Design and Analysis of Algorithms (CSC-314)

B.Sc. CSIT

Course Description

- This course introduces basic elements of the design and analysis of computer algorithms.
- Topics include asymptotic notations and analysis, divide and conquer strategy, greedy methods, dynamic programming, basic graph algorithms,NP-completeness, and approximation algorithms.
- For each topic, beside in-depth coverage, one or more representative problems and their algorithms shall be discussed

Objectives of the course

- Analyze the asymptotic performance of algorithms.
- Demonstrate a familiarity with major algorithm design techniques.
- Apply important algorithmic design paradigms and methods of analysis.
- Solve simple to moderately difficult algorithmic problems arising in applications.
- Able to demonstrate the hardness of simple NP-complete problems.

Prerequisites

- For learning this DAA, you should know the basic programming and mathematics concepts and data structure concepts.

- The basic knowledge of algorithms will also help you learn and understand the DAA concepts easily and quickly.

Text and Reference Books

- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, “Introduction to algorithms”, Third Edition.. The MIT Press, 2009.
- Ellis Horowitz, SartajSahni, SanguthevarRajasekiaran, “Computer Algorithms”, Second Edition, Silicon Press, 2007.
- Kleinberg, Jon, and Eva Tardos, “ Algorithm Design” , Addison-Wesley, First Edition, 2005

Unit-1: Foundation of Algorithm Analysis

- What are Algorithms?
- Why is the study of algorithms worthwhile?
- What is the role of algorithms relative to other technologies used in computers?

Unit-1: Foundation of Algorithm Analysis

What is an algorithm?

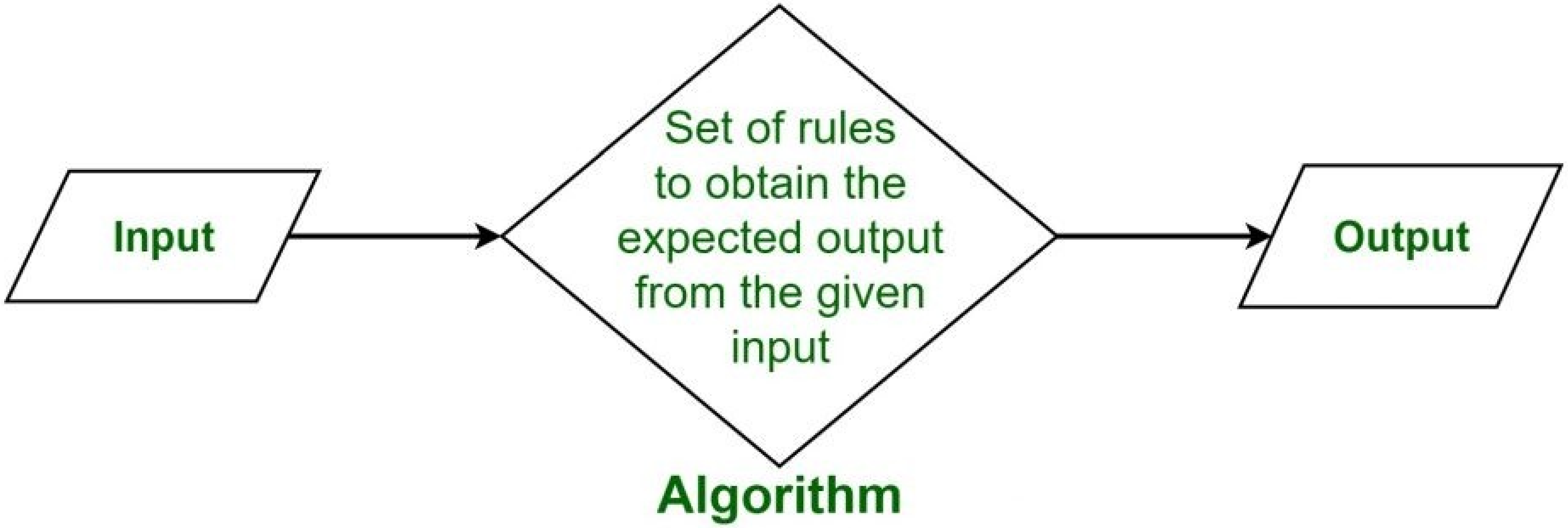
- Algorithm is a set of steps to complete a task.

For Example:

- Task: to make a cup of tea
- Algorithm:
  - add water and milk to the kettle,
  - boil it, add tea leaves,
  - Add sugar, and then serve it in cup.

Unit-1: Foundation of Algorithm Analysis

What is an algorithm?


Input
Set of rules to obtain the expected output from the given input
Output
Algorithm

Unit-1: Foundation of Algorithm Analysis

What is Algorithms?

- A set of steps to accomplish or complete a task that is described precisely enough that a computer can run it.
- Described precisely: very difficult for a machine to know how much water, milk to be added etc. in the above tea making algorithm.
- These algorithms run on computers or computational devices. For example, GPS in our smartphones, Google hangouts.
- GPS uses shortest path algorithm.

Unit-1: Foundation of Algorithm Analysis

What is Algorithms?

- An Algorithm is a set of well-defined instructions designed to perform a specific set of tasks.

- Algorithms are used in Computer science to perform calculations, automatic reasoning, data processing, computations, and problem-solving.

- Designing an algorithm is important before writing the program code as the algorithm explains the logic even before the code is developed.

Unit-1: Foundation of Algorithm Analysis

What is Algorithms?

- Informally, an algorithm is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output.
- An algorithm is thus a sequence of computational steps that transform the input into the output.
- We can also view an algorithm as a tool for solving a well-specified computational problem.
- The statement of the problem specifies in general terms the desired input/output relationship.
- The algorithm describes a specific computational procedure for achieving that input/output relationship.

Unit-1: Foundation of Algorithm Analysis

What is Algorithms?

- For example, we might need to sort a sequence of numbers into non decreasing order.
- This problem arises frequently in practice and provides fertile ground for introducing many standard design techniques and analysis tools.
- Here is how we formally define the sorting problem:
  - Input: A sequence of n numbers (a_1 , a_2, …. , a_n)
  - Output: A permutation (reordering) (a'_1 , a'_2 , … , a'_n) of the input sequence such that ($a'_1 \le a'_2 \le \ldots \le a'_n$)

Unit-1: Foundation of Algorithm Analysis

What is Algorithms?

- For example, given the
  - Input sequence (31, 41, 59, 26, 41, 58), a sorting algorithm returns as output the sequence (26, 31, 41, 41, 58, 59)
- Such an input sequence is called an instance of the sorting problem.
- In general, an instance of a problem consists of the input needed to compute a solution to the problem.

