper bound of **$f(n)$** is **$g(n)$**.

VI.1.1. Big-O Notation

Definition I.1

We consider functions from $\mathbb{N}$ to $\mathbb{R}$ that are eventually positive, i.e., functions belonging to:

$$\boldsymbol{\mathcal{F}} = \{\mathbf{f}: \mathbb{N} \to \mathbb{R} : \exists\, \mathbf{m} \in \mathbb{N}, \forall \mathbf{n} \geq \mathbf{m}\ \ \mathbf{f(n)} > \mathbf{0}\}$$

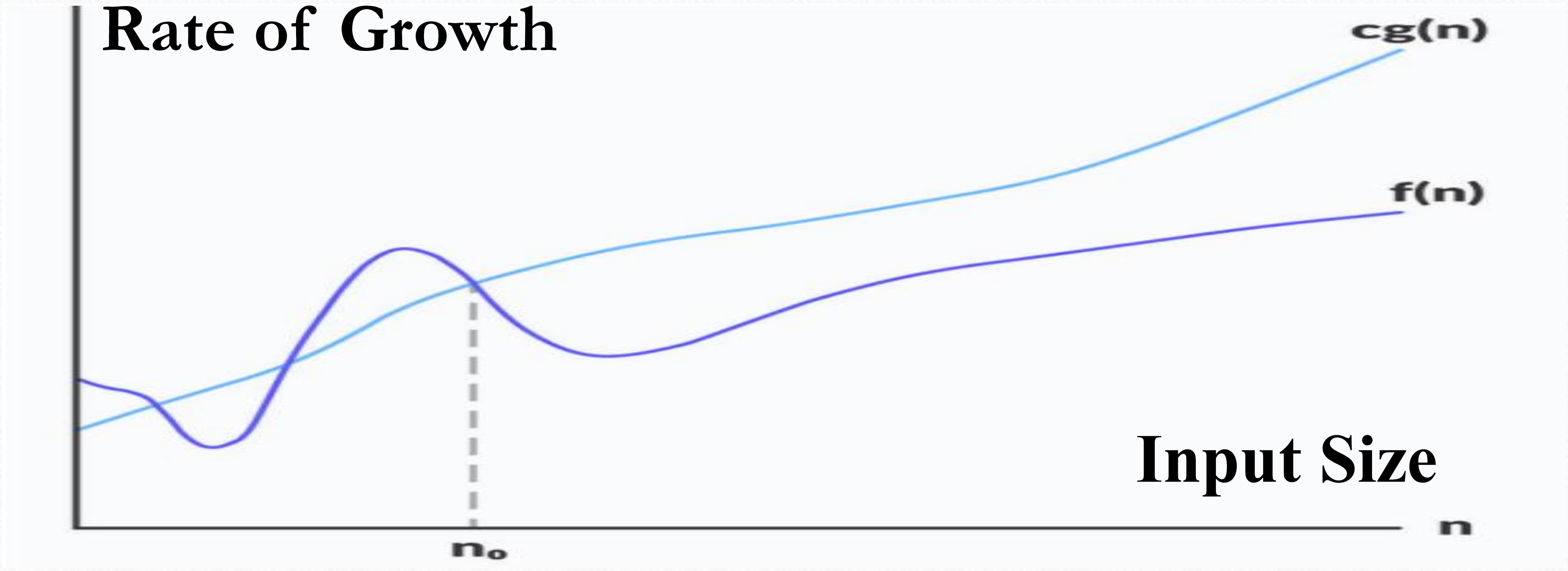
Let $\boldsymbol{g} \in \boldsymbol{\mathcal{F}}$. The set $\boldsymbol{\mathcal{O}}(\boldsymbol{g})$ is defined by:

$$\boldsymbol{\mathcal{O}}(\boldsymbol{g}) = \{\boldsymbol{f} \in \boldsymbol{\mathcal{F}}: \exists\, \boldsymbol{c} \in \mathbb{R}^{+}, \exists\, \boldsymbol{n_0} \in \mathbb{N}, \forall \boldsymbol{n} \geq \boldsymbol{n_0}\ \ \boldsymbol{f}(\boldsymbol{n}) \leq \boldsymbol{c}.\boldsymbol{g}(\boldsymbol{n})\}$$

- **Intuitively, $\mathbf{f} \in \boldsymbol{\mathcal{O}}(\mathbf{g})$ indicates that f increases as quickly or less quickly than the function g.**
- **We call the values c and $\mathbf{n_0}$ a multiplicative constant and a threshold.**

VI.1.1. Big-O Notation

- The graphical representation of **f(n) = O(g(n))** is shown in the figure, where the running time increases considerably when **n** increases.



VI.1.1. Big-O Notation

Example:

- Consider the function $\boldsymbol{f(n) = 5n^2 + 400n + 9}$
- The function $\boldsymbol{f}$ is greater than $\boldsymbol{g(n) = n^2}$
- However, we can show that $\boldsymbol{f \in \mathcal{O}(n^2)}$.

VI.1.1. Big-O Notation

Example:

We have : $\mathbf{f(n) = 5n^2 + 400n + 9}$

$\leq \mathbf{5}\boldsymbol{n}^2 + \mathbf{400}\boldsymbol{n} + \boldsymbol{n}^2$ for all $\boldsymbol{n} \geq \mathbf{3}$

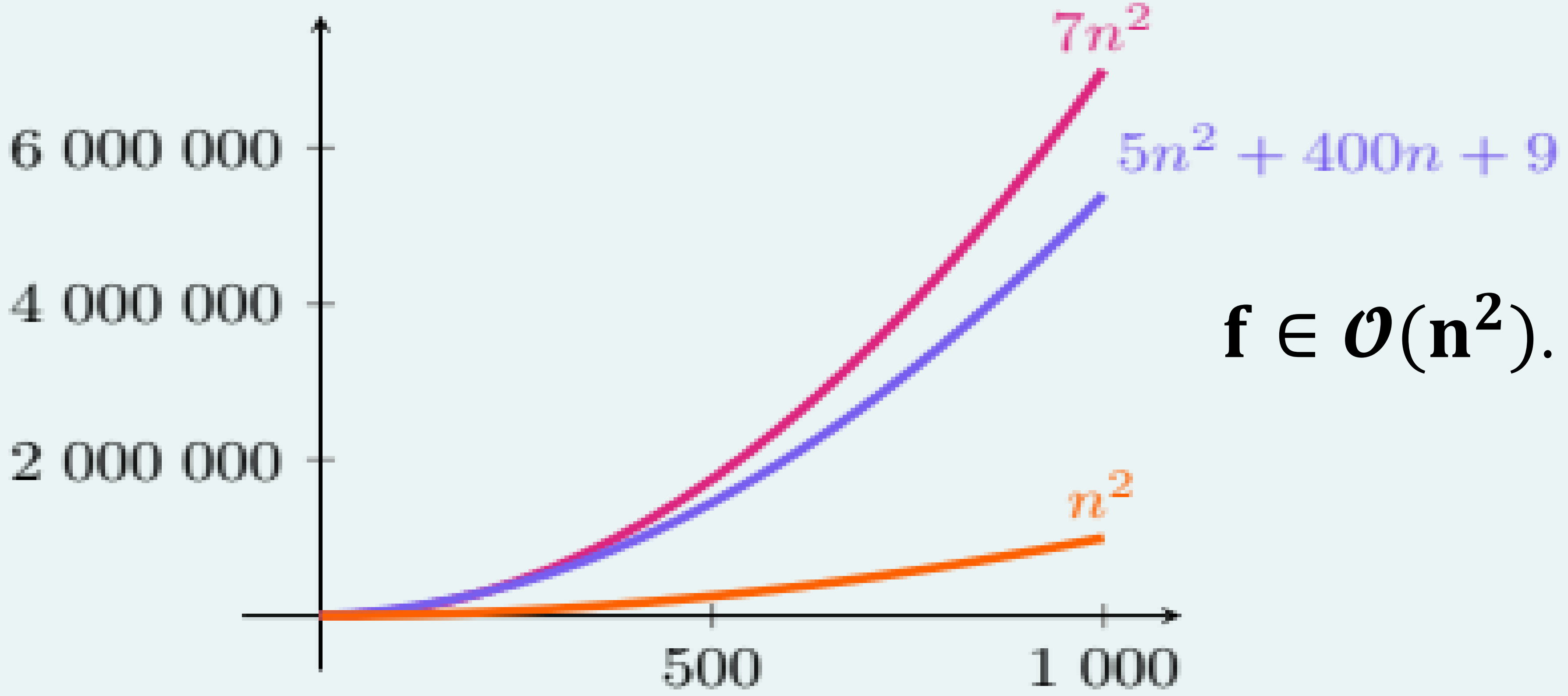
$\leq \mathbf{5}\boldsymbol{n}^2 + \boldsymbol{n.n} + \boldsymbol{n}^2$ for all $\boldsymbol{n} \geq \mathbf{400}$

$= \mathbf{7}\boldsymbol{n}^2$

- Thus, taking $\mathbf{c = 7}$ as a multiplicative constant and $\boldsymbol{n_0} = \mathbf{400}$ as a threshold, we conclude that $\boldsymbol{f} \in \boldsymbol{O}(\boldsymbol{n}^2)$.

