CHAPTER 1

Algorithm Analysis

I.1. Introduction

This chapter focuses on the analysis of execution time as well as the smooth running of algorithms. Analysis of an algorithm is the same thing as estimating the efficiency of the algorithm. There are two fundamental parameters based on which we can analyze the algorithm and they are **Space** and **Time Complexity**. There is also the concept in time complexity of estimating the running time of an algorithm and we have the **Worst-case**, **Average-case**, and **Best-case**. We will introduce several tools to accomplish such an analysis: asymptotic notation **(O, Θ, Ω)**. These notions will be accompanied by examples.

I.2. 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.

Data structure can be classified into two categories:

1) **Primitive data structures** consist of the numbers and the characters which are built in programs. These can be manipulated or operated directly by the machine level instructions. Basic data types such as integer, real, character, and Boolean come under primitive data structures. These data types are also known as simple data types because they consist of characters that cannot be divided.

2) **Non-primitive data structures** are those that are derived from primitive data structures. These data structures cannot be operated or manipulated directly by the machine level instructions. They focus on formation of a set of data elements that is either homogeneous (same data type) or heterogeneous (different data type).

   These are further divided into linear and non-linear data structure based on the structure and arrangement of data.

   - **Linear data structures**: Elements are accessed in a sequential order but it is not compulsory to store all elements sequentially (say, Linked Lists). Examples: Linked Lists, Stacks and Queues.
   - **Non-linear data structures**: Elements of this data structure are stored/accessed in a non-linear order. Examples: Trees and graphs.

The following figure (Figure 1.1) shows the different classifications of Data Structures.

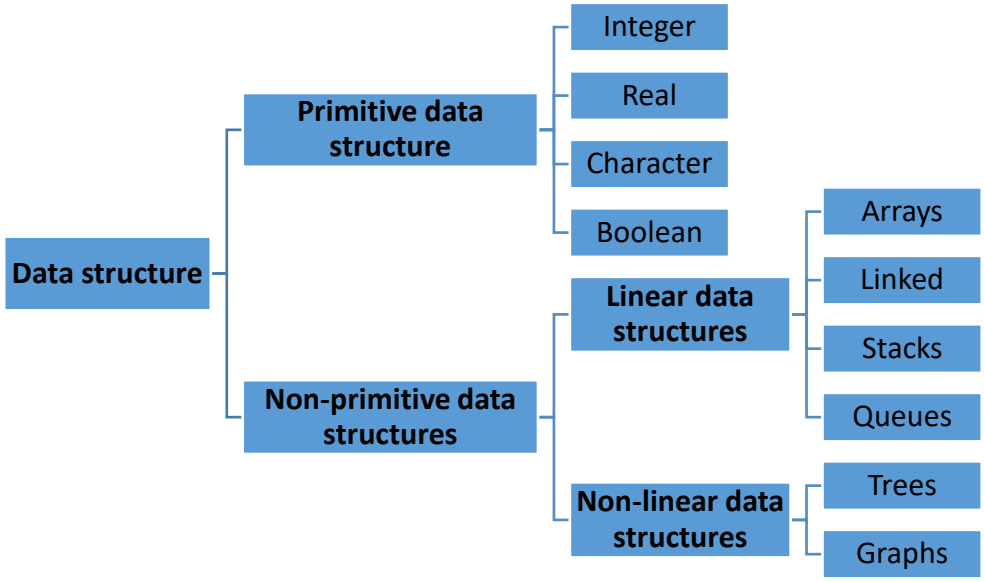


Figure 1.1 : Classifications of Data Structures.

I.3. 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.

I.3.1. Characteristics of Algorithms

The main Characteristics or features of Algorithms are:

- **Input**: Algorithms take zero or more inputs and produce at least one output based on these inputs.
- **Output**: Every algorithm should produce a result; it can be a value, a set of values, a decision, or some other form of output.
- **Definiteness:** Algorithms should have clear and unambiguous instructions for each step. Each step should be precisely defined and executable.
- **Finiteness**: An algorithm should terminate after executing a finite number of steps. It should not run indefinitely or enter an infinite loop.
- **Feasible**: The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Effectiveness**: Algorithms should be practical and feasible. They should be able to be executed within the constraints of time and space available.
- **Language Independent**: The Algorithm designed must be language independent, i.e., it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

I.3.2. 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.