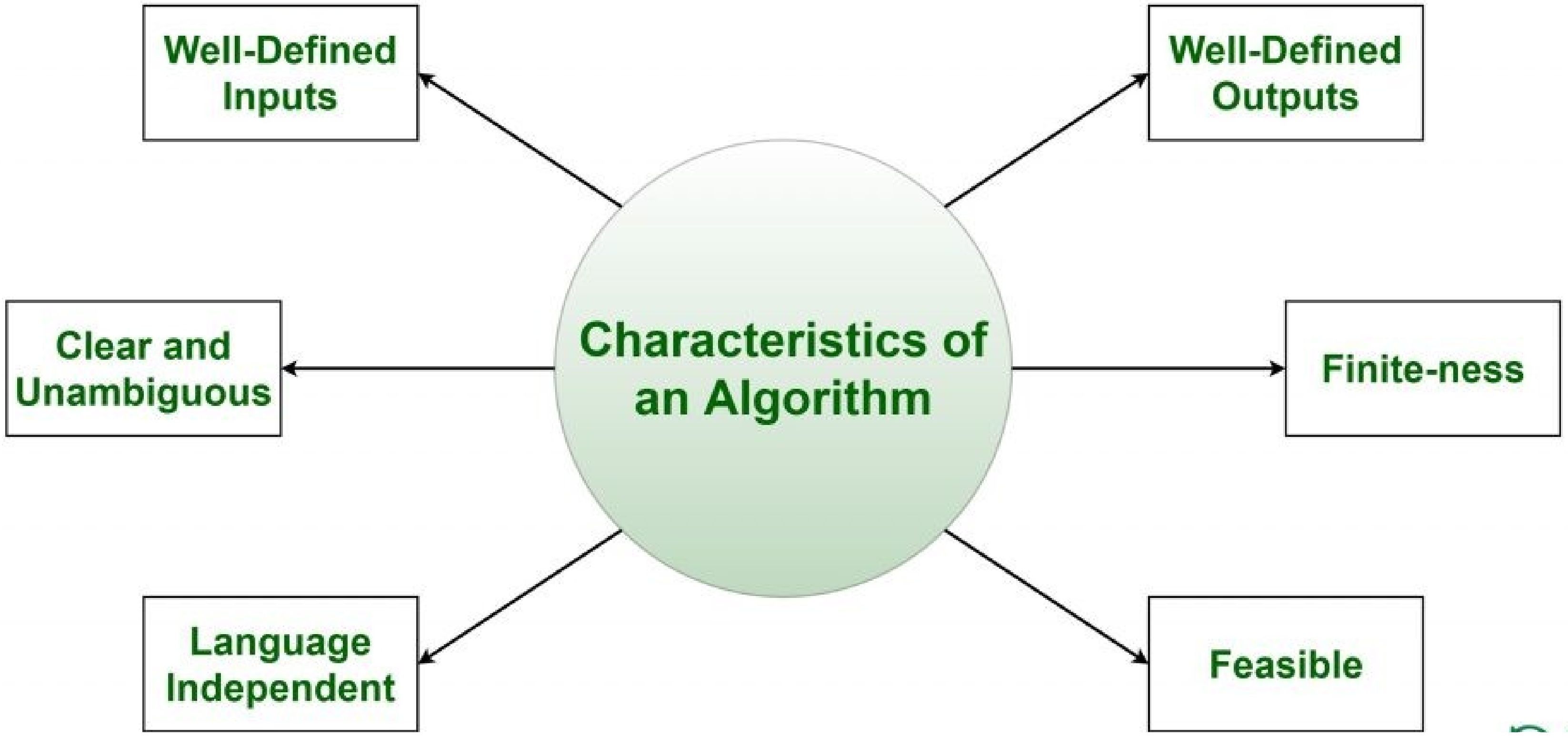
Unit-1: Foundation of Algorithm Analysis

What is Algorithms?

- An algorithm is said to be correct if, for every input instance, it halts with the correct output.
- We say that a correct algorithm solves the given computational problem.
- An incorrect algorithm might not halt at all on some input instances, or it might halt with an incorrect answer.

Unit-1: Foundation of Algorithm Analysis

Properties of Algorithms:



Unit-1: Foundation of Algorithm Analysis

Properties of Algorithms:

- **Input(s)/output(s)**: There must be some inputs from the standard set of inputs and an algorithm’s execution must produce outputs(s).
- **Definiteness**: Each step must be clear and unambiguous.
- **Finiteness:** Algorithms must terminate after finite time or steps.
- **Correctness:** Correct set of output values must be produced from the each set of inputs.
- **Effectiveness:** Each step must be carried out in finite time.

Unit-1: Foundation of Algorithm Analysis

Properties of Algorithms:

- **Feasibility**: It must be feasible to execute each individual instructions.
- **Efficient:** Efficiency is always measured in terms of time and space requires implementing the algorithm, so efficient algorithm uses the minimal running time and memory space as possible.
- **Independent:** Independent of language**.** Such that, an algorithm should focus only on what are inputs, outputs and how to derive output.

Unit-1: Foundation of Algorithm Analysis

Need of Algorithms:

- To understand the basic idea of a problem.
- To find the best approach to solve the problem.
- To improve the efficiency of existing techniques.
- To understand the basic principles of designing the algorithms.
- And so on…!!!

Unit-1: Foundation of Algorithm Analysis

Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

Unit-1: Foundation of Algorithm Analysis

How to Design an Algorithm?

In order to write an algorithm, following things are needed as a prerequisites:

- **The problem that is to be solved by this algorithm.**
- **The constraints of the problem that must be considered while solving the problem.**
- **The input to be taken to solve the problem.**
- **The output to be expected when the problem the is solved.**
- **The solution to this problem, in the given constraints.**

Then, the algorithm is written with the help of above parameters such that it solves the problem.

Unit-1: Foundation of Algorithm Analysis

Algorithm analysis

- It involves calculating the complexity of algorithms, usually either the time-complexity or the space-complexity.
- Two common tools used for algorithm analysis are:
  - RAM model of computation and
  - Asymptotic analysis of worst-case complexity

Unit-1: Foundation of Algorithm Analysis

RAM Model:

- Before we can analyze an algorithm, we must have a model of the implementation technology that we will use, including a model for the resources of that technology and their costs.
- The generic processor random-access machine (RAM) model of computation as our implementation technology and understand that our algorithms will be implemented as computer programs.
- In the RAM model, instructions are executed one after another, with no concurrent operations.

Unit-1: Foundation of Algorithm Analysis

RAM Model:

- The RAM (Random Access Machine) model of computation measures the run time of an algorithm by summing up the number of steps needed to execute the algorithm on a set of data.
- The RAM model contains instructions commonly found in real computers: arithmetic (such as add, subtract, multiply, divide, remainder), data movement (load, store, copy), and control (conditional and unconditional branch, subroutine call and return).
- Each such instruction takes a constant amount of time.
- The data types in the RAM model are integer and floating point (for storing real numbers).
- In the RAM model, we do not attempt to model the memory hierarchy that is common in contemporary computers. That is, we do not model caches or virtual memory.

Unit-1: Foundation of Algorithm Analysis

RAM Model:

- This model encapsulates the core functionality of computers but does not mimic them completely.
- For example, an addition operation and a multiplication operation are both worth a single time step, however, in reality it will take a machine more operations to compute a product versus a sum.
- The reason the RAM model makes these assumptions is because doing so allows a balance between simplicity and completely imitating underlying machine, resulting in a tool that is useful in practice.
- The exact analysis of algorithms is a difficult task. It is the nature of algorithm analysis to be both machine and language independent. For example, if your computer becomes twice as fast after a recent update, the complexity of your algorithm still remains the same.

Unit-1: Foundation of Algorithm Analysis

RAM Model:

- This model encapsulates the core functionality of computers but does not mimic them completely.
- For example, an addition operation and a multiplication operation are both worth a single time step, however, in reality it will take a machine more operations to compute a product versus a sum.
- The reason the RAM model makes these assumptions is because doing so allows a balance between simplicity and completely imitating underlying machine, resulting in a tool that is useful in practice.
- The exact analysis of algorithms is a difficult task. It is the nature of algorithm analysis to be both machine and language independent. For example, if your computer becomes twice as fast after a recent update, the complexity of your algorithm still remains the same.

Unit-1: Foundation of Algorithm Analysis

Unit-1: Foundation of Algorithm Analysis

Time and Space Complexity

- We measure an algorithm's efficiency using the time and space (memory) it takes for execution.
- Time is important because we need our programs to run as fast as possible to deliver the results quickly.
- Space is important because machines have only a limited amount of space to spare for programs.
- The best algorithm is the one that completes its execution in the least amount of time using the least amount of space.
- But often, in reality, algorithms have to tradeoff between saving space or time.
- That's why the best algorithm for a given task is not something that's fixed in stone. The best algorithm depends on our requirements.
- If we need our algorithm to run as fast as possible despite the memory usage, we can pick the most time-efficient algorithm as the best algorithm and vice versa.

Unit-1: Foundation of Algorithm Analysis

Time and Space Complexity

- Time Complexity: The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.
- Space Complexity: The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input.

Unit-1: Foundation of Algorithm Analysis

- Time and Space Complexity depends on lots of things like machine hardware, operating system, etc.
- However, we do not consider any of these factor while analyzing the algorithm. We will only consider the execution time of an algorithm. The time complexity of an algorithm is commonly expressed using asymptotic notations as:
  - Big O -O(n)
  - Big Theta -Ɵ(n)
  - Big Omega -Ω(n)

Unit-1: Foundation of Algorithm Analysis

Best, Worst, and Average Cases

- Usually, in asymptotic analysis, we consider three cases when analyzing an algorithm:
  - best,
  - worst, and
  - average.