VI.1.1. Big-O Notation

Example:



$$\mathbf{f} \in \mathcal{O}(\mathbf{n}^2).$$

VI.1.2. Big-O Notation for Complexity Class

The most common complexity classes (in increasing order) are the following:

- $\mathcal{O}(1)$
- $\mathcal{O}(log_2 log_2 n)$
- $\mathcal{O}(log_2 n)$
- $\mathcal{O}(n)$
- $\mathcal{O}(n log_2 n)$
- $\mathcal{O}(n^2)$
- $\mathcal{O}(n^3)$
- $\mathcal{O}(2^n)$
- $\mathcal{O}(n!)$
- $\mathcal{O}(n^n)$

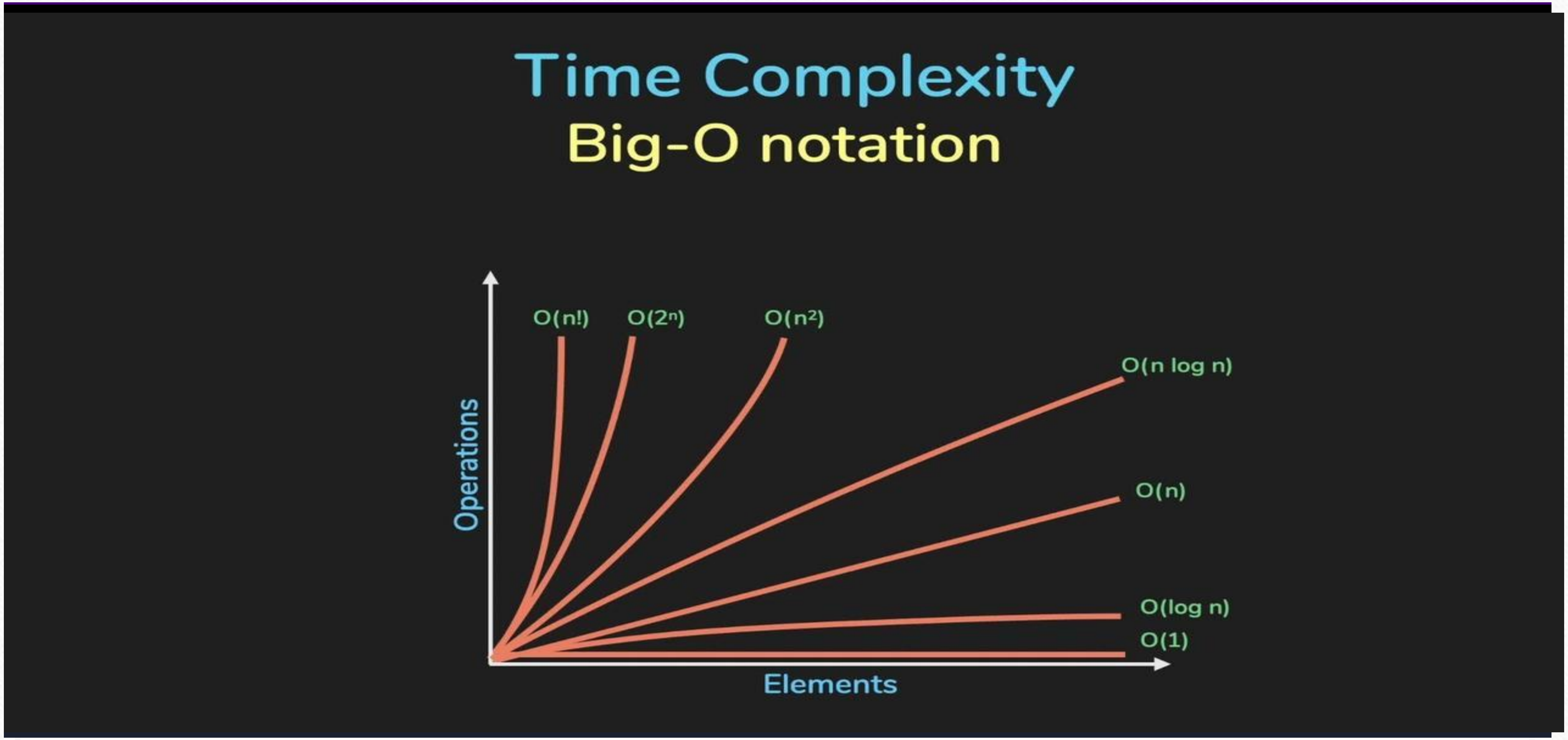
VI.1.2. Big-O Notation for Complexity Class

$f(n)$	$n = 4$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1	1	1	1.00×10^0	1.00×10^0
$log_2 log_2 n$	1	2	3	3.32×10^0	4.32×10^0
$log_2 n$	2	4	8	1.00×10^1	2.00×10^1
n	4	16	2.56×10^2	1.02×10^3	1.05×10^6
$n log_2 n$	8	64	2.05×10^3	1.02×10^4	2.10×10^7
n^2	16	256	6.55×10^4	1.05×10^6	1.10×10^{12}
n^3	64	4096	1.68×10^7	1.07×10^9	1.15×10^{18}
2^n	16	65536	1.16×10^{77}	1.80×10^{308}	6.74×10^{315652}

VI.1.2. Big-O Notation for Complexity Class

$f(n)$	$n = 4$	$n = 16$	$n = 256$	$n = 1024$	$n = 1048576$
1	1 μsec	1 μsec	1 μsec	1 μsec	1 μsec
$log_2 log_2 n$	1 μsec	2 μsec	3 μsec	3.32 μsec	4.32 μsec
$log_2 n$	2 μsec	4 μsec	8 μsec	10 μsec	20 μsec
n	4 μsec	16 μsec	2.56 μsec	1.02 msec	1.05 sec
$n log_2 n$	8 μsec	64 μsec	2.05 msec	10.2 msec	21 sec
n^2	16 μsec	256 μsec	65.5 msec	1.05 sec	1.8 wk
n^3	64 μsec	4.1 msec	16.8 sec	17.9 min	36.559 yr
2^n	16 μsec	65.5 msec	3.7×10^{63} yr	5.7×10^{294} yr	2.1×10^{315639} yr

VI.1.2. Big-O Notation for Complexity Class



VI.1.3. Properties of Asymptotic Notations

a) General (Constant Factor)

If $\boldsymbol{f(n) = O\big(g(n)\big)}$ $\boldsymbol{then\ a \times f(n) = O\big(g(n)\big)}$

e.g.,

$$\boldsymbol{f(n) = 2n^2 + 5}\ is\ \boldsymbol{O\big(n^2\big)},$$

$$\boldsymbol{then}$$

$$\boldsymbol{4 \times f(n) \quad = 8n^2 + 20}\ is\ also\ \boldsymbol{O\big(n^2\big)}$$

VI.1.3. Properties of Asymptotic Notations

b) Reflexive

Given $\boldsymbol{f(n)}, \boldsymbol{then}\ \boldsymbol{f(n) = O\big(f(n)\big)}$, since maximum value of $\boldsymbol{f(n)}$ will be $\boldsymbol{f(n)}$ itself, i.e., every function is an upper-bound of itself.

e.g.,

$$\boldsymbol{f(n) = n^2 = O\left(n^2\right)}$$

VI.1.3. Properties of Asymptotic Notations

c) Transitive

$$if\ f(n) = O(g(n))\ and\ g(n) = O(h(n)),$$

$$then$$

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

e.g., $f(n) = n,\ g(n) = n^2, and\ h(n) = n^3$

here, $f(n) = O(n^2)\ \&\ g(n) = O(n^3)$

$$\Longrightarrow f(n) = O(n^3)\ or\ O(h(n))$$

VI.1.4. Limitations of Big O Notation

- Many algorithms are **too hard to analyze mathematically**.
- Big-O **analysis only tells us how the algorithm grows with the size of the problem**, not how efficient it is, because it does not consider the programming effort.
- **It does not take into account important constants**. For example, if one algorithm takes $\boldsymbol{O(n^2)}$ time to execute and the other takes $\boldsymbol{O(100000n^2)}$ time to execute, then according to Big O, both algorithms have the same time complexity. In real-time systems, this is an important consideration.

VII. Running-Time Calculations

- There are **several ways to estimate the running time** of a program. To simplify the analysis, we will adopt the convention that there are **no particular units of time**.
- We are **essentially doing is computing** a Big-Oh running time. Since Big-Oh is an upper bound, we must be careful never to underestimate the running time of the program.
- In effect, the answer provided is a guarantee that **the program will terminate within a certain period**.

VII.1. Assumptions in Complexity Analysis

- **Uniform Cost Model:** Assumes that all basic instruction (such as arithmetic, comparisons, assignments, . . .) have the same cost. This simplifies the analysis by treating each operation as taking constant time, represented by the notation $\boldsymbol{O(1)}$.
- **Input size:** Analysis is generally performed based on the input size, often referred to as "n", which represents the number of elements or the size of the input data structure.

VII.1. Assumptions in Complexity Analysis

- **Worst-case scenario:** Complexity analysis often focuses on the worst-case input, i.e., assuming the input that leads to the maximum number of operations.
- **Loops:** each iteration of a loop adds the complexity of what is done in the loop body.
- **Functions (procedures):** each function call adds the complexity of that function to the total complexity.

VII.2. Simplification Rules

- **Drop Constants:** Constants in front of the dominant term are dropped in Big-O notation. For example: $\boldsymbol{O(5n^3+4)}$ simplifies to $\boldsymbol{O(n^3)}$.
- **Dominant Term Rule:** only the term with the largest growth rate dominates in Big-O notation. For example, in $\boldsymbol{O(n^2+n)}$, a dominant term is $\boldsymbol{n^2}$, so the complexity is $\boldsymbol{O(n^2)}$.
- **Additive Rule:** When analyzing multiple operations in sequence, the complexities are added. For example, if an algorithm has two separate loops with complexities $\boldsymbol{O(n^3)}$ and $\boldsymbol{O(n)}$, the total complexity is $\boldsymbol{O(n^3)+O(n)=O(n^3+n)=O(n^3)}$.

VII.2. Simplification Rules

- **Multiplicative Rule:** When analyzing nested operations, the complexities are multiplied. For example, if an algorithm has nested loops with complexities $\mathcal{O}(n)$ and $\mathcal{O}(n)$, the total complexity is $\mathcal{O}(n) * \mathcal{O}(n) = \mathcal{O}(n^2)$.
- **Logarithmic Rules:** Base of logarithms can be ignored in Big-O notation, as they represent constant factors. Therefore, $\mathcal{O}(log\ n)$ and $\mathcal{O}(log_2\ n)$ are considered the same.

VII.3. How to Calculate the Approximate Time Taken by the Algorithm?

First, we need to understand the types of algorithms available to us. There are two types of algorithms:

- **Iterative algorithm:** In the iterative approach, the function executes repeatedly until the condition is met or it fails. It involves the construction of a loop.
- **Recursive Algorithm:** The function calls itself until the condition is met in the recursive approach. It integrates the branching structure.

VII.4. Guidelines for Asymptotic Analysis of Iterative Algorithms

There are some general rules to help us determine the running time of iterative algorithms

- **Simple Statement and $\boldsymbol{O(1)}$**
- **Sequence of Simple Statements**
- **Sequence of Statements**
- **Selection**
- **Iterations (Loops)**

VII.4.1. Simple Statement and O(1)

- We define a simple statement as one that **does not have any control flow component** (subprogram call, loop, selection, etc.) in it.
- An **assignment** is a typical example of a simple statement.
- Time taken by a simple statement is considered to be **constant**.