I.3.3. 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.

I.4. 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.

We will be looking more at time rather than space because time is instead a more limiting parameter in terms of the hardware. It is not easy to take a computer and change its speed. So, if we run an algorithm on a particular platform, we are more or less stuck with the performance the platform can give us in terms of speed.

I.4.1. Execution Time / Runtime

Usually, program execution time (program run-time) is referred to as time complexity, denoted by $\boldsymbol{T_p}$ (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.**

I.4.1.1. Runtime Based on a Single Parameter

Let $\boldsymbol{\mathcal{A}}$ be an algorithm and $\boldsymbol{f}$ the function such that $\boldsymbol{f(x)}$ denotes the number of elementary operations performed by $\boldsymbol{\mathcal{A}}$ on input $\boldsymbol{x}$. The execution time of $\boldsymbol{\mathcal{A}}$ in the worst case is the function $\boldsymbol{T_{max}} : \mathbb{N} \to \mathbb{N}$ such that:

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

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

Example:

The code below finds the sum of $\boldsymbol{n}$ 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.

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, Assignment	n	2n
`return s;}`	0	/	-	0
Total			T(n)=	6n+3

Intuitively, then, Algorithm (function) works in linear time concerning n, i.e., whatever the input, the execution time is approximately n except for certain constants.

I.4.1.2. 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:

$$\boldsymbol{T_{max}(n_1, n_2, \ldots, n_k) = Max\{f(x): input\ x\ of\ size\ (n_1, n_2, \ldots, n_k)\}.}$$

Example:

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

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

I.4.2. Types of Time Complexity Analysis

We have three types of analysis related to time complexity, which are:

1) **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.

2) **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**.
3) **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**.

I.5. 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.

$\boldsymbol{O(f)}$ notation represents the complexity of an algorithm, which is also termed an Asymptotic notation or **"Big O"** notation. Here, the $\boldsymbol{f}$ corresponds to the function whose size is the same as that of the input data. The complexity of the asymptotic computation $\boldsymbol{O(f)}$ determines in which order the resources such as CPU time, memory, etc. are consumed by the algorithm that is articulated as a function of the size of the input data.

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".

I.6. 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.

I.6.1. Asymptotic Notations

A problem may have various algorithmic solutions. In order to choose the best algorithm for a particular process, you must be able to judge the time taken to run a particular solution. More precisely, you must be able to judge the time it takes to run two solutions and choose the best one. 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. It can generally be represented as the fastest possible, the slowest possible or the average possible.

The notations such as **O (Big-O)**, **Θ (Theta),** and **Ω (Omega)** are called as asymptotic notations. These are the mathematical notations that are used in three different cases of time complexity.

Since **Big-O notation** gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario. Therefore, in this course, we'll focus on the **Big-O notation**.

I.6.1.1. 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)**.

Definition I.1

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

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

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

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

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

The graphical representation of **f(n) = O(g(n))** is shown in figure 1.2, where the running time increases considerably when **n** increases.

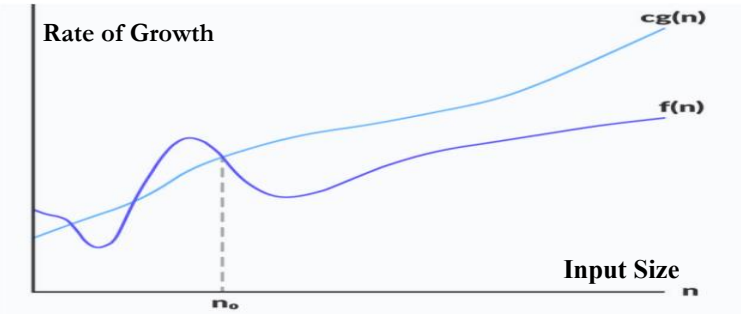


Figure 1.2: Big O Notation f(n) = O(g(n))

Example:

Consider the function $\boldsymbol{f(n) = 5n^2 + 400n + 9}$

The function $\boldsymbol{f}$ is greater than $\boldsymbol{g(n) = n^2}$

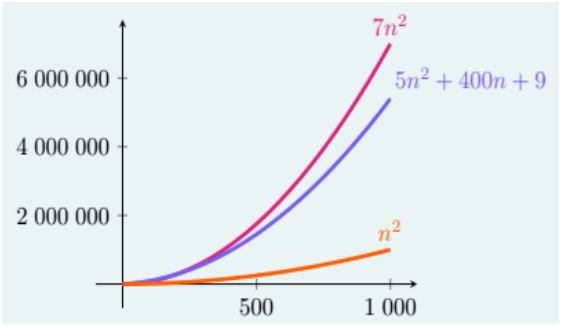
However, we can show that $\boldsymbol{f \in \mathcal{O}(n^2)}$. We have :

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

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

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

$= \mathbf{7n^2}$



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

I.6.1.2. Big-O Notation for Complexity Class

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

- $\mathcal{O}(\mathbf{1})$ pronounced 'Oh of one', or constant complexity;
- $\mathcal{O}(\boldsymbol{log_2 log_2 n})$ 'Oh of log log en';
- $\mathcal{O}(\boldsymbol{log_2 n})$ 'Oh of log en', or logarithmic complexity;
- $\mathcal{O}(\boldsymbol{n})$ 'Oh of en', or linear complexity;
- $\mathcal{O}(\boldsymbol{n log_2 n})$ 'Oh of en log en';
- $\mathcal{O}(\boldsymbol{n^2})$ 'Oh of en squared', or quadratic complexity;
- $\mathcal{O}(\boldsymbol{n^3})$ 'Oh of en cubed', or cubic complexity;
- $\mathcal{O}(\boldsymbol{2^n})$ 'Oh of two to the en', or exponential complexity.
- $\mathcal{O}(\boldsymbol{n!})$ 'Oh of en factorial', or factorial complexity.

As a representative, we choose the function which gives the class its name e.g., for $\mathcal{O}(\boldsymbol{n})$ we choose the function $\boldsymbol{f(n) = n}$, for $\mathcal{O}(\boldsymbol{log_2 n})$ we choose $\boldsymbol{f(n) = log_2 n}$, and so on. Let's assume we have algorithms with these functions describing their complexity. Table 1.1 lists how many operations it will take them to deal with a problem of a given size:

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

Table 1.1: Number of operations.

Some of these numbers are so large that it is rather difficult to imagine just how long a time span they describe. Hence, Table 1.2 gives time spans rather than instruction counts, based on the assumption that we have a computer which can operate at a speed of 1 MIPS (MIPS : a million instructions per second):

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

Table 1.2: Time spent.

It is clear that, as the sizes of the problems get really big, there can be huge differences in the time it takes to run algorithms from different complexity classes. For algorithms with exponential

complexity, $\boldsymbol{O(2^n)}$, even modest-sized problems have run times that are greater than the age of the universe (about 1,4 × 1010 years), and current computers rarely run uninterrupted for more than a few years. This is why complexity classes are so important they tell us how feasible it is likely to be to run a program with a particularly large number of data items. Typically, people do not worry much about complexity for sizes below 10, or maybe 20, but the above numbers make it clear why it is worth thinking about complexity classes where bigger applications are concerned.

I.6.1.3. Properties of Asymptotic Notations

To understand the concept of asymptotic notations fully, let us look at some of the general properties of asymptotic notations.

a) General (Constant Factor)

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

e.g., $\boldsymbol{f(n) = 2n^2 + 5\ is\ O(n^2), then\ 4 \times f(n) = 8n^2 + 20\ is\ also\ O(n^2)}$

b) Reflexive

Given $\boldsymbol{f(n), then\ f(n) = O(f(n))}$, 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., $\boldsymbol{f(n) = n^2 = O(n^2)}$

c) Transitive

If $\boldsymbol{f(n) = O(g(n))}$ *and* $\boldsymbol{g(n) = O(h(n)), then\ f(n) = O(h(n))}$

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

here, $\boldsymbol{f(n) = O(n^2)}$ & $\boldsymbol{g(n) = O(n^3) \implies f(n) = O(n^3)\ or\ O(h(n))}$

I.6.1.4. Limitations of Big-O Notation

Big-O notation has certain limitations when it comes to expressing the complexity of algorithms. These limitations are as follows:

- 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. this is an important consideration.

I.7. 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. Thus, we throw away leading constants. We will also throw away lower-order terms, so what 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. The program may stop earlier than this, but never later.