Unit-1: Foundation of Algorithm Analysis

Best Case

- We consider the combination of inputs that allows the algorithm to complete its execution in the minimum amount of time as the best-case scenario in terms of time complexity. The execution time in this case acts as a lower bound to the time complexity of the algorithm.
- In linear search, the best-case scenario occurs when k is stored at the 0th index of the list. In this case, the algorithm can complete execution after only one iteration of the for loop.
  - nums = [1, 2, 3, 4, 5, 6]
  - n = 6
  - k = 1

Unit-1: Foundation of Algorithm Analysis

Worst case

- Worst case scenario occurs when the combination of inputs that takes the maximum amount of time for completion is passed to the algorithm. The execution time of the worst case acts as an upper bound to the time complexity of the algorithm.
- In linear search, the worst case occurs when k is not present in the list. This takes the algorithm n+1 iterations to figure out that the number is not in the list.
  - nums = [1, 2, 3, 4, 5, 6]
  - n = 6
  - k = 7

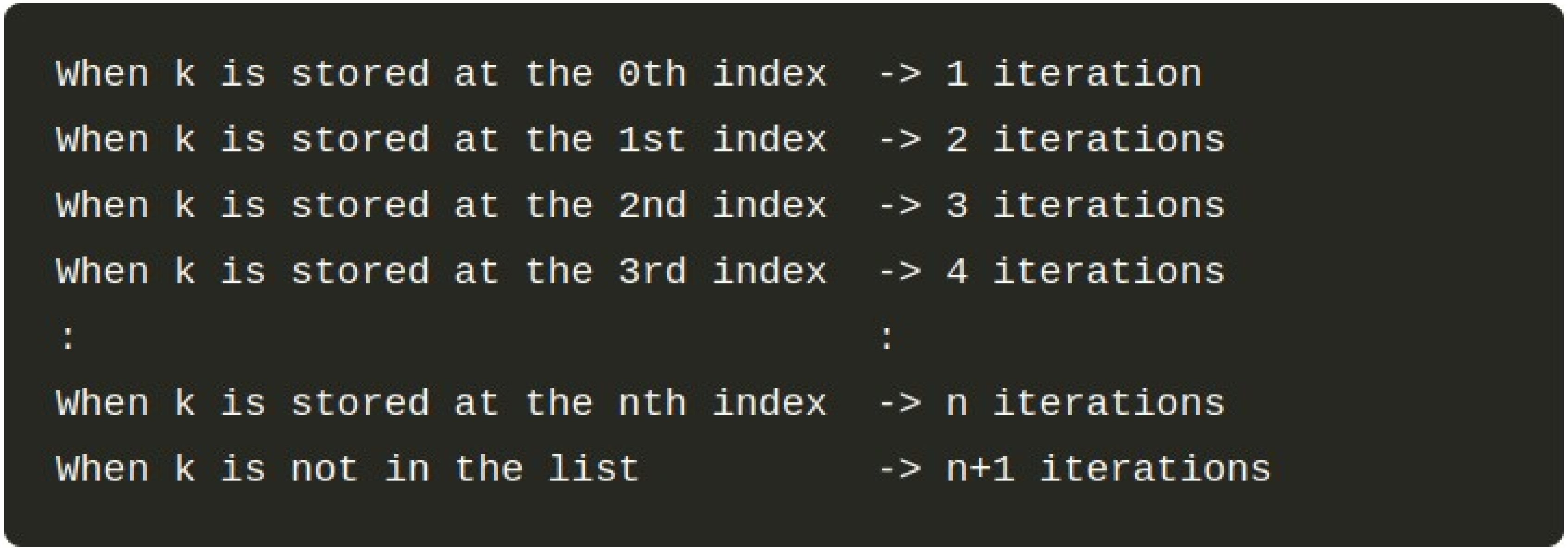
Unit-1: Foundation of Algorithm Analysis

Average Case

- To find the average case, we get the sum of running times of the algorithm for every possible input combination and take their average.
- In linear search, the number of iterations the algorithm takes to complete execution follows this pattern.
  - nums = [1, 2, 3, 4, 5, 6]
  - n = 6

Unit-1: Foundation of Algorithm Analysis

Average Case



```
When k is stored at the 0th index  -> 1 iteration
When k is stored at the 1st index  -> 2 iterations
When k is stored at the 2nd index  -> 3 iterations
When k is stored at the 3rd index  -> 4 iterations
:                                  :
When k is stored at the nth index  -> n iterations
When k is not in the list          -> n+1 iterations
```

- So, we can calculate the average running time of the algorithm this way.

$$\frac{\sum_{i=1}^{n+1} i}{n+1} = \frac{(n+1)*(n+2)/2}{n+1} = n/2+1$$

Unit-1: Foundation of Algorithm Analysis

Examples of Time complexity:

```
for( i=0; i<n; i++)      -n+1 times
{
    .....
    statements;                 - n times
}
```

- Here, we are only concerned about the execution of statements, so Time complexity here is **O(n)**.

Unit-1: Foundation of Algorithm Analysis

Examples of Time complexity:

```
for( i=0; i<n; i--)        -n+1 times
{
    …..
    statements;                - n times
}
```

- Here, we are only concerned about the execution of statements, so Time complexity here is **O(n)**.

Unit-1: Foundation of Algorithm Analysis

Examples of Time complexity:

```
for( i=0; i<n; i+2)
{
    …..
    statements;              - n/2 times
}
```

- Here, we are only concerned about the execution of statements, so Time complexity here is **O(n)**.

Unit-1: Foundation of Algorithm Analysis

Examples of Time complexity:

```
for( i=0; i<n; i++)            - n+1
{
    for(j=0; j<n; j++)            -n(n+1)
    {
    …..
    statements;          - n*n times
    }
}
```

- Here, we are only concerned about the execution of statements, so Time complexity here is roughly **O(n^2)**.

Unit-1: Foundation of Algorithm Analysis

Examples of Time complexity:

```
for( i=0; i<n; i++)
{
    for(j=0; j<i; j++)
    {
    …..
    statements;
    }
}
```

i	j	No of times
0	0*	0
1	0	1
	1*	
2	0 1 2*	2
3	0 1 2 3*	3
..		
n	n	n

- T(n)= 0+1+2+3+…..+n

  i.e. n(n+1)/2

- Here, we are only concerned about the execution of statements, so Time complexity here is roughly **O(n^2)**.

Unit-1: Foundation of Algorithm Analysis

Examples of Time complexity:

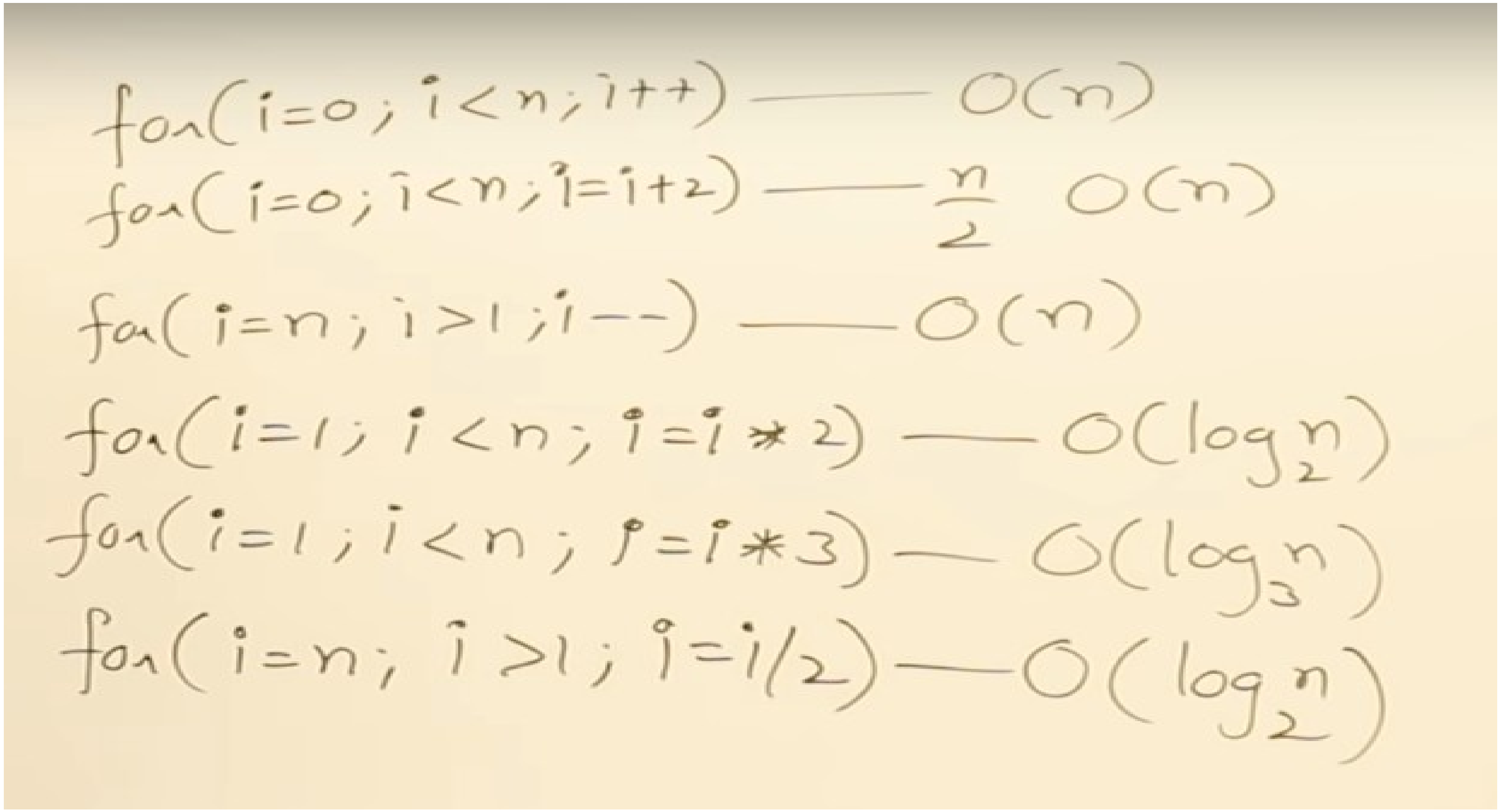
```
for( i=0; i<n; i*2)
{   .....
    statements;
}
```