E.g.,

`int x = 1;` ⟵ 1 Operation (assig) $\Longrightarrow \boldsymbol{k}\ units\ of\ time$

VII.4.1. Simple Statement and O(1)

- Let us assume that a given simple statement takes $\boldsymbol{k}$ units of time. If we factor out the constant, we would be left with **1**, yielding $\boldsymbol{k = O(1)}$. That is, a constant amount of time is denoted by $\boldsymbol{O}(1)$. It may be noted that we are not concerned with the value of the constant; as long as it is constant, we will have $\boldsymbol{O}(1)$.

E.g.,

`int x = 1;` ⟵ 1 Operation $\Longrightarrow$ **k** units of time $\Longrightarrow \boldsymbol{O(1)}$

`x=x+1;` ⟵ 2 Operations $\Longrightarrow$ **g** units of time $\Longrightarrow \boldsymbol{O(1)}$

VII.4.2. Sequence of Simple Statements

- As stated above, a simple statement takes a constant amount of time. Therefore, time taken by a sequence of such statements will simply be the sum of time taken by individual statements, which will again be a constant. Therefore, the time complexity of a sequence of simple statements will also be $\boldsymbol{\mathcal{O}(1)}$.

E.g.,

$$\left.\begin{array}{ll}\texttt{\textbf{int} x = 1;} & \longleftarrow \boldsymbol{\mathcal{O}(1)} \\ \texttt{x=x+1;} & \longleftarrow \boldsymbol{\mathcal{O}(1)}\end{array}\right\} \boldsymbol{\mathcal{O}(1)}$$

VII.4.3. Sequence of Statements

- Time taken by a sequence of statements is the **sum of time taken** by individual statements which is the **maximum** of these complexities.

E.g.,

```
statement1 // O(n^2)
statement2 // O(1)
statement3 // O(n)
statement4 // O(n)
```

$$\mathcal{O}(n^2) + \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n)$$
$$=$$
$$Max\left(\mathcal{O}(n^2) + \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n)\right)$$
$$=$$
$$\mathcal{O}(n^2)$$

VII.4.4. Selection

A selection can have many branches and can be implemented with the help of ***if*** or ***switch*** statement. In order to determine the time complexity of an ***if*** or ***switch*** statement,

- we first independently **determine the time complexity of each one of the branches separately**.
- Since, in a selection, each branch is **mutually exclusive**, the worst-case scenario would be the case of the branch which **required the largest amount of computing resources**.

VII.4.4. Selection

Example:

```
if (cond1) statement1                     // O(n^2)
else if (cond2) statement2                // O(1)
    else if (cond3) statement3            // O(n)
        else statement4                   // O(n log n)
```

MAX

The time complexity of this code segment will be:

$$Max\Big(\mathcal{O}(n^2)\ +\ \mathcal{O}(1) + \mathcal{O}(n) + \mathcal{O}(n\, log\, n)\Big) = \mathcal{O}(n^2)$$

VII.4.5. Iterations (Loops)

- In C (C++) language $\boldsymbol{for}$, $\boldsymbol{while}$, and $\boldsymbol{do - while}$ statements are used to implement an iterative step or loop.
- Complexity analysis of iterative statements is usually **the most difficult of all the statements**.
- The main task involved in the analysis of an iterative statement is **the estimation of the number of iterations** of the body of the loop.
- There are many different scenarios that need separate attention and are discussed below.

VII.4.5. Iterations (Loops)

- There are many different scenarios that need separate attention and are discussed below.
- **Simple Loops**
- **Loops with Uniform Step Size**
- **Simple Loops with Variable Step Size**
- **A Deceptive Case**
- **Nested Loop**
  - **Independent Loops**
  - **Dependent Loops**

VII.4.5.1. Simple Loops

We define a simple loop as a loop which **does not contain another loop inside its body** (that is no nested loops). Analysis of a simple loop involves the following steps:

1. Determine the complexity of **the code written in the loop body** as a function of the problem size. Let it be $\mathcal{O}(f(n))$.
2. Determine the **number of iterations of the loop** as a function of the problem size. Let it be $\mathcal{O}(g(n))$.
3. Then the complexity of the loop would be: $\mathcal{O}(f(n)) . \mathcal{O}(f(n)) = \mathcal{O}(f(n) . g(n))$

VII.4.5.1. Simple Loops

Example:

```
for( i = 1; i <= n; i ++)
    S = S + 2;
```

$$\left.\begin{array}{l}\texttt{for(i = 1; i <= n; i ++)} \\ \quad \texttt{S = S + 2;} \quad \text{// constants time, } \boldsymbol{O}(\mathbf{1})\end{array}\right\} \textbf{number of iterations is n, } \boldsymbol{O}(\mathbf{n})$$

Total time = $\boldsymbol{O}(\mathbf{1}) \times \boldsymbol{O}(\boldsymbol{n}) = \boldsymbol{O}(\mathbf{1} \times \boldsymbol{n}) = \boldsymbol{O}(\boldsymbol{n})$.

VII.4.5.2. Loops with Uniform Step Size

- As stated above, we define a simple loop with uniform step size as **the simple loop where the loop control variable is incremented/decremented by a fixed amount**.
- That is, in this case, **the values of the loop control variable follow an algebraic sequence.** These are perhaps the most commonly occurring loops in our programs.
- These are **usually very simple** and have a time complexity of $O(n)$. They are also very **simple to analyze**.

VII.4.5.2. Loops with Uniform Step Size

Example: The following code for Linear Search elaborates this in more detail.

```
index = -1;                              // 𝒪(1)
for (int i = 0; i < N; i++)              // 𝒪(1)
    if (a[i] == key) {                   // 𝒪(1)
        index = i;                       // 𝒪(1)
        break;                           // 𝒪(1)
    } //for
```

number of iterations is n, $\mathcal{O}(\mathbf{n})$

- Total time = $\mathcal{O}(1) \times \mathcal{O}(n) = \mathcal{O}(1 \times n) = \mathcal{O}(n)$.

VII.4.5.2. Loops with Uniform Step Size

We can easily see that:

- Time complexity of the code in the body of the loop is $\boldsymbol{O(1)}$
- In the worst case (when key is not found), there will be n iterations of the for loop, resulting in $\boldsymbol{O(n)}$.
- Therefore, the time complexity of the Linear Search algorithms is: $\boldsymbol{O(1)} \times \boldsymbol{O(n)} = \boldsymbol{O(n)}$.

VII.4.5.3. Simple Loops with Variable Step Size

- In loops with variable step size, the loop control variable is increased or decreased by a variable amount.
- In such cases, **the loop control variable is usually multiplied or divided by a fixed amount**, thus it usually follows a **geometric sequence**.

VII.4.5.3. Simple Loops with Variable Step Size

Example: The following code for Linear Search elaborates this in more detail.

```
high = N-1;
low = 0;
index = -1;
while(high >= low)
{
    mid = (high + low)/2;
    if (key == a[mid]) { index = mid; break;}
        else if (key > a[mid]) low = mid + 1;
             else high = mid - 1;
}
```

$\mathcal{O}(1)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

$\mathcal{O}(1)$

number of iterations ???

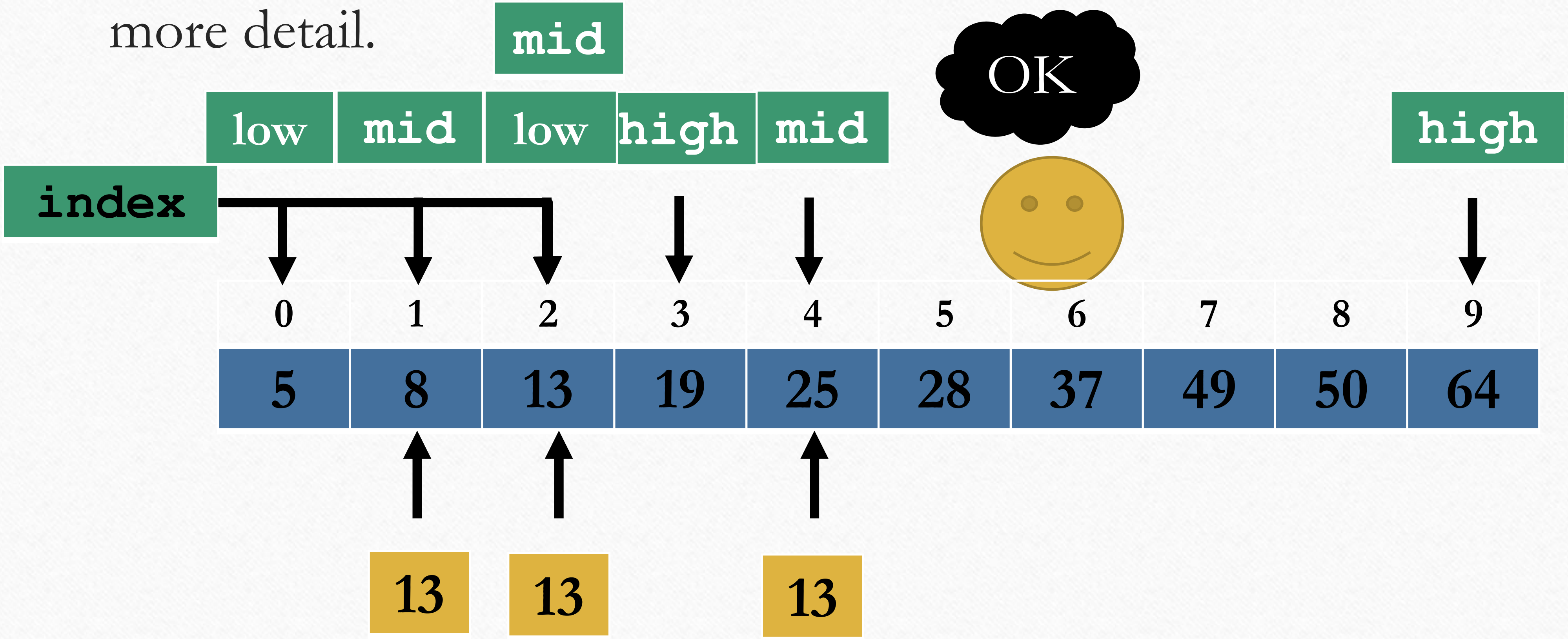
VII.4.5.3. Simple Loops with Variable Step Size

Example: The following code for Linear Search elaborates this in more detail.