I.7.1. Assumptions in Complexity Analysis

Calculating the complexity of an algorithm is based on the following assumptions:

- **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 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.
- **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.

I.7.2. Simplification Rules

In algorithm analysis, particularly when using Big-O notation, there are simplification rules that are commonly employed to analyze the time and space complexity of algorithms. Here are some common simplification rules used in complexity analysis:

- **Drop Constants:** Constants in front of the dominant term are dropped in Big-O notation. For example: $\boldsymbol{\mathcal{O}(5n^3 + 4)}$ simplifies to $\boldsymbol{\mathcal{O}(n^3)}$.
- **Dominant Term Rule:** only the term with the largest growth rate dominates in Big-O notation. For example, in $\boldsymbol{\mathcal{O}(n^2 + n)}$, a dominant term is $\boldsymbol{n^2}$, so the complexity is $\boldsymbol{\mathcal{O}(n^2)}$.
- **Additive Rule:** When analyzing multiple operations in sequence, the complexities are added. For example, if an algorithm has two separate loops with complexities $\boldsymbol{\mathcal{O}(n^3)}$ and $\boldsymbol{\mathcal{O}(n)}$, the total complexity is $\boldsymbol{\mathcal{O}(n^3) + \mathcal{O}(n) = \mathcal{O}(n^3 + n) = \mathcal{O}(n^3)}$.
- **Multiplicative Rule:** When analyzing nested operations, the complexities are multiplied. For example, if an algorithm has nested loops with complexities $\boldsymbol{\mathcal{O}(n)}$ and $\boldsymbol{\mathcal{O}(n)}$, the total complexity is $\boldsymbol{\mathcal{O}(n) * \mathcal{O}(n) = \mathcal{O}(n^2)}$.
- **Logarithmic Rules**: Base of logarithms can be ignored in Big-O notation, as they represent constant factors. Therefore, $\boldsymbol{\mathcal{O}(log\ n)}$ and $\boldsymbol{\mathcal{O}(log_2\ n)}$ are considered the same.

I.7.3. How to Calculate the Approximate Time Kaken 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.

I.7.4. Guidelines for Asymptotic Analysis of Iterative Algorithms

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

I.7.4.1. Simple Statement and O(1)

For the purpose of this discussion, 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. 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 $\boldsymbol{O(1)}$.

I.7.4.2. 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 $\boldsymbol{O(1)}$.

I.7.4.3. 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. As an example, consider the following sequence of statements:

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

The time complexity of the sequence will be:

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

I.7.4.4. 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. As has already been mentioned, Big O provides us growth rate in the worst-case scenario. 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. Hence the Big O for an ***if*** or a ***switch*** statement would be equal to the maximum of the Big O of its individual branches. As an example, consider the following code segment:

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

The time complexity of this code segment will be:

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

I.7.4.5. Iterations (Loops)

In C (C++) language ***for***, ***while***, and ***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.

I.7.4.5.1. 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: $\mathcal{O}(f(n))\ .\mathcal{O}(f(n)) = \mathcal{O}(f(n).g(n))$

Example:

```
for( i = 1; i <= n; i ++)
   S = S + 2;  // constants time, O(1)
```

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

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

Simple loops come in many different varieties. The most common ones are the counting loops with uniform step size and loops where the step size is multiplied or divided by a fixed number. These are discussed in the following subsections.

I.7.4.5.2. 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 $\mathcal{O}(n)$. They are also very simple to analyze.

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

```
index = -1;    // O(1)
```

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

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

We can easily see that:

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

I.7.4.5.3. 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.

As an **example**, let us consider the following code for the Binary Search Algorithm:

```
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;
}
```

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

For the sake of simplicity, let us assume that **N** is a power of **2**. That is, we can write $N = 2^k$ where **k** is a non-negative integer. Once again, the worst case will occur if the key is not found. In each iteration, the search space spans from low to high. So, in the first iteration we have **N** elements in this range. After the first iteration, the range is halved as either the low or the high is moved to **mid + 1** or **mid - 1** respectively, effectively reducing the search space to approximately half the original size. This pattern continues until we either find the key or the condition **high >= low** is false, which is the worst-case scenario. This pattern is captured in this Table 1.3.