Initially: i=1
$1*2=2$
$2*2=2^2$
$2^2*2=2^3$
..
....
2^k

- Assuming stopping condition, i>=n
  - Here, i= 2^k
  - i.e. $2^k >= n$
  - So, $2^k = n$
  - $K = \log_2 n$
- Here, we are only concerned about the execution of statements, so Time complexity here is roughly **O(**$\log_2 n$**)**.

Unit-1: Foundation of Algorithm Analysis

Summary

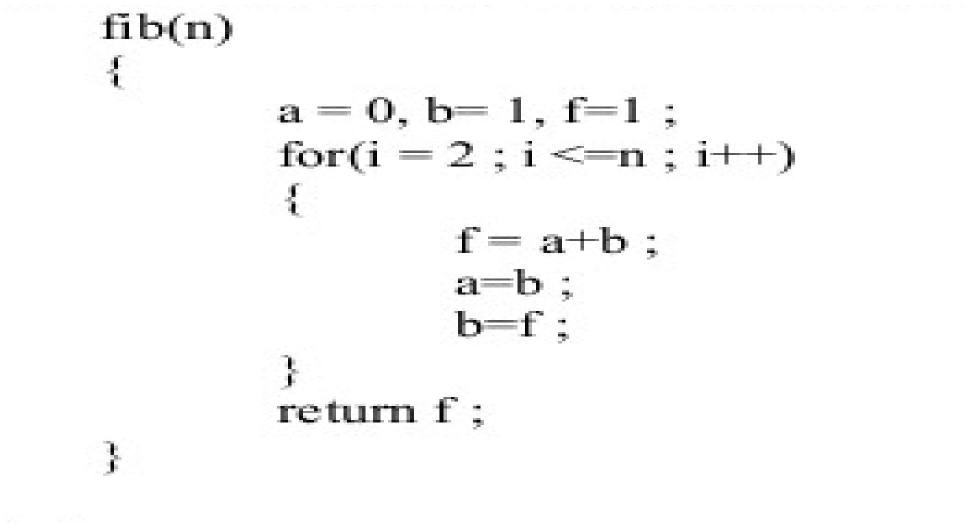


`for(i=0; i<n; i++)`	—	$O(n)$
`for(i=0; i<n; i=i+2)`	—	$\frac{n}{2}$ $O(n)$
`for(i=n; i>1; i--)`	—	$O(n)$
`for(i=1; i<n; i=i*2)`	—	$O(\log_2 n)$
`for(i=1; i<n; i=i*3)`	—	$O(\log_3 n)$
`for(i=n; i>1; i=i/2)`	—	$O(\log_2 n)$

Unit-1: Foundation of Algorithm Analysis

Example : Fibonacci Series

- Input: n
- Output: nth Fibonacci number.
- Algorithm: assume a as first(previous) and b as second(current) numbers

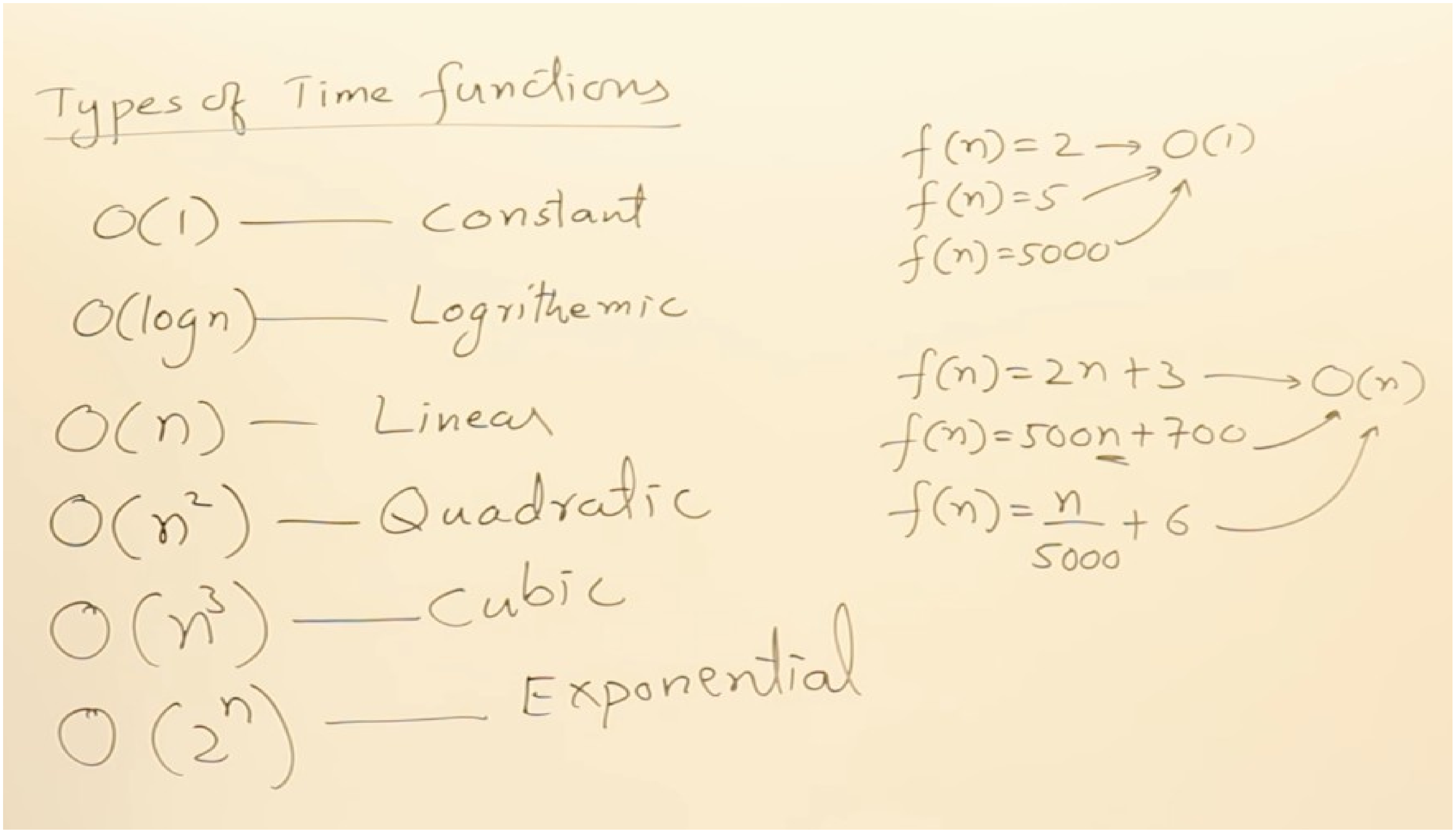


```
fib(n)
{
        a = 0, b= 1, f=1 ;
        for(i = 2 ; i <=n ; i++)
        {
                f= a+b ;
                a=b ;
                b=f ;
        }
        return f ;
}
```

- Efficiency
  - Time Complexity: The algorithm above iterates up to n-2 times, so time complexity is O(n).
  - Space Complexity: The space complexity is constant i.e. O(1).