- Once again, we can easily see that the **code written in the body of the loop** has a complexity of $\boldsymbol{O(1)}$.
- **We now need to count the number of iterations.**

VII.4.5.3. Simple Loops with Variable Step Size

Example: The following code for Linear Search elaborates this in more detail.



VII.4.5.3. Simple Loops with Variable Step Size

Example: The following code for Linear Search elaborates this in more detail.

- Once again, we can easily see that the **code written in the body of the loop** has a complexity of $\boldsymbol{O(1)}$.
- **We now need to count the number of iterations.**

Iteration	1	2	3	…	…	K+1
Search Size	$N = 2^k$	$\frac{N}{2} = 2^{k-1}$	$\frac{N}{4} = 2^{k-2}$	…	…	$1 = 2^{k-k} = 2^0$

VII.4.5.3. Simple Loops with Variable Step Size

Example: The following code for Linear Search elaborates this in more detail.

- That is, after **k+1** steps the loop will terminate.
- Since $\boldsymbol{N = 2^k}$, by taking log base **2** of both sides we get $\boldsymbol{k = log_2 N}$.
- In general, when **N** cannot be written as power of **2**, the number of iterations will be given by the ceiling function $\boldsymbol{log_2 N}$.
- So, in the worst case, the number of iterations will be given by $\boldsymbol{\mathcal{O}([log_2\, N])}$.

$$\boldsymbol{Total\ time\ T(n) = \mathcal{O}(1) \times \mathcal{O}(log\, n) = \mathcal{O}(log\, n)}$$

VII.4.5.4. A Deceptive Case

Before concluding this discussion on simple loops, let us look at a last, but quite deceiving

Example: The following piece of code prints the table for number **m**.

```
for(int i=1; i<=10; i++)
    printf("%d X %d = %d\n", m, i, m*i);
```

$\mathcal{O}(1)$

$\mathcal{O}(1)$

number of iterations is 10, $\mathcal{O}(1)$

$$Total\ time\ T(n) = \mathcal{O}(1) \times \mathcal{O}(1) = \mathcal{O}(1)$$

VII.4.5.5. Nested Loop

For the purpose of this discussion, we can divide the nested loops in two categories:

1. **Independent loops:** the number of iterations of the inner loop are independent of any variable controlled in the outer loop.
2. **Dependent loops:** the number of iterations of the inner loop are dependent upon some variable controlled in the outer loop.

VII.4.5.5. Nested Loop (Independent Loops)

Let us consider the following code segment written to add two matrices **a**, and **b** and store the result in matrix **c**.

```
for (int i = 0; i < ROWS; i++)
    for (int j = 0; j < COLS; j++)
        c[i][j] = a[i][j] + b[i][j];
```

$\leftarrow \mathcal{O}(1)$

Nbr_It = CLOS

Nbr_It = ROWS

- Total Number of iterations is $\boldsymbol{ROWS \times COLS}$
- The time complexity of the algorithms is $\boldsymbol{\mathcal{O}(ROWS \times COLS)}$.

VII.4.5.5. Nested Loop (Independent Loops)

- In general, in the case of **independent nested** loops, all we have to do is to determine the time complexity of each one of these loops independent of the other and then multiply them to get the overall time complexity.
- That is, if we have two independent nested loops with time complexity of $\mathcal{O}(\mathbf{x})$ and $\mathcal{O}(\mathbf{y})$, the overall time complexity of the algorithm will be $\mathcal{O}(\mathbf{x}) \times \mathcal{O}(\mathbf{y}) = \mathcal{O}(\mathbf{x} \times \mathbf{y})$.

VII.4.5.5. Nested Loop (Dependent Loops)

- When the number of iterations of the **inner loop is dependent upon some variable controlled in outer loop**, we need to do a little bit more to find out the time complexity of the algorithm.
- What we need to do is **to determine the number of times the body of the inner loop will execute**.

VII.4.5.5. Nested Loop (Dependent Loops)

Example 1:

```
for (i = 0; i < N ; i++) {
      min = i;
      for (j = i; j < N; j++)
           if (a[min] > a[j]) min = j;
      swap(a[i], a[min]);
}
```

- As can be clearly seen, the value of **j** is **dependent upon i**, and hence the number of iterations of the inner loop depend upon the value of **i** which is controlled in the outer loop.

VII.4.5.5. Nested Loop (Dependent Loops)

Example 1:

Value of i	0	1	2	…	I	…	N-1
Number of iterations of the inner loop	N	$N-1$	$N-2$	…	$N-i$	…	1
Total Number of times the body of the inner loop is executed	$T(N) = N + (N-1) + (N-2) + \cdots + (N-i) + \cdots + 1$ $= \frac{N(N+1)}{2}$ $= \mathcal{O}(N^2)$						

VII.4.5.5. Nested Loop (Dependent Loops)

Example 2:

```
for (i = 1; i < N; i++)
    for (j = 1; j <= i; j = j * 2)
        k++;
```

- The first thing to note here is that the **inner loop follows a geometric progression**, going from **1** to **i**, with **2** as the common ratio between consecutive terms.
- We have already seen that in such cases the number of iterations is given by $[\boldsymbol{log_2}\,(\boldsymbol{i}+\mathbf{1})]$

VII.4.5.5. Nested Loop (Dependent Loops)

Example 2:

Value of i	1	2	3	4	…	N-1
Number of iterations of the inner loop	$1 = [\log(2)]$	$2 = [\log(3)]$	$2 = [\log(4)]$	$3 = [\log(5)]$	…	$[\log(N)]$
Total Number of times the body of the inner loop is executed	$T(N) = \log 2 + \log 3 + \log 4 + \log 5 + \cdots + \log N$ $= \log(2 \times 3 \times 4 \times 5 \times \cdots \times N) = \log(N!)$ $= \boldsymbol{O(N \log N)}$ *by Stirling's formula*					

VII.4.5.5. Nested Loop (Dependent Loops)

Example 1:

$\mathcal{O}(n) \times \cancel{\mathcal{O}(n)} = \mathcal{O}(n^2)$

```
for (i = 0; i < N ; i++) {
    min = i;
    for (j = i; j < N; j++)
        if (a[min] > a[j]) min = j;
    swap(a[i], a[min]);
}
```

$\mathcal{O}(n)$

$\mathcal{O}(n)$

Example 2:

$\mathcal{O}(\log n) \times \cancel{\mathcal{O}(n)} = \mathcal{O}(n \log n)$

```
for (i = 1; i < N; i++)
    for (j = 1; j <= i; j = j * 2)
        k++;
```

$\mathcal{O}(\log n)$

$\mathcal{O}(n)$

VII.4.5.5. Nested Loop (Dependent Loops)

$$\mathcal{O}(n) \times \mathcal{O}(\log n) = \mathcal{O}(n \log n)$$

Example 3:

```
for (i = 1; i < N; i = i*2)
    for (j = 1; j < i; j++)
        k++;
```

$\mathcal{O}(n)$

$\mathcal{O}(\log n)$

- Now, in this case, if we calculated the complexity of each loop independently and then multiplied the results, we would get the time complexity as $\boldsymbol{O(N \log N)}$.

- This is very different from what we get if we apply the approach used in the previous examples.

VII.4.5.5. Nested Loop (Dependent Loops)

Example 3:

Value of i	1	2	3	4	…	N-1
Number of iterations of the inner loop	$1 = 2^0$	$2 = 2^1$	$4 = 2^2$	$8 = 2^3$	$\ldots$	$N \approx 2^k$ $Where$ $k = [logN]$
Total Number of times the body of the inner loop is executed	$T(N) = 2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^k$ $= 2^{k+1} - 1 = 2 \times 2^k - 1$ $= \boldsymbol{\mathcal{O}\left(2^k\right) = \mathcal{O}(N)}$					

VII.4.6. Non-Recursive Subprogram Call

- A non-recursive subprogram call is simply a statement whose complexity is **determined by the complexity of the subprogram.**
- Therefore, we simply **determine the complexity of the subprogram separately** and then use that value in our complexity derivations. The rest is as usual.

VII.4.6. Non-Recursive Subprogram Call

Example 1:

```
float sum(float a[], int size) // assuming size > 0
{
    float temp = 0;
    for (int i = 0; i < size; i++)
        temp = temp + a[i];
    return temp;
}
```

$\mathcal{O}(1)$

$\mathcal{O}(n)$

$\mathcal{O}(1)$

$$\mathcal{O}(1 \times n \times 1) = \mathcal{O}(n)$$

- ➢ It is easy to see that this function has a linear time complexity, that is, $\boldsymbol{O(N)}$.

VII.4.6. Non-Recursive Subprogram Call

Example 2:

```
float average(float data[], int size) // assuming size > 0
{
    float total, mean;
    total = sum(data, size);
    mean = total/size;
    return mean;
}
```

$$\left.\begin{array}{l}\mathcal{O}(1)\\ \mathcal{O}(n)\\ \mathcal{O}(1)\\ \mathcal{O}(1)\end{array}\right\} \mathcal{O}(n)$$

- It is easy to see that this function has a linear time complexity, that is, $O(N)$.

VII.4.6. Non-Recursive Subprogram Call

Example 3:

```
int foo(int n) {
    int count = 0;
    for (int j = 0; j < n; j++)
        count++;
    return count;
}
```

$\mathcal{O}(1)$

$\left.\begin{array}{l}\mathcal{O}(1)\\ \mathcal{O}(1)\end{array}\right\} \mathcal{O}(n)$

$\mathcal{O}(1)$

$\mathcal{O}(n)$

- It is easy to see that this function has a linear time complexity, that is, $O(N)$.

VII.4.6. Non-Recursive Subprogram Call

Example 4:

`int bar(int n) {`	
`    temp = 0;`	$\mathcal{O}(1)$
`    for (int i = 1; i < n; i = i * 2) {`	$\mathcal{O}(1)$
`        temp1 = foo(i);`	$\mathcal{O}(n)$
`        temp = temp1 + temp;`	$\mathcal{O}(1)$
`    }`	
`}`	

$\mathcal{O}(\log n)$

$\mathcal{O}(n \log n)$

➢ It is easy to see that this function has a linear time complexity, that is, $\mathcal{O}(n \log n)$.