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

Table 1.3: The number of iterations in binary search

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

I.7.4.5.4. 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);
```

This looks like an algorithm with time complexity $\mathcal{O}(N)$. However, since the number of iterations is fixed, hence it is not dependent upon any input data size and therefore it will always take the same amount of time. Therefore, its time complexity is $\mathcal{O}(1)$.

I.7.4.5.5. 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.

a) 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];
```

For each iteration of the outer loop, the inner loop will iterate **COLS** number of times and is independent of any variable controlled in the outer loop. Since the outer loop iterates **ROWS** number of times, the statement in the body of the inner loop will be executed **ROWS x COLS** number of times. Hence the time complexity of the algorithms is $\mathcal{O}(\mathbf{ROWS} \times \mathbf{COLS})$.

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 $\mathcal{O}(\mathbf{x}) \times \mathcal{O}(\mathbf{y}) = \mathcal{O}(\mathbf{x} \times \mathbf{y})$.

b) 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.

As an **example**, consider the following algorithm:

```
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. In order to analyze such cases, we can use a table like the one given in Table 1.4.

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	$T(N) = N + (N-1) + (N-2) + \cdots + (N-i) + \cdots + 1$ $= \frac{N(N+1)}{2}$ $= \boldsymbol{\mathcal{O}(N^2)}$						

Table 1.4: Analysis of the algorithm.

As shown in the table, the time complexity of algorithm is thus $\boldsymbol{\mathcal{O}(N^2)}$.

Let us look at another interesting **example**.

```
k = 0;
for (i = 1; i < N; i++)
    for (j = 1; j <= i; j = j * 2)
        k++;
```

Once again, the number of iterations in the inner loop depend upon that value of **i** which is controlled by the outer loop. Hence it is a case of dependent loops. Let us now use the same technique to find the time complexity of this algorithm. 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 $[\boldsymbol{log_2\ (i+1)}]$. Using this information, the time complexity is calculated as shown in Table 1.5.

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	$T(N) = \log 2 + \log 3 + \log 4 + \log 5 + \cdots + \log N$ $= \log(2 \times 3 \times 4 \times 5 \times \ldots \times N) = \log(N!)$ $= \boldsymbol{\mathcal{O}(N \log N)}$ *by Stirling's formula*					

Table 1.5: Analysis of a first program involving nested dependent loop

Analyses of both the algorithms discussed above are a little deceiving. It appears that in both these cases we can get the right answer by simply finding the complexity of each loop independently and then multiplying them just like we did in the case of independent loops. Hence this whole exercise seems to be an overkill. Although this statement is true to the extent that we will indeed get the same answer, however, this is not the right approach and will not always give the right answer in such cases. To understand why, let us consider the following code:

```
for (int i = 1; i < N; i = i * 2)
   for (int j = 0; j < i; j++)
      k++;
```

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. The analysis is shown in Table 1.6.

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

Table 1.6: Analysis of a second program involving nested dependent loop

I.7.4.6. 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.

For **example**, let us consider the following program:

```
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;
}
```

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

Let us now use it to determine the time complexity of the following function:

```
float average(float data[], int size) // assuming size > 0
{
      float total, mean ;
      total = sum(data, size);      //O(N)
      mean = total/size;            //O(1)
      return mean;                  //O(1)
}
```

We first determine the time complexity of each statement. Then we determine the complexity of the entire function by simply using the method discussed earlier to determine the complexity of a sequence of statements. This gives us the time complexity of O(n) for the function.

It is important to remember that if we have a function call inside the body of a loop, we need to apply the guidelines for the loops. In that case we need to carefully examine the situation as sometimes the time taken by the function call is dependent upon a variable controlled outside the function and hence gives rise to a situation just like dependent loops discussed above. In such cases, the complexity is determined with the help of techniques similar to the one used in the case of dependent loops.

As an **example**, consider the following example:

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

As can be easily seen, if considered independently, this function has linear time complexity, i.e., it is $O(N)$. Let us now use it inside another function:

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

If we are not careful, we can be easily deceived by it. If we treat it casually, we can put the complexity of the for loop to be $O(\log N)$ and we have already determined that the complexity of the function foo is $O(N)$. Hence, the overall complexity would be calculated to be $O(N \log N)$. However, a more careful analysis would show that $O(N)$ is a tighter upper bound for this function

and that should be used to describe the complexity of this function. The detailed analysis of this problem is left as an exercise for the students.

Example: Let's calculate the complexity of the following algorithm (n! : factorial of n) :