Unit-1: Foundation of Algorithm Analysis

Types of Functions



Types of Time functions

$O(1)$ —— constant

$O(\log n)$ —— Logrithemic

$O(n)$ — Linear

$O(n^2)$ — Quadratic

$O(n^3)$ —— cubic

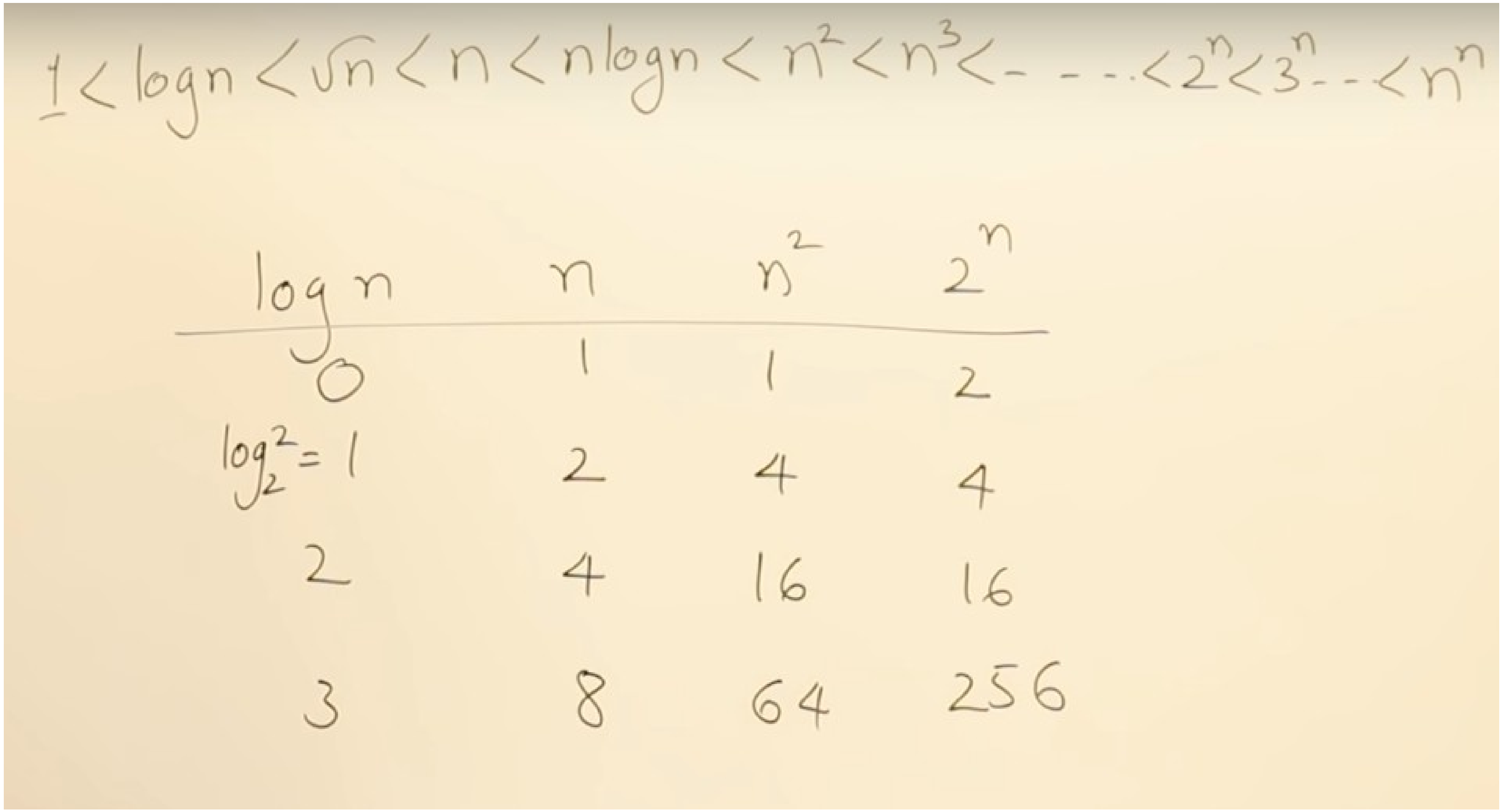
$O(2^n)$ —— Exponential

$$\begin{tikzcd}
f(n)=2 \arrow[r] & O(1) \\
f(n)=5 \arrow[ur] & \\
f(n)=5000 \arrow[uur] &
\end{tikzcd}$$

$$\begin{tikzcd}
f(n)=2n+3 \arrow[r] & O(n) \\
f(n)=500\underline{n}+700 \arrow[ur] & \\
f(n)=\frac{n}{5000}+6 \arrow[uur] &
\end{tikzcd}$$

Unit-1: Foundation of Algorithm Analysis

Comparison of Functions



$1 < \log n < \sqrt{n} < n < n\log n < n^2 < n^3 < \dots < 2^n < 3^n \dots < n^n$

$\log n$	n	n^2	2^n
0	1	1	2
$\log_2 2 = 1$	2	4	4
2	4	16	16
3	8	64	256

Unit-1: Foundation of Algorithm Analysis

Asymptotic Notations:

- In mathematics, asymptotic analysis, also known as asymptotics, is a method of describing the limiting behavior of a function.
- In computing, asymptotic analysis of an algorithm refers to defining the mathematical boundation of its run-time performance based on the input size.
- For example, the running time of one operation is computed as f(n), and maybe for another operation, it is computed as g(n2).
- This means the first operation running time will increase linearly with the increase in n and the running time of the second operation will increase exponentially when n increases.
- Similarly, the running time of both operations will be nearly the same if n is small in value.

Unit-1: Foundation of Algorithm Analysis

Asymptotic Notations:

- Complexity analysis of an algorithm is very hard if we try to analyze exact.
- we know that the complexity (worst, best, or average) of an algorithm is the mathematical function of the size of the input.
- So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier.
- For this purpose we need the concept of asymptotic notations.
  - Best Case (Omega Notation)
  - Average Case (Theta Notation)
  - Worst Case (O Notation)

Unit-1: Foundation of Algorithm Analysis

Big-O Notations:

- The Big-O notation describes the worst-case running time of a program.
- We compute the Big-O of an algorithm by counting how many iterations an algorithm will take in the worst-case scenario with an input of N.
- We typically consult the Big-O because we must always plan for the worst case.

Unit-1: Foundation of Algorithm Analysis

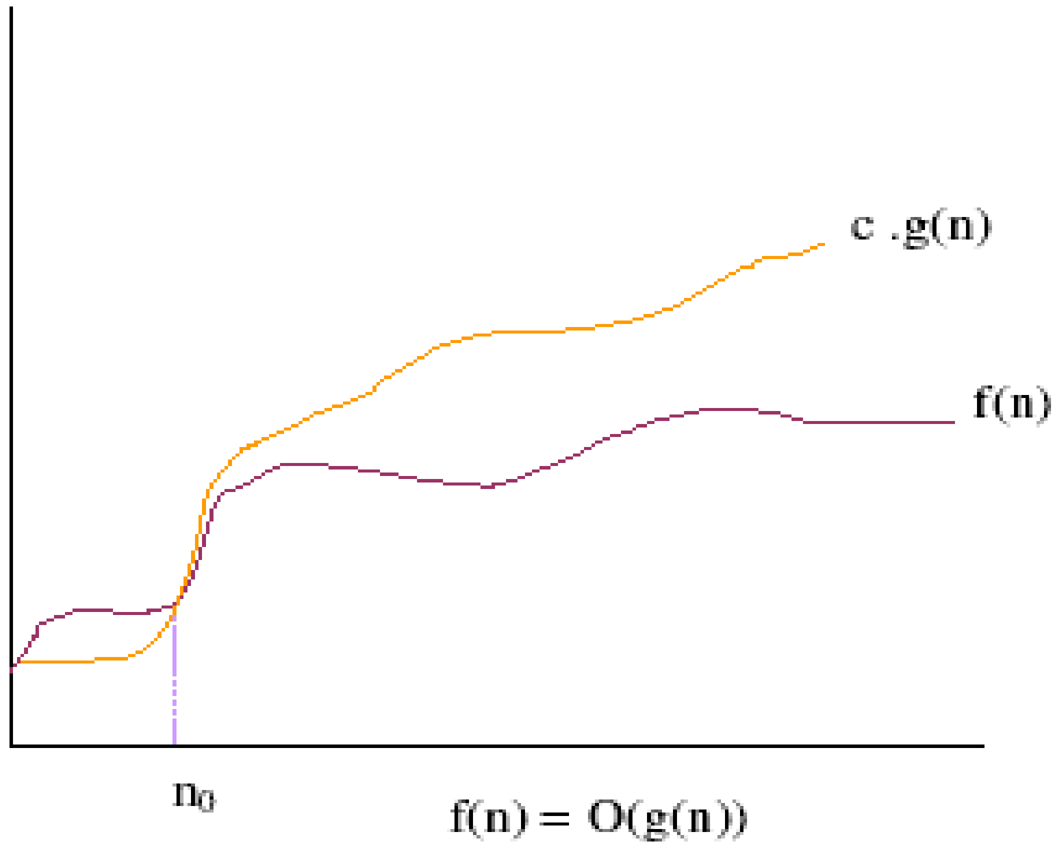
Big-O Notations:

- When we have only asymptotic upper bound then we use O notation.
- A function f(x)=O(g(x)) (read as f(x) is big oh of g(x)) **iff there exists two positive constants c and x_0** such that for all x >=x_0 , 0 <= f(x) <= c*g(x)
- The above relation says that g(x) is an upper bound of f(x).