VII.4.6. Non-Recursive Subprogram Call

Example 5:

```
int factorial(int n){ //Returns a factorial of the natural n.
    int fact = 1;          O(1)
    int i = 2;             O(1)
    while (i <= n){        O(1)
        fact = fact * i;   O(1)          O(n)      O(n)
        i = i + 1;         O(1)
    }
    return fact;           O(1)
}
```

VII.5. Guidelines for Asymptotic Analysis of Recursive Algorithms

- Determining the running time of a function that calls itself recursively **requires more work than analyzing** iterative (non-recursive) functions.
- A useful technique is to describe the execution time using a **recurrence relation.**

VII.5. Guidelines for Asymptotic Analysis of Recursive Algorithms

A recurrence relation, also known as a **difference equation**, is an equation that defines a sequence recursively: **each term in the sequence is defined as a function of the preceding terms.**

VII.5. Guidelines for Asymptotic Analysis of Recursive Algorithms

To analyze the time complexity of a recursive function, you can follow these steps:

- **Determine the recurrence relation:** Identify the recursive calls and their respective inputs. Write an equation that expresses the time complexity of the function in terms of its inputs.
- **Solve the recurrence relation:** Solve the equation to get a closed-form solution for the time complexity.
- **Analyze the solution:** Determine the dominant term(s) in the closed-form solution to get the time complexity of the function.

VII.5.1. Recurrence Relation

- In the following sections, we will discuss methods that can be used to find the time complexities of recursive algorithms.
- Previously, we defined the runtime of an algorithm as a function $\boldsymbol{T(n)}$ of its input size $\boldsymbol{n}$. We will still do this when dealing with recursive functions.
- However, because a recursive algorithm calls itself with a smaller input size, we will write $\boldsymbol{T(n)}$ as a recurrence relation, or an equation that defines the runtime of a problem in terms of the runtime of a recursive call on smaller input.

VII.5.1. Recurrence Relation

Example:

Compute the factorial function $\boldsymbol{F(n) = n}$ for an arbitrary non-negative integer $\boldsymbol{n}$. Since

$$\boldsymbol{n! = 1 \times 2 \times \cdots \times (n-1) \times n = n(n-1)! \qquad for\, n \geq 1}$$

and $\boldsymbol{0! = 1}$by definition, we can compute $\boldsymbol{F(n) = F(n-1).n}$ with the following recursive algorithm.

VII.5.1. Recurrence Relation

Example:

```
    int Fact(int n) {
(1)   if (n==0);
(2)     return 1;              /* basis */
      else
(3)     return n*Fact(n-1); /*induction */
    }
```

VII.5.1. Recurrence Relation

Example:

```
    int Fact(int n) {
(1)   if (n==0);
(2)      return 1;
      else
(3)      return n*Fact(n-1);
    }
```

$O(1)$

a

$b + T(n-1)$

$O(1) + T(n-1)$

VII.5.1. Recurrence Relation

Example:

We can thus define $\boldsymbol{T(n)}$ by the following recurrence relation:

$$\boldsymbol{T(n) = \begin{cases} a & if\ n = 0 \\ T(n-1) + b & if\ n > 0 \end{cases}}$$

$$\boldsymbol{OR}$$

$$\boldsymbol{T(n) = \begin{cases} \mathcal{O}(1) & if\ n = 0 \\ T(n-1) + \mathcal{O}(1) & if\ n > 0 \end{cases}}$$

VII.5.2. Solving Recurrence Relations

- After we convert a recursive algorithm into a recurrence relation, **we can use that recurrence relation to determine its time complexity.**
- There are many techniques for solving recurrence relations. In this section, we will examine three methods:
  - **The iterative substitution method,**
  - **The recurrence tree method,**
  - **The master theorem**

1) Iterative Substitution Method

After we have a recurrence relation, the steps of the iterative substitution method (also known as the **iteration method**) are as follows:

- **Step 1:** Write out the recursive terms $(\boldsymbol{T(n-1), T(n-2), \ldots, etc.})$, as their own recurrence relations and substitute their equations into the original $\boldsymbol{T(n)}$ formula at each step.
- **Step 2:** Look for a pattern that describes $\boldsymbol{T(n)}$ at the $\boldsymbol{k^{th}}$ step (for any arbitrary $\boldsymbol{k}$), and express it using a summation formula.

1) Iterative Substitution Method

After we have a recurrence relation, the steps of the iterative substitution method (also known as the **iteration method**) are as follows:

- ❑ **Step 3:** Solve for $\boldsymbol{k}$ such that the base case is the only recursive term that is present on the right-hand side of the equation for $\boldsymbol{T(n)}$. Determine the closed form solution by replacing instances of the base case with its value (e.g., replacing $\boldsymbol{T(1)}$ with $\mathbf{1}$ if the base case is $\boldsymbol{T(n) = O(1)}$).

1) Iterative Substitution Method

Example 1:

Consider the Recurrence: $\boldsymbol{T(n) = \begin{cases} \mathcal{O}(1) & if\ n = 0 \\ T(n-1) + \mathcal{O}(1) & if\ n > 0 \end{cases}}$

- **Solve the following recurrence relation using the iterative substitution method ?**

1) Iterative Substitution Method

Example 1 (Solution):

From now on, we will substitute $\boldsymbol{O(1)}$ with the constant $\mathbf{1}$. This simplifies our math without changing the result of our analysis.

$$\boldsymbol{T(n)} = \begin{cases} \mathbf{1} & \boldsymbol{if\ n = 0} \\ \boldsymbol{T(n-1) + 1} & \boldsymbol{if\ n > 0} \end{cases}$$

1) Iterative Substitution Method

Example 1 (Solution):

Now, let's follow the procedure specified above.

$$
\begin{aligned}
T(n) &= T(n-1)+1 \\
&= (T(n-2)+1)+1 = T(n-2)+2 \\
&= (T(n-3)+1)+2 = T(n-3)+3 \\
&\quad \vdots \qquad\qquad\qquad \vdots \qquad\qquad \vdots \\
&= T(n-k)+k
\end{aligned}
$$

1) Iterative Substitution Method

Example 1 (Solution):

- To solve the generalized a last relation, we have to find the value of $\boldsymbol{k}$. We note that $\boldsymbol{T(0)}$, that is the number of operations needed to raise a number $\boldsymbol{x}$ to power $\boldsymbol{1}$ needs just one operation.
- We can reduce $\boldsymbol{T(n-k)}$ to $\boldsymbol{T(0)}$ by setting

$$\boldsymbol{(n-k)=0 \implies k=n}$$

1) Iterative Substitution Method

Example 1 (Solution):

- We can reduce $T(n-k)$ to $T(0)$ by setting

$$(n-k)=0 \Rightarrow k=n$$

- Substituting this value of k in a last relation, we get

$$T(n)=T(0)+n$$

- Now we substitute the value of $T(0)$ to get the solution.

$$\boxed{T(n)=n+1=O(n)}$$

1) Iterative Substitution Method

Example 2:

Consider the Recurrence: $\boldsymbol{T(n)} = \begin{cases} \mathbf{2} & \boldsymbol{if\ n = 1} \\ \mathbf{2}\boldsymbol{T}\left(\frac{\boldsymbol{n}}{\mathbf{2}}\right) + \boldsymbol{n} & \boldsymbol{if\ n > 1} \end{cases}$

- **Solve the following recurrence relation using the iterative substitution method ?**

1) Iterative Substitution Method

Example 2 (Solution):

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2[2T\left(\frac{n}{4}\right) + \frac{n}{2}] + n = 4T\left(\frac{n}{4}\right) + 2n \\
&= 4[2T\left(\frac{n}{8}\right) + \frac{n}{4}] + 2n = 8T\left(\frac{n}{8}\right) + 3n \\
&= 8[2T\left(\frac{n}{16}\right) + \frac{n}{8}] + 3n = 16T\left(\frac{n}{16}\right) + 4n \\
&\quad \vdots \qquad\qquad\qquad \vdots \qquad\qquad\qquad \vdots \\
&= 2^k T\left(\frac{n}{2^k}\right) + k.n \qquad\qquad ; 1 \le k \le log_2 n
\end{aligned}
$$

1) Iterative Substitution Method

Example 2 (Solution):

- The maximum value of $\boldsymbol{k = log_2 n}$ (then only we get $\boldsymbol{T(n)}$)

$$T(n) \ = 2^{log_2 n} T\left(\frac{n}{2^{log_2 n}}\right) + n.\, log_2 n$$

$$= n.\, T(1) \ + n.\, log_2 n$$

$$= 2n \ + n.\, log_2 n$$

$$\boxed{= O(n \, log \, n)}$$

1) Iterative Substitution Method

Note: If you want to write a recurrence relation for $\boldsymbol{T(n-1)}$, you must replace ALL instances of $\boldsymbol{n}$ with $\boldsymbol{(n-1)}$, even the terms outside the recursive call! A common mistake is only substituting the $\boldsymbol{n}$ in the recursive term.

Example 3:

$$T(n) = \begin{cases} \boldsymbol{1} & \boldsymbol{if\ n = 0} \\ \boldsymbol{T(n-1) + log\ n} & \boldsymbol{if\ n > 0} \end{cases}$$

- Correct: $\boldsymbol{T(n-1) = T(n-2) + log(n-1)}$
- Incorrect: $\cancel{\boldsymbol{T(n-1) = T(n-2) + log(n)}}$

2) Recurrence Tree Method

- While the substitution method works well for many recurrence relations, it **is not an appropriate** technique when recurrence relation model algorithms are based on the "**divide and conquer**" paradigm.
- The recurrence tree method is a popular technique for solving such recurrence relations, particularly for solving **unbalanced recurrence relations**.

2) Recurrence Tree Method

- In the recursion method, we draw a recurrence tree and **calculate the time taken at each level of the tree.**
- Finally, **we sum the costs** within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.
- To draw the recurrence tree, we start from the given recurrence and **keep drawing until we find a pattern among levels.** This pattern is usually **an arithmetic or geometric series.**

2) Recurrence Tree Method

Given a recurrence relation, we can build a recursion tree by repeating the following steps until we reach the base case:

- At each level, each node is assigned the total amount of work that is done outside the recursive call(s).
- Each recursive call creates a branch to the next level of the tree.