```
int factorial(int n) {
      // Returns the factorial of the natural n.
      int fact = 1;          ← initialization: O(1)
      int i = 2;             ← initialization: O(1)
      while (i <= n){        ← test in : O(1)
          fact = fact * i;   ← multiplication and assignment: O(1)
          i = i + 1;         ← increment and assignment: O(1)
      }                                              } n − 1 iterations: O(n)
      return fact;           ← return: O(1)
}
```

According to the rules seen above, the complexity is: $\boldsymbol{O(1) + O(n) \times O(1) + O(1) = O(n)}$

I.7.5. Guidelines for Asymptotic Analysis of Recursive Algorithms

In this section, we will see how to apply the general framework for analysis of algorithms to 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. 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.

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.

I.7.5.1. 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.

Example:

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

$$\boldsymbol{n! = 1 \times 2 \times ... \times (n-1) \times n = n(n-1)! \quad for\ n \geq 1}$$

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

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

Since there is only one function, fact, involved, we shall use $\boldsymbol{T(n)}$ for the unknown running time of this function. We shall use $\boldsymbol{n}$, the value of the argument, as the size of the argument. Clearly, recursive calls made by fact when the argument is $\boldsymbol{n}$ have a smaller argument, $\boldsymbol{(n-1)}$ to be precise.

For the basis of the inductive definition of $\boldsymbol{T(n)}$ we shall take $\boldsymbol{n = 0}$, since no recursive call is made by fact when its argument is $\boldsymbol{0}$. With $\boldsymbol{n = 0}$, the condition of line $\boldsymbol{(1)}$ is true, and so the call to fact executes lines $\boldsymbol{(1)}$ and $\boldsymbol{(1)}$. Each takes constant $\boldsymbol{a}$ time i.e., $\boldsymbol{\mathcal{O}(1)}$ time, and so the running time of $\boldsymbol{Fact}$ in the basis case is $\boldsymbol{a}$ time i.e., $\boldsymbol{\mathcal{O}(1)}$. That is, $\boldsymbol{T(0)}$ is $\boldsymbol{\mathcal{O}(1)}$.

Now, consider what happens when $\boldsymbol{n > 0}$. The condition of line $\boldsymbol{(1)}$ is false, and so we execute only lines $\boldsymbol{(1)}$ and $\boldsymbol{(3)}$. Line $\boldsymbol{(1)}$ takes $\boldsymbol{b}$ time i.e., $\boldsymbol{\mathcal{O}(1)}$ time, and line $\boldsymbol{(3)}$ takes $\boldsymbol{a}$ time for the multiplication and assignment, plus $\boldsymbol{T(n-1)}$ for the recursive call to $\boldsymbol{Fact}$. That is, for $\boldsymbol{n > 0}$, the running time of $\boldsymbol{Fact}$ is $\boldsymbol{b + T(n-1)}$. We can thus define $\boldsymbol{T(n)}$ by the following recurrence relation:

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

Here, $\boldsymbol{T(n)}$ represents the runtime of $\boldsymbol{Fact(n)}$, $\boldsymbol{T(n-1)}$ represents the runtime of the recursive call to $\boldsymbol{Fact(n-1)}$, and the additional $\boldsymbol{\mathcal{O}(1)}$ term represents the constant work we do outside the recursive call. Since the recurrence relation will be used to calculate time complexities, having an exact number for this constant term does not matter.

I.7.5.2. 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, namely the iterative substitution method, the recurrence tree method, and the master theorem to analyze recurrence relations. Solutions to recurrence relations give the time complexity of the underlying algorithms.