Unit-1: Foundation of Algorithm Analysis

Big-O Notations:

- For all values of n >= n 0 , plot shows clearly that f(n) lies below or on the curve of c*g(n)



Big-O Notations:

Example:

- Given, $f(n) = 3n^2 + 4n + 7$ and $g(n) = n^2$, then prove that $f(n) = O(g(n))$.
  - **Proof:** let us choose c and n_0 values as 14 and 1 respectively then we can have
    - $f(n) <= c*g(n), n>=n_0$ as
    - $3n^2 + 4n + 7 <= 14*n^2$ for all $n >= 1$
- The above inequality is true. Hence $f(n) = O(g(n))$

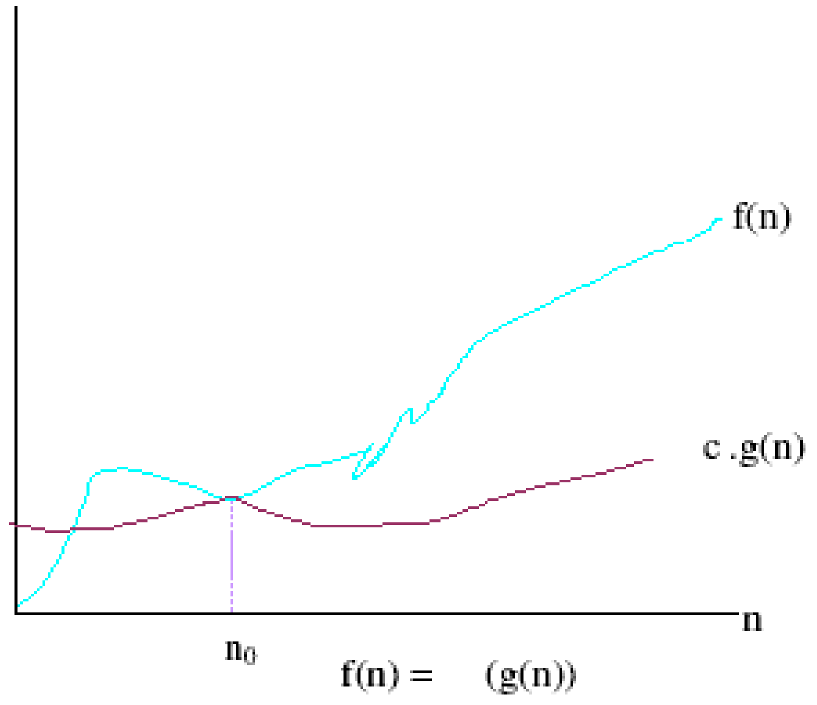
Unit-1: Foundation of Algorithm Analysis

Big-Ω Notation:

- Big-Ω (Omega) describes the best running time of a program.
- We compute the big-Ω by counting how many iterations an algorithm will take in the best-case scenario based on an input of N.
- Big omega notation gives asymptotic lower bound. A function f(x) = Ω (g(x)) (read as g(x) is big omega of g(x)) iff there exists two positive constants c and x 0 such that for all x >= x_0 , 0 <= c*g(x)<= f(x).
- The above relation says that g(x) is a lower bound of f(x).

Unit-1: Foundation of Algorithm Analysis

Big-Ω Notation:



For all values of n >= n_0, plot shows clearly that f(n) lies above or on the curve of c*g(n).

Unit-1: Foundation of Algorithm Analysis

Big-Ω Notation:

- **Example:** $f(n) = 3n^2 + 4n + 7$ and $g(n) = n^2$, then prove that $f(n) = \Omega(g(n))$.
- **Proof:** let us choose c and n_0 values as 1 and 1, respectively then we can have
  - $f(n) >= c*g(n)$, $n>=n_0$ as
  - $3n^2 + 4n + 7 >= 1*n^2$ for all $n >= 1$
- The above inequality is trivially true. Hence $f(n) = \Omega(g(n))$

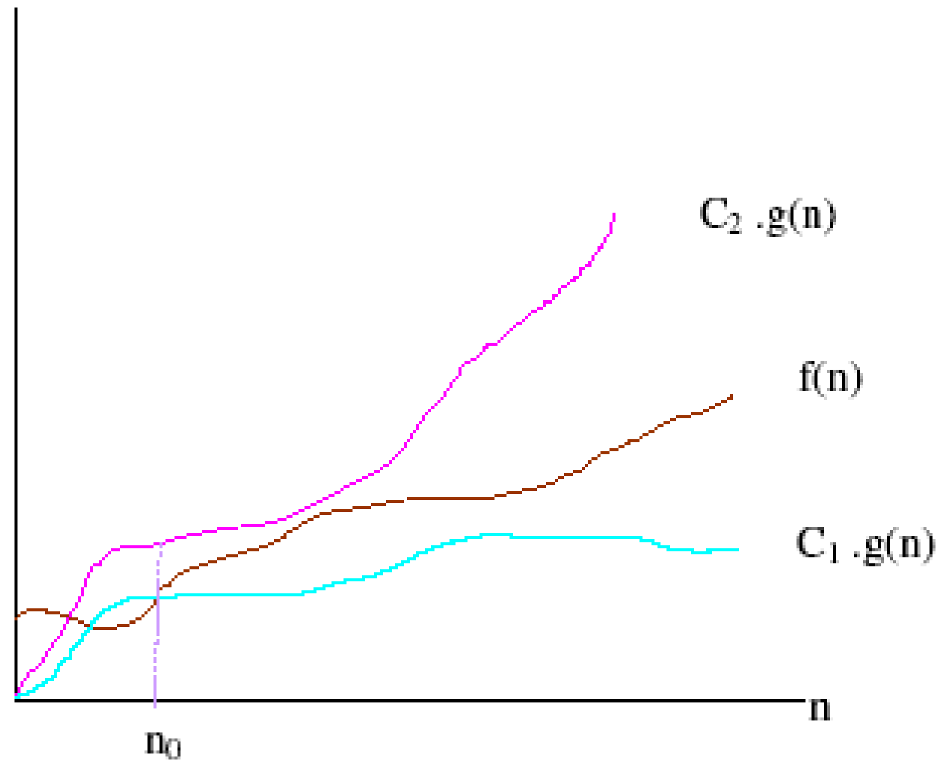
Unit-1: Foundation of Algorithm Analysis

Big-Θ Notation:

- We compute the big-Θ of an algorithm by counting the number of iterations the algorithm always takes with an input of n.
- When we need asymptotically tight bound then we use notation.
- A function f(x) = (g(x)) (read as f(x) is big theta of g(x)) iff there exists three positive constants c_1 , c_2 and x_0 such that for all $x >= x_0$, $c_1 *g(x) <= f(x) <= c_2 *g(x)$
- The above relation says that f(x) is order of g(x).

Unit-1: Foundation of Algorithm Analysis

Big-Θ Notation:



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

For all values of $n >= n_0$, plot shows clearly that f(n) lies between c_1* g(n)and c_2*g(n).