2) Recurrence Tree Method

Example:

Consider the following recurrence relation :

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

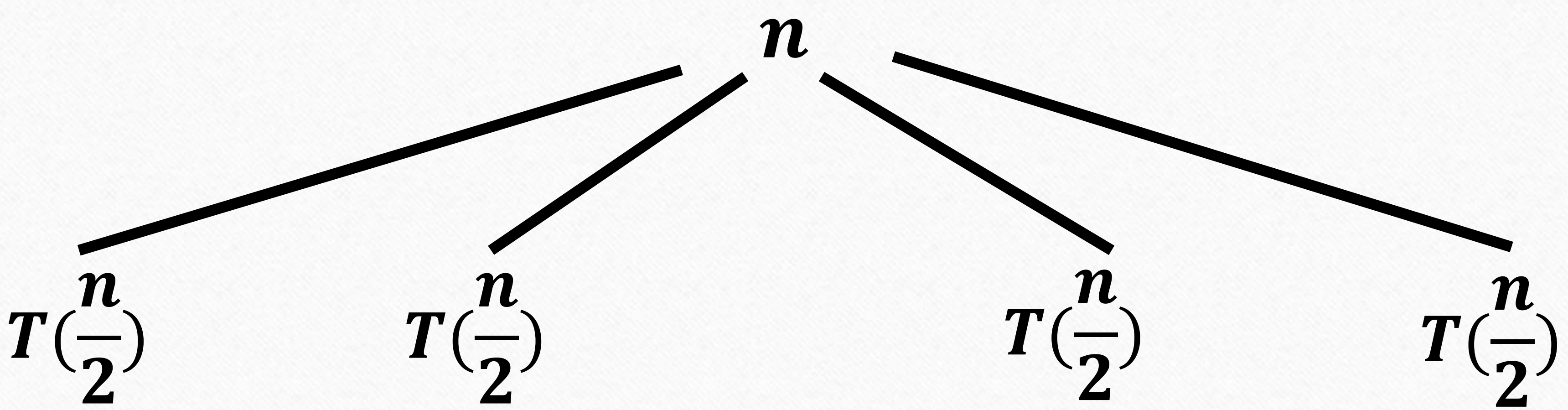
- This recurrence relation calls itself four times with half the input size and does an additional n work.
- With a recursion tree approach, we can visualize the first recursive call as follows

2) Recurrence Tree Method

Example:

Consider the following recurrence relation :

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$



2) Recurrence Tree Method

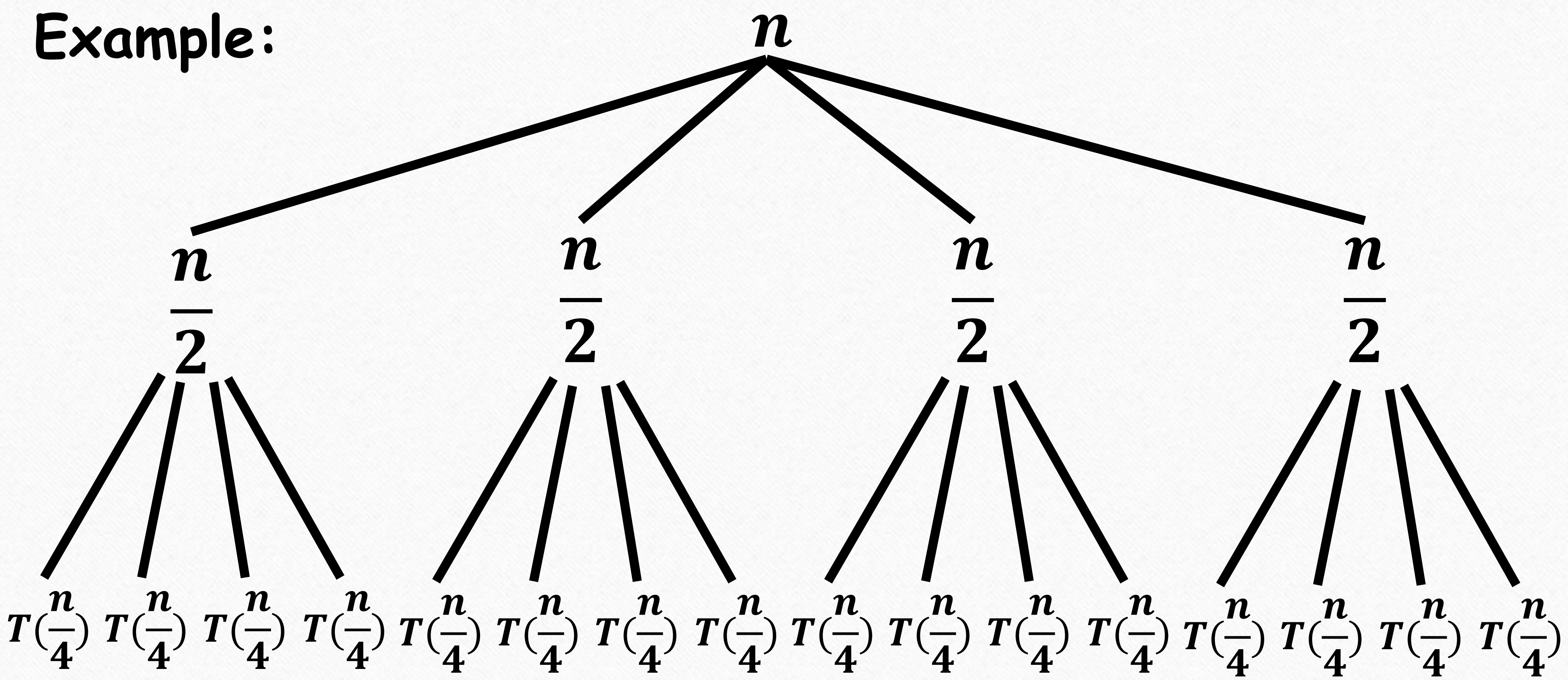
Example:

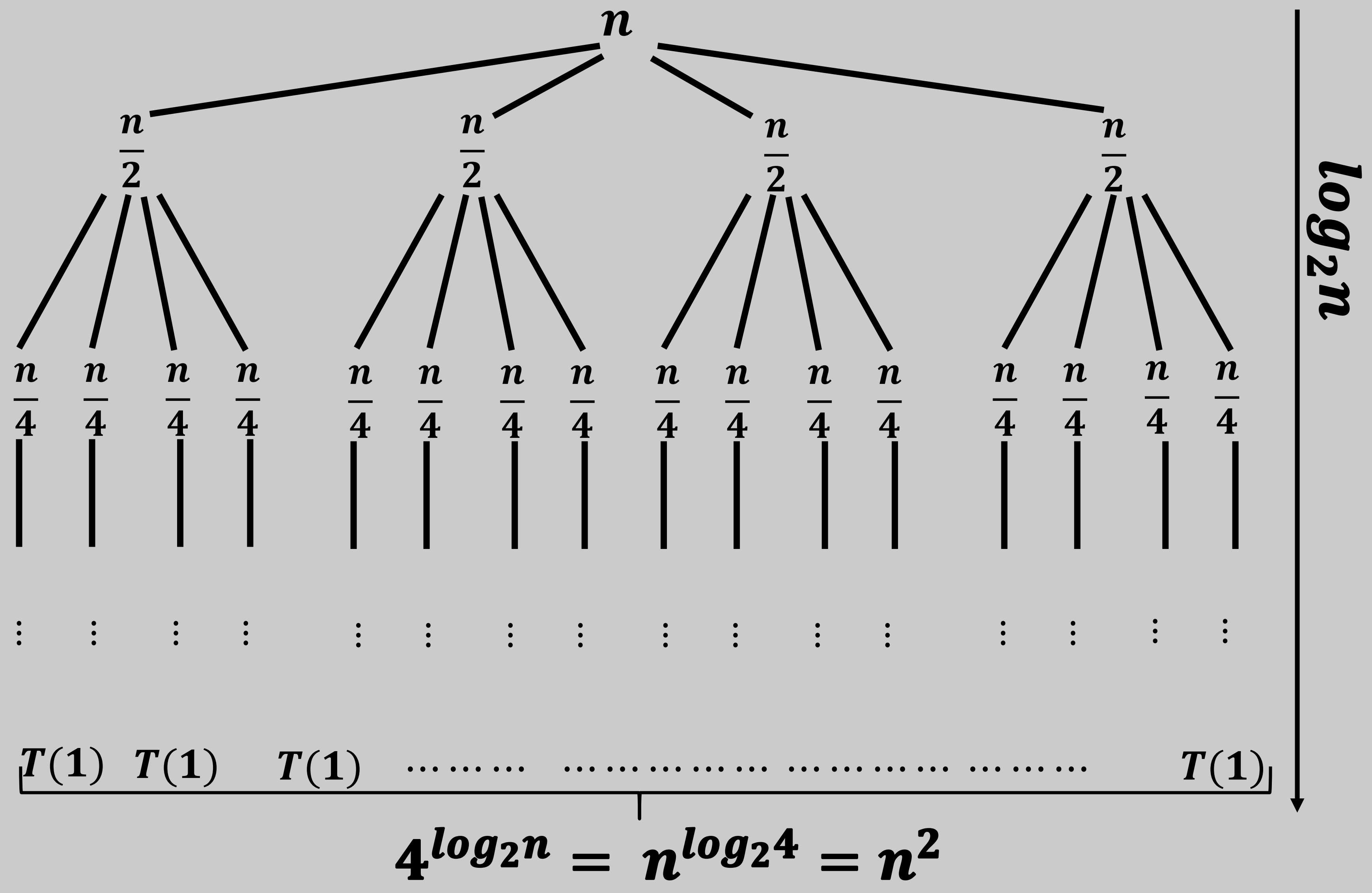
How much work do these recursive calls to $\boldsymbol{T\left(\frac{n}{2}\right)}$ do?

- We know from our recurrence relation that $\boldsymbol{T\left(\frac{n}{2}\right) = 4T\left(\frac{n}{4}\right) + \frac{n}{2}}$, where a recursive call with an input size of $\boldsymbol{\frac{n}{2}}$ does $\boldsymbol{\frac{n}{2}}$ work and makes two recursive calls, each with input size $\boldsymbol{\frac{n}{4}}$.
- We can use this to extend our tree down another level.