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.

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 $\boldsymbol{1}$ if the base case is $\boldsymbol{T(n) = O(1)}$).

Example 1:

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

Solve the following recurrence relation using the iterative substitution method ?

Solution:

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

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

Now, let's follow the procedure specified above.

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

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

Substituting this value of $\boldsymbol{k}$ in a last relation, we get

$$\boldsymbol{T(n) = T(0) + n}$$

Now we substitute the value of $\boldsymbol{T(0)}$ to get the solution.

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

Example 2:

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

Solve the following recurrence relation using the iterative substitution method ?

Solution:

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

The maximum value of $k = log_2 n$ **(then only we get $T(n)$)**

$$\begin{aligned} T(n) &= 2^{log_2 n} T\left(\frac{n}{2^{log_2 n}}\right) + n.log_2 n \\ &= n.T(1) + n.log_2 n \\ &= 2n + n.log_2 n \\ &= O(n\ log\ n) \end{aligned}$$

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

Example 3:

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

Correct: $T(n-1) = T(n-2) + log(n-1)$

Incorrect: $T(n-1) = T(n-2) + log(n)$

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.

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.

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

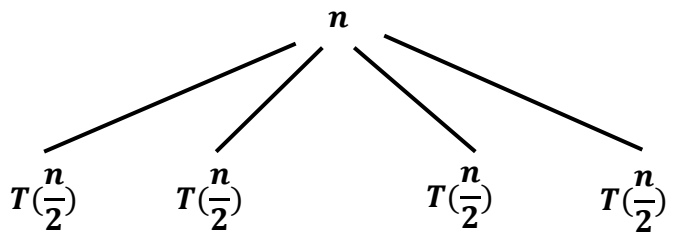
- At each level, each node is assigned the total amount of work that is done outside the recursive call(s).
- Each recursive call creates a branch to the next level of the tree.

Example:

Consider the following recurrence relation :

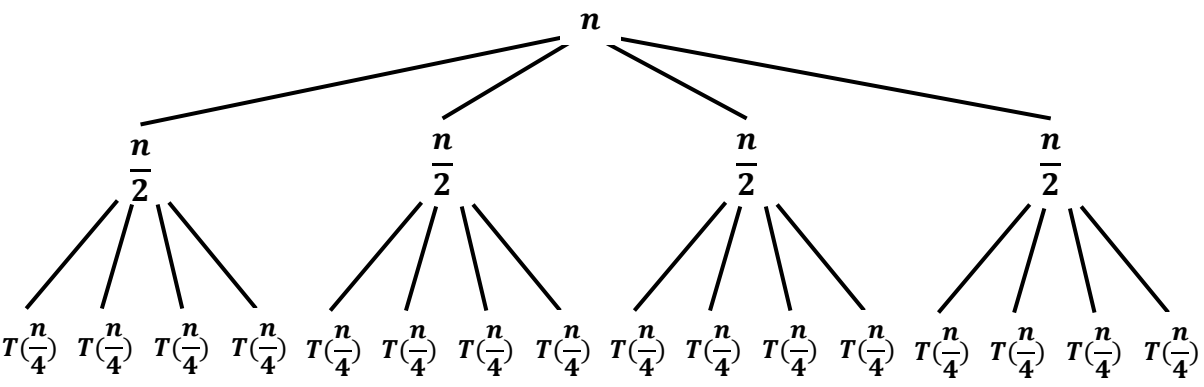
$$\boldsymbol{T(n) = 4T\left(\frac{n}{2}\right) + n}$$

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



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 $\frac{n}{2}$ does $\frac{n}{2}$ work and makes two recursive calls, each with input size $\frac{n}{4}$. We can use this to extend our tree down another level.



By repeating this process to the base case, we can construct the full recursion tree.

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

$\log n$

$4^{\log_2 n} = n^{\log_2 4} = n^2$

$$\begin{aligned} Total &= n + 2n + 4n + \cdots + 2^{\log_2 n - 1} n + n^2 T(1) \\ &= n\left[1 + 2 + 4 + \cdots + 2^{\log_2 n - 1}\right] + n^2 T(1) \\ &= n \sum_{i=0}^{\log_2 n - 1} 2^i + n^2 T(1) \\ &= n\left(\frac{2^{\log_2 n - 1} - 1}{2 - 1}\right) + n^2 T(1) = n\left(\frac{n-2}{1}\right) + n^2 T(1) \\ &= O(n^2) \end{aligned}$$

Summary:

Let $T(n)$ is defined by the recurrence relation : $T(n) = aT\left(\frac{n}{b}\right) + f(n)$

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

We conclude our work with these affirmations:

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

3) Master Theorem

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 $\boldsymbol{n}$.)

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

The Master Theorem provides a model that can be used to calculate the time complexity of a preceding recurrence relation. The theorem is defined below:

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:

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

More formally,

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

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.