Unit-1: Foundation of Algorithm Analysis

Big-Θ Notation:

- Example :f(n) = $3n^2 + 4n + 7$ and g(n) = n^2 , then prove that f(n) = (g(n)).
- **Proof:** let us choose c_1 , c_2 and n_o values as 14, 1 and 1 respectively then we can have,
  - f(n) <= c_1 *g(n), n>=n_0 as $3n^2 + 4n + 7 <= 14*n^2$, and
  - f(n) >= c_2 *g(n), n>=n_0 as $3n^2 + 4n + 7 >= 1*n^2$ for all n >= 1(in both cases).
- So, c_2 *g(n) <= f(n) <= c_1 *g(n) is trivial.
- Hence f(n) = Θ (g(n)).

Unit-1: Foundation of Algorithm Analysis

Mathematical Foundation

Multiplication Rule	$a^x \times a^y = a^{x+y}$
Division Rule	$a^x \div a^y = a^{x-y}$
Power of a Power Rule	$(a^x)^y = a^{xy}$
Power of a Product Rule	$(ab)^x = a^x b^x$
Power of a Fraction Rule	$\left(\frac{a}{b}\right)^x = \frac{a^x}{b^x}$
Zero Exponent	$a^0 = 1$
Negative Exponent	$a^{-x} = \frac{1}{a^x}$
Fractional Exponent	$a^{\frac{x}{y}} = \sqrt[y]{a^x}$

Unit-1: Foundation of Algorithm Analysis

Mathematical Foundation:

(1) $\log_a a = 1, \ \log_a 1 = 0$

(2) $\log_a b . \log_b a = 1 \Rightarrow \log_a b = \dfrac{1}{\log_b a}$

(3) $\log_c a = \log_b a . \ \log_c b$ or $\log_c a = \dfrac{\log_b a}{\log_b c}$

(4) $\log_a(mn) = \log_a m + \log_a n$

(5) $\log_a \left(\dfrac{m}{n} \right) = \log_a m - \log_a n$

(6) $\log_a m^n = n \log_a m$ (7) $a^{\log_a m} = m$

(8) $\log_a \left(\dfrac{1}{n} \right) = -\log_a n$ (9) $\log_{a^\beta} n = \dfrac{1}{\beta} \log_a n$

(10) $\log_{a^\beta} n^\alpha = \dfrac{\alpha}{\beta} \log_a n \ , (\beta \neq 0)$

(11) $a^{\log_c b} = b^{\log_c a} \ , \ (a, b, c > 0$ and $c \neq 1)$

Unit-1: Foundation of Algorithm Analysis

What is Recursion?

- The process in which a function calls itself directly or indirectly is called recursion and the corresponding function is called as recursive function.

- Using recursive algorithm, certain problems can be solved quite easily. Examples of such problems are Towers of Hanoi (TOH), Inorder/Preorder/Postorder Tree Traversals, DFS of Graph, etc.

Unit-1: Foundation of Algorithm Analysis

What is Recursion?

- Let us consider a problem that a programmer have to determine the sum of first n natural numbers, there are several ways of doing that but the simplest approach is simply add the numbers starting from 1 to n. So the function simply looks like,
- approach(1) – Simply adding one by o
  - f(n) = 1 + 2 + 3 +……..+ n
- approach(2) – Recursive adding
  - f(n) = 1 n=1
  - f(n) = n + f(n-1) n>1

Unit-1: Foundation of Algorithm Analysis

What is recursive algorithm?

- A recursive algorithm is an algorithm which calls itself with "smaller (or simpler)" input values, and which obtains the result for the current input by applying simple operations to the returned value for the smaller (or simpler) input.
- More generally if a problem can be solved utilizing solutions to smaller versions of the same problem, and the smaller versions reduce to easily solvable cases, then one can use a recursive algorithm to solve that problem.

Unit-1: Foundation of Algorithm Analysis

What is recursive algorithm?

- Example: Factorial
  - n! = 1*2*3...*n and 0! = 1 (called initial case)
  - So the recursive defintiion n! = n*(n-1)!

- Algorithm F(n):
  - if n = 0 then return 1 // base case
  - else F(n-1)*n // recursive call

Unit-1: Foundation of Algorithm Analysis

What is recurrence relation?

- A recurrence relation is an equation that recursively defines a sequence where the next term is a function of the previous terms (Expressing F_n as some combination of F_i with $i<n$).
- Example −
  - Fibonacci series − $F_n = F_{n-1} + F_{n-2}$

Unit-1: Foundation of Algorithm Analysis

Solving recurrence relation?

- Recursion Tree Method,
- Substitution Method,
- Masters Method

Unit-1: Foundation of Algorithm Analysis

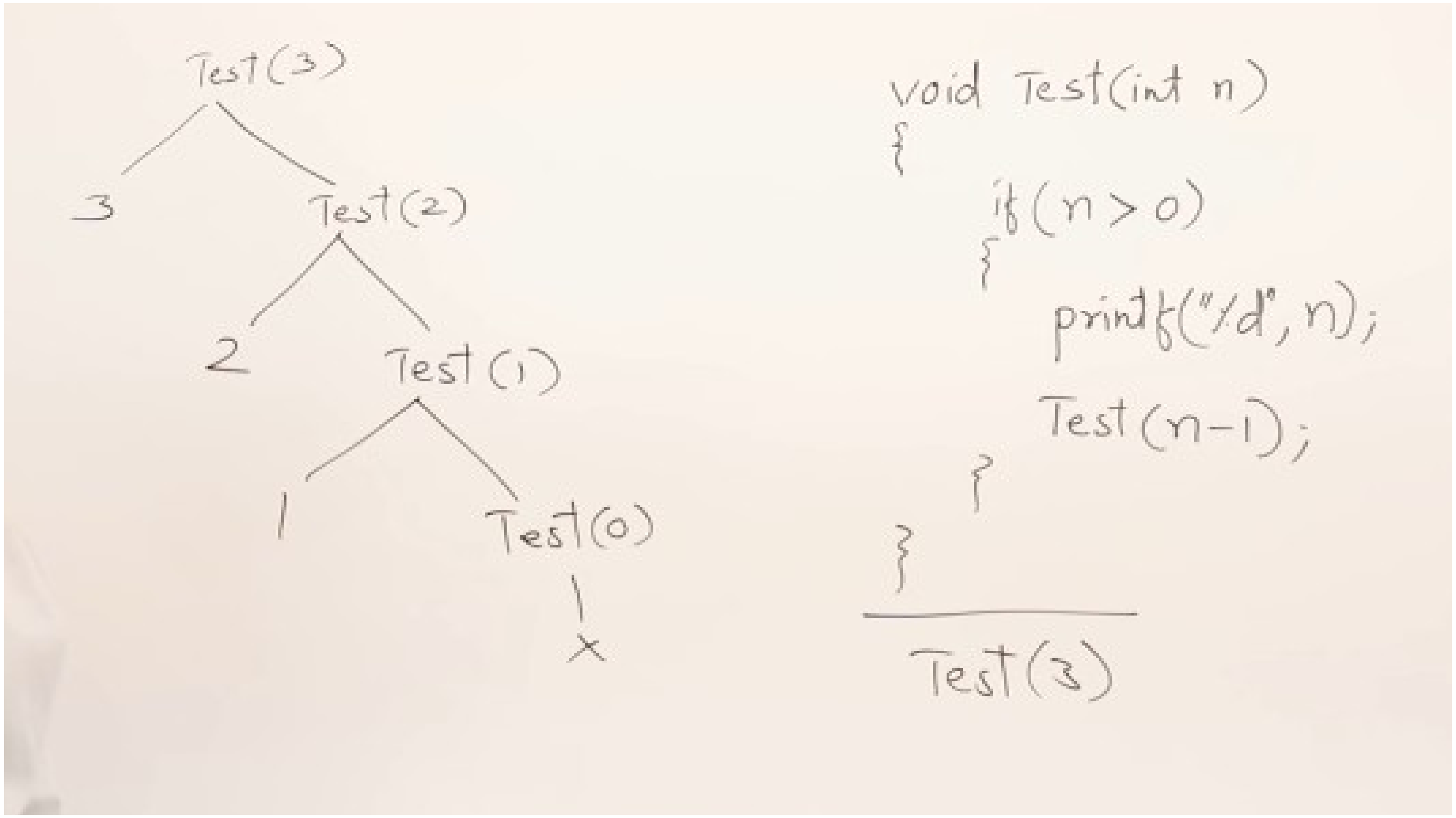
Recursion Tree Method

- This methods is a pictorial representation of every iteration which is in the form of tree where each level nodes are expanded.
- In recursion tree each root and child represent the cost of a single sub program.
- We sum the costs within each of the levels of the tree to obtain a set of pre-level cost and sum all pre-level cost to determine the total cost of all level of recursion.

Unit-1: Foundation of Algorithm Analysis

Recursion Tree Method

- Examples:



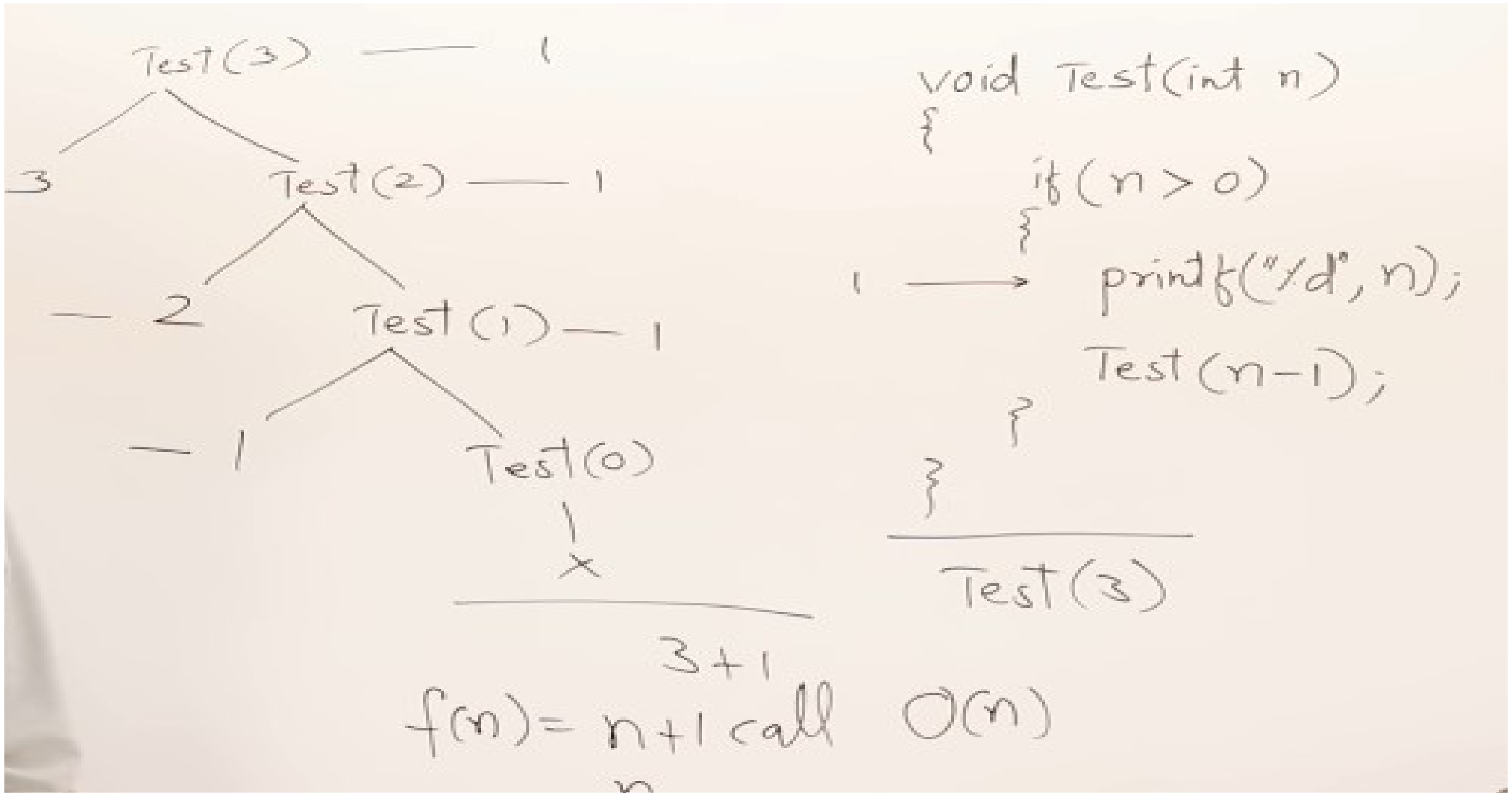
```
void Test(int n)
{
    if(n>0)
    {
        printf("%d",n);
        Test(n-1);
    }
}
```

Test(3)

Unit-1: Foundation of Algorithm Analysis

Recursion Tree Method

- Examples:

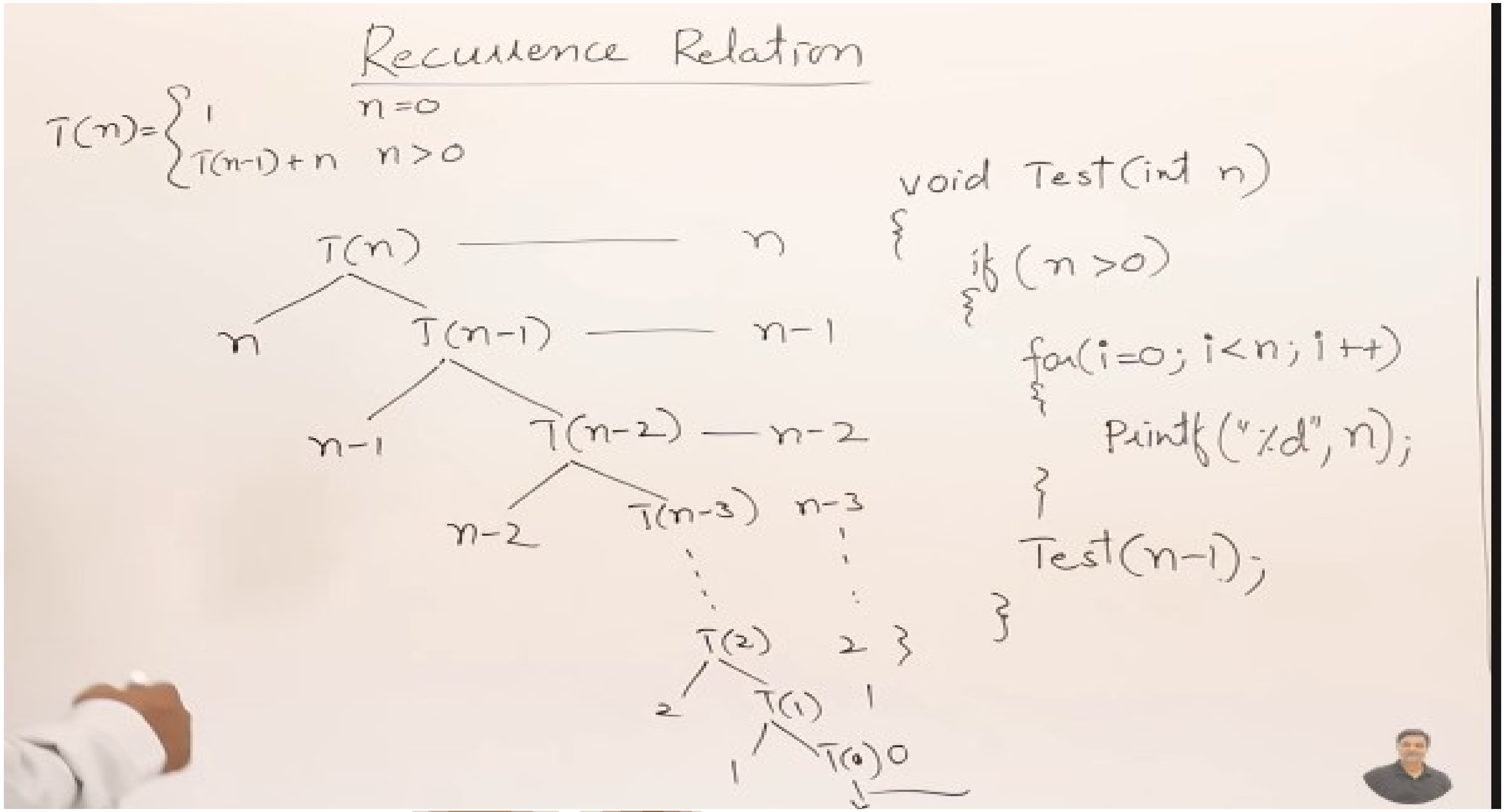


```
void Test(int n)
{
    if(n>0)
    {
        printf("%d", n);
        Test(n-1);
    }
}
```

Unit-1: Foundation of Algorithm Analysis

Recursion Tree Method

- Examples:



Recurrence Relation