2) Recurrence Tree Method

Example:



Level	Recursion Tree for $T(n) = 4T\left(\frac{n}{2}\right) + n$	Work
$T(n)$		n
$T\left(\frac{n}{2}\right)$		$4(\frac{n}{2}) = 2n$
$T\left(\frac{n}{4}\right)$		$16(\frac{n}{4}) = 4n$
$T\left(\frac{n}{2^i}\right)$		$4^i\left(\frac{n}{2^i}\right) = 2^i n$
		$n^2 T(1)$

2) Recurrence Tree Method

Example:

$$Total = n + 2n + 4n + \cdots + 2^{log_2 n - 1} n + n^2 T(1)$$

$$= n\left[1 + 2 + 4 + \cdots + 2^{log_2 n - 1}\right] + n^2 T(1)$$

$$= n \sum_{i=0}^{log_2 n - 1} 2^i + n^2 T(1)$$

$$= n\left(\frac{2^{log_2 n - 1} - 1}{2 - 1}\right) + n^2 T(1) = n\left(\frac{n-2}{1}\right) + n^2 T(1)$$

$$\boxed{= O\left(n^2\right)}$$

2) Recurrence Tree Method

Summary:

- Let $\boldsymbol{T(n)}$ is defined by the recurrence relation :

$$\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

- where $\boldsymbol{a \geq 1}$ and $\boldsymbol{b > 1}$ are constants and $\boldsymbol{f(n)}$ is an asymptotically positive function.

2) Recurrence Tree Method

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Summary:

We conclude our work with these affirmations:

- Number of nodes at level $\boldsymbol{i}$: $\boldsymbol{a^i}$
- Input Size at level $\boldsymbol{i}$: $\boldsymbol{\frac{n}{b^i}}$
- Height of tree: $\boldsymbol{log_b\, n}$
- Number of levels: $\boldsymbol{a^{log_b\, n} = n^{log_b\, a}}$
- Cost at level $\boldsymbol{i}$: $\boldsymbol{a^i f(\frac{n}{b^i})}$.
- Total complexity : $\boldsymbol{n^{log_b a}.T(1) + \sum_{i=0}^{log_b n-1} a^i f(\frac{n}{b^i})}$

3) Master Theorem Method

The master method provides a “cookbook” method for solving recurrences of the form

$$\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

where $\boldsymbol{a \geq 1}$ and $\boldsymbol{b > 1}$ are constants and $\boldsymbol{f(n)}$ is an asymptotically positive function (Asymptotically positive means that the function is positive for all sufficiently large $\boldsymbol{n}$).

3) Master Theorem Method

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- n is the size of the problem.
- a is the number of sub-problems in the recursion.
- $\frac{n}{b}$ is the size of each sub-problem. (Here, it is assumed that all sub-problems are essentially the same size.)
- $f(n)$ is the sum of the work done outside the recursive calls, which includes the cost of decomposing the problem and recomposing the results.

3) Master Theorem Method

Theorem I.1 (Master theorem)

If the recurrence relation of an algorithm is of the form:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $a \geq 1$, $b > 1$, and $f(n)$ is an asymptotically positive function that is $\mathcal{O}(n^c)$, then the following is true:

- <u>Case 1:</u> If $a > b^c$, the complexity of $T(n) = \mathcal{O}\left(n^{log_b(a)}\right)$.
- <u>Case 2:</u> If $a = b^c$, the complexity of $T(n) = \mathcal{O}(n^c \, log_b(n))$.
- <u>Case 3:</u> If $a < b^c$, the complexity of $T(n) = \mathcal{O}(n^c)$.

3) Master Theorem Method

Theorem I.1 (Master theorem)

If the recurrence relation of an algorithm is of the form:

$$\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + f(n)}$$

where $\boldsymbol{a \geq 1}$, $\boldsymbol{b > 1}$, and $\boldsymbol{f(n)}$ is an asymptotically positive function that is $\boldsymbol{\mathcal{O}(n^c)}$, then the following is true:

More formally,

$$\boldsymbol{T(n) = \begin{cases} \mathcal{O}\left(n^{log_b(a)}\right) & if\ a > b^c \\ \mathcal{O}(n^c\ log_b(n)) & if\ a = b^c \\ \mathcal{O}(n^c) & if\ a < b^c \end{cases}}$$

3) Master Theorem Method

Notes:

- There must exist a base case that is solvable in constant time.
- The value $\boldsymbol{a}$ represents the number of times a recursive call is made,
- The value $\boldsymbol{b}$ represents the factor that the input is divided by with each recursive call,
- The value $\boldsymbol{c}$ represents the highest power of the polynomial term $\boldsymbol{f(n)}$.
- The values of $\boldsymbol{a}$ and $\boldsymbol{b}$ are independent of $\boldsymbol{n}$.
- The Master Theorem can only be used if all these conditions are met.

3) Master Theorem Method

Summary:

The steps for using the Master Theorem are as follows:

1) Determine the values of $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$.
2) Make sure that the Master Theorem can be used on the recurrence relation.

3) Master Theorem Method

Summary:

2) Make sure that the Master Theorem can be used on the recurrence relation.
   - The coefficient of the recursive call, $\boldsymbol{a}$, must be at least one $(\boldsymbol{a \geq 1})$.
   - The argument of the recursive call must be divided by some number, $\boldsymbol{b}$, that is larger than one $(\boldsymbol{b > 1})$.
   - The function $\boldsymbol{f(n)}$ must be an asymptotically positive function with complexity $\boldsymbol{O(n^c)}$.
   - There exists a base case that can be solved in constant time e.g., $\boldsymbol{T(1) = 1}$.

3) Master Theorem Method

Summary:

The steps for using the Master Theorem are as follows:

1) Determine the values of $\boldsymbol{a}$, $\boldsymbol{b}$, and $\boldsymbol{c}$.
2) Make sure that the Master Theorem can be used on the recurrence relation.
3) Compare the values of a $\boldsymbol{a}$ and $\boldsymbol{b^c}$ to determine which case of the Master Theorem should be used.

3) Master Theorem Method

Example 1:

Consider the following recurrence relation :

$$T(n) = 4T\left(\frac{n}{2}\right) + n$$

- Solve the recurrence by using the master theorem?

3) Master Theorem Method

Example 1 (Solution):

- For this recurrence, we have

$$a = 4, b = 2, \text{ and } f(n) = n = O\left(n^1\right) \Rightarrow c = 1$$

- Since $a > b^c$, $\left[4 > 2^1\right]$
- As per ***case* 1** of the master theorem, the solution is :

$$T(n) = O\left(n^{log_2(4)}\right) = O\left(n^2\right)$$

3) Master Theorem Method

Example 2:

Consider the following recurrence relation :

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$

- Solve the recurrence by using the master theorem?

3) Master Theorem Method

Example 2 (Solution):

- For this recurrence, we have

$$\boldsymbol{a} = \mathbf{2}, \boldsymbol{b} = \mathbf{2}, \text{ and } \boldsymbol{f}(\boldsymbol{n}) = \boldsymbol{n} = \boldsymbol{O}\left(\boldsymbol{n}^{\mathbf{1}}\right) \Rightarrow \boldsymbol{c} = \mathbf{1}$$

- Since $\boldsymbol{a} = \boldsymbol{b}^{\boldsymbol{c}}$, $\left[\mathbf{2} = \mathbf{2}^{\mathbf{1}}\right]$
- As per ***case* 2** of the master theorem, the solution is :

$$\boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{O}\left(\boldsymbol{n}^{\mathbf{1}}\, \boldsymbol{log}_{\mathbf{2}}(\boldsymbol{n})\right) = \boldsymbol{O}(\boldsymbol{n} \log \boldsymbol{n})$$

3) Master Theorem Method

Example 3:

Consider the following recurrence relation :

$$T(n) = 3T\left(\frac{n}{4}\right) + n^2$$

- Solve the recurrence by using the master theorem?

3) Master Theorem Method

Example 3 (Solution):

- For this recurrence, we have

$$a = 3, b = 4, \text{ and } f(n) = n^2 = O(n^2) \Rightarrow c = 2$$

- Since $a < b^c$, $[3 < 4^2]$
- As per *case* **3** of the master theorem, the solution is :

$$T(n) = O(n^2)$$

3) Master Theorem Method

Example 4:

Consider the following recurrence relation :

$$T(n) = 2T\left(\frac{n}{2}\right) + n\log n$$

- Solve the recurrence by using the master theorem?

3) Master Theorem Method

Example 4 (Solution):

- For this recurrence, we have

  $\boldsymbol{a = 2, b = 2}$, and $\boldsymbol{f(n) = n \log n = O(n \log n)}$

- Does not satisfy either **Case 1 or 2 or 3** of the Master's theorem.

- Here, the master theorem does not apply.

a) The Extended Master Theorem for Polylogarithmic Functions

- There is actually an extension of the Master Theorem that can be used if $\boldsymbol{f(n)}$ is a **polylogarithmic function**, but you will not need to know this for the class.
- Given a recurrence relation of the form

$$\boldsymbol{T(n) = aT\left(\frac{n}{b}\right) + f(n)},$$

where $\boldsymbol{f(n)}$ is of the form $\boldsymbol{T(n) = O\left(n^c\, log^k(n)\right)}$, the following are true:

a) The Extended Master Theorem for Polylogarithmic Functions

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Extended Master theorem

- <u>Case 1:</u> If $a > b^c$, then $T(n) = \mathcal{O}\left(n^{log_b(a)}\right)$.
- <u>Case 2:</u> If $a = b^c$, then
  - ❑ If $k > -1$, then $T(n) = \mathcal{O}\left(n^{log_b(a)}\, log^{k+1}(n)\right)$.
  - ❑ If $k = -1$, then $T(n) = \mathcal{O}\left(n^{log_b(a)}\, log(log(n))\right)$.
  - ❑ If $k < -1$, then $T(n) = \mathcal{O}\left(n^{log_b(a)}\right)$.
- <u>Case 3:</u> If $a < b^c$, then
  - ❑ If $k \geq 0$, then $T(n) = \mathcal{O}\left(n^c\, log^k(n)\right)$.
  - ❑ If $k < 0$, then $T(n) = \mathcal{O}(n^c)$.

a) The Extended Master Theorem for Polylogarithmic Functions

Example 1:

Consider the following recurrence relation :

$$T(n) = 3T\left(\frac{n}{2}\right) + \log_2 n$$

- Solve the recurrence by using the master theorem or the extended master theorem?

a) The Extended Master Theorem for Polylogarithmic Functions

Example 1 (Solution):

- Here, the master theorem does not apply, but rather we apply the extended master theorem.
- For this recurrence, we have

  $\boldsymbol{a} = \mathbf{3}, \boldsymbol{b} = \mathbf{2}$, and $\boldsymbol{f}(\boldsymbol{n}) = \boldsymbol{\mathcal{O}}(\,\boldsymbol{log}\ \boldsymbol{n}) \Longrightarrow \boldsymbol{c} = \mathbf{0}\ \&\ \boldsymbol{k} = \mathbf{1}$
- Since $\boldsymbol{a} > \boldsymbol{b}^{\boldsymbol{c}}$, $[\mathbf{3} > \mathbf{2}^{\mathbf{0}}]$
- As per $\boldsymbol{case}\ \mathbf{1}$ of the extended master theorem, the solution is :

$$\boldsymbol{T}(\boldsymbol{n}) = \boldsymbol{\mathcal{O}}\left(\boldsymbol{n}^{\boldsymbol{log}_{\boldsymbol{b}}(\boldsymbol{a})}\right) = \boldsymbol{\mathcal{O}}\left(\boldsymbol{n}^{\boldsymbol{log}_{\mathbf{2}}(\mathbf{3})}\right)$$

a) The Extended Master Theorem for Polylogarithmic Functions

Example 2:

Consider the following recurrence relation :

$$T(n) = 2T\left(\frac{n}{2}\right) + n \log n$$

- Solve the recurrence by using the master theorem or the extended master theorem?

a) The Extended Master Theorem for Polylogarithmic Functions

Example 2 (Solution):

- Here, the master theorem does not apply, but rather we apply the extended master theorem.
- For this recurrence, we have

$$\boldsymbol{a = 2, b = 2}, \text{ and } \boldsymbol{f(n) = O(n \log n) \Rightarrow c = 1 \;\&\; k = 1}$$

- Since $\boldsymbol{a = b^c}$, $[2 = 2^1]$
- As per **case 1** of the extended master theorem, $\boldsymbol{k = 1 \Rightarrow k > -1}$, the solution is :

$$\boldsymbol{T(n) = O\left(n^{\log_b(a)} \log^{k+1}(n)\right) = O\left(n^{\log_2 2} \log^2 n\right) = \left(n \log^2 n\right)}$$

Exercise

Consider the following recurrence relation :

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

- **Solve the following recurrence relation ?**

Exercise

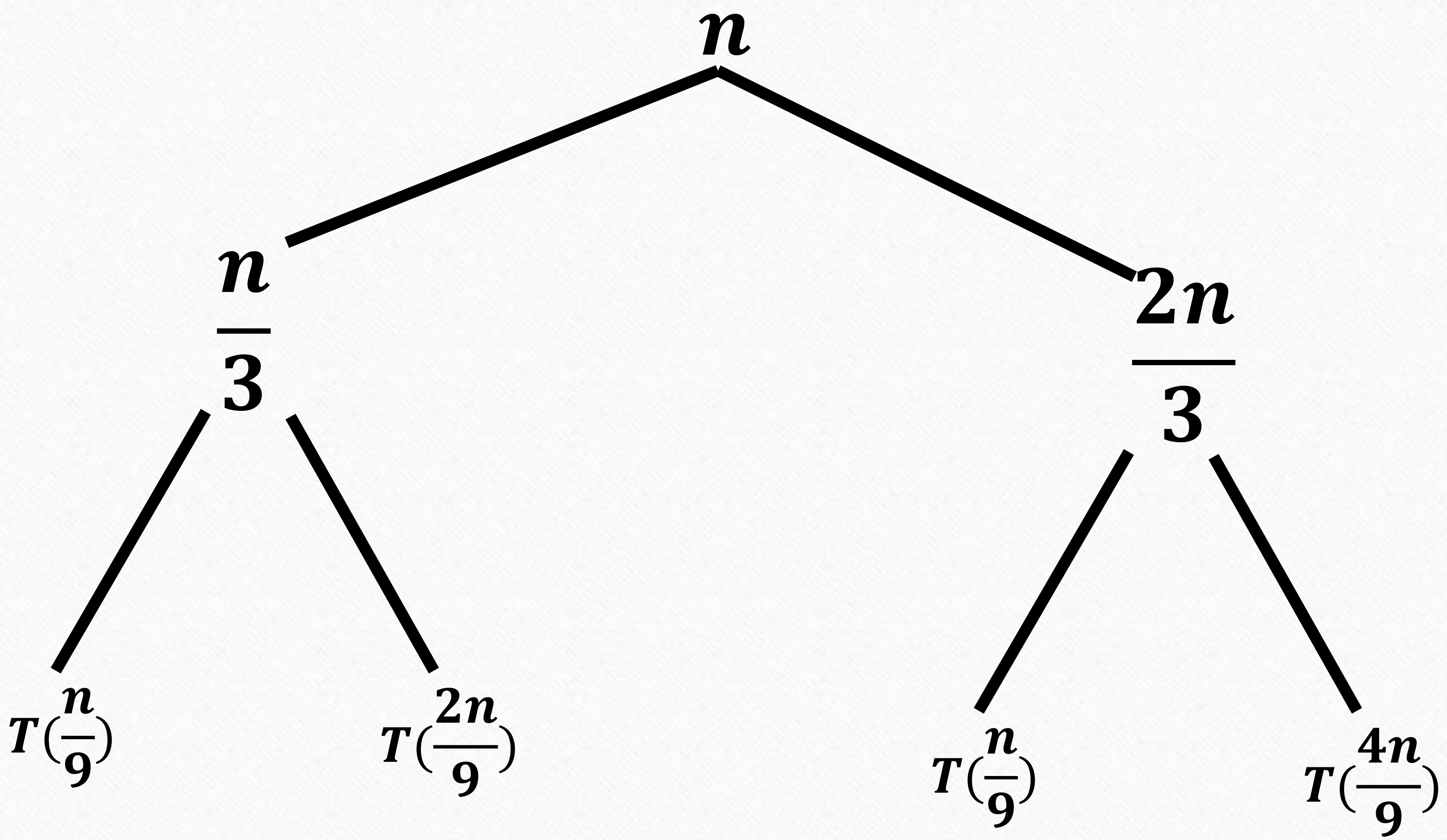
Consider the following recurrence relation :

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

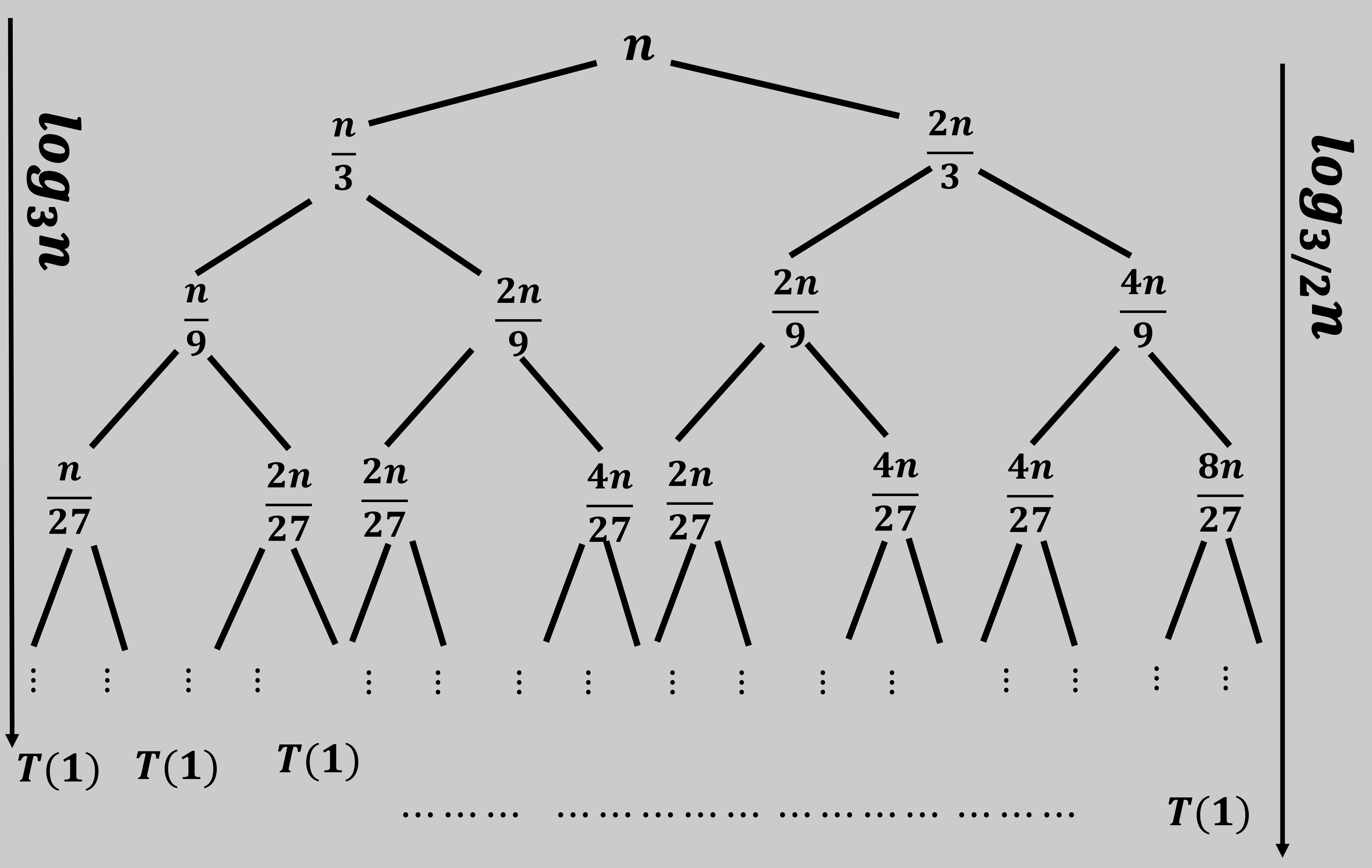
$$n$$

$$T\left(\frac{n}{3}\right) \qquad\qquad\qquad T\left(\frac{2n}{3}\right)$$

Exercise



Level	Recursion Tree for $T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$	Work
0		n
1		$\left(\frac{n}{3}\right) + \left(\frac{2n}{3}\right) = n$
2		$\left(\frac{9n}{9}\right) = n$
3		$\left(\frac{27n}{27}\right) = n$
k		$\left(\frac{3^k n}{3^k}\right) = n$
		$< n$



Exercise

$$Total = \underbrace{n + n + n + \cdots + n + n}_{\log_{3/2} n \ times}$$

$$= n \log_{3/2} n$$

$$\boxed{= O(n \log n)}$$

Questions ?