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.
   - 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 $\boldsymbol{\mathcal{O}(n^c)}$.
   - There exists a base case that can be solved in constant time e.g., $\boldsymbol{T(1) = 1}$.
3) Compare the values of a $\boldsymbol{a}$ and $\boldsymbol{b^c}$ to determine which case of the Master Theorem should be used.

Example 1: Consider the following recurrence relation : $\boldsymbol{T(n) = 4T\left(\frac{n}{2}\right) + n}$

Solve the recurrence by using the master theorem?

Solution:

For this recurrence, we have $\boldsymbol{a = 4, b = 2}$, and $\boldsymbol{f(n) = n = \mathcal{O}(n^1) \implies c = 1}$

Since $\boldsymbol{a > b^c}$, $\boldsymbol{[4 > 2^1]}$

As per $\boldsymbol{case\ 1}$ of the master theorem, the solution is : $\boldsymbol{T(n) = \mathcal{O}\left(n^{log_2(4)}\right) = \mathcal{O}(n^2)}$

Example 2: Consider the following recurrence relation : $\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + n}$

Solve the recurrence by using the master theorem?

Solution:

For this recurrence, we have $\boldsymbol{a = 2, b = 2}$, and $\boldsymbol{f(n) = n = \mathcal{O}(n^1) \implies c = 1}$

Since $\boldsymbol{a = b^c}$, $\boldsymbol{[2 = 2^1]}$

As per $\boldsymbol{case\ 2}$ of the master theorem, the solution is : $\boldsymbol{T(n) = \mathcal{O}(n^1\ log_2(n)) = \mathcal{O}(n \log\ n)}$

Example 3: Consider the following recurrence relation : $\boldsymbol{T(n) = 3T\left(\frac{n}{4}\right) + n^2}$

Solve the recurrence by using the master theorem?

Solution:

For this recurrence, we have $\boldsymbol{a = 3, b = 4}$, and $\boldsymbol{f(n) = n^2 = \mathcal{O}(n^2) \implies c = 2}$

Since $\boldsymbol{a < b^c}$, $\boldsymbol{[3 < 4^2]}$

As per $\boldsymbol{case\ 3}$ of the master theorem, the solution is : $\boldsymbol{T(n) = \mathcal{O}(n^2)}$

Example 4: Consider the following recurrence relation : $\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + n\ log\ n}$

Solve the recurrence by using the master theorem?

Solution:

For this recurrence, we have $\boldsymbol{a = 2, b = 2}$, and $\boldsymbol{f(n) = n\ log\ n = \mathcal{O}(n\ log\ n)}$

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 $\boldsymbol{T(n) = \mathcal{O}\left(n^c\ log^k(n)\right)}$, the following are true:

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

Example 1: Consider the following recurrence relation : $\boldsymbol{T(n) = 3T\left(\frac{n}{2}\right) + log_2\ n}$

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

Solution:

Here, the master theorem does not apply, but rather we apply the extended master theorem.

For this recurrence, we have $\boldsymbol{a = 3, b = 2}$, and $\boldsymbol{f(n) = \mathcal{O}(\ log\ n) \Longrightarrow c = 0\ \&\ k\ = 1}$

Since $\boldsymbol{a > b^c}$, $\boldsymbol{[3 > 2^0]}$

As per $\boldsymbol{case\ 1}$ of the extended master theorem**,** the solution is :

$$\boldsymbol{T(n) = \mathcal{O}\left(n^{log_b(a)}\right) = \mathcal{O}\left(n^{log_2(3)}\right)}$$

Example 2: Consider the following recurrence relation : $\boldsymbol{T(n) = 2T\left(\frac{n}{2}\right) + n\ log\ n}$

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

Solution:

Here, the master theorem does not apply, but rather we apply the extended master theorem.

For this recurrence, we have $\boldsymbol{a = 2, b = 2}$, and $\boldsymbol{f(n) = \mathcal{O}(n\ log\ n) \Longrightarrow c = 1\ \&\ k\ = 1}$

Since $\boldsymbol{a = b^c}$, $\boldsymbol{[2 = 2^1]}$

As per $\boldsymbol{case\ 2}$ of the extended master theorem, $\boldsymbol{k\ = 1\ \Longrightarrow k > -1,}$ the solution is :

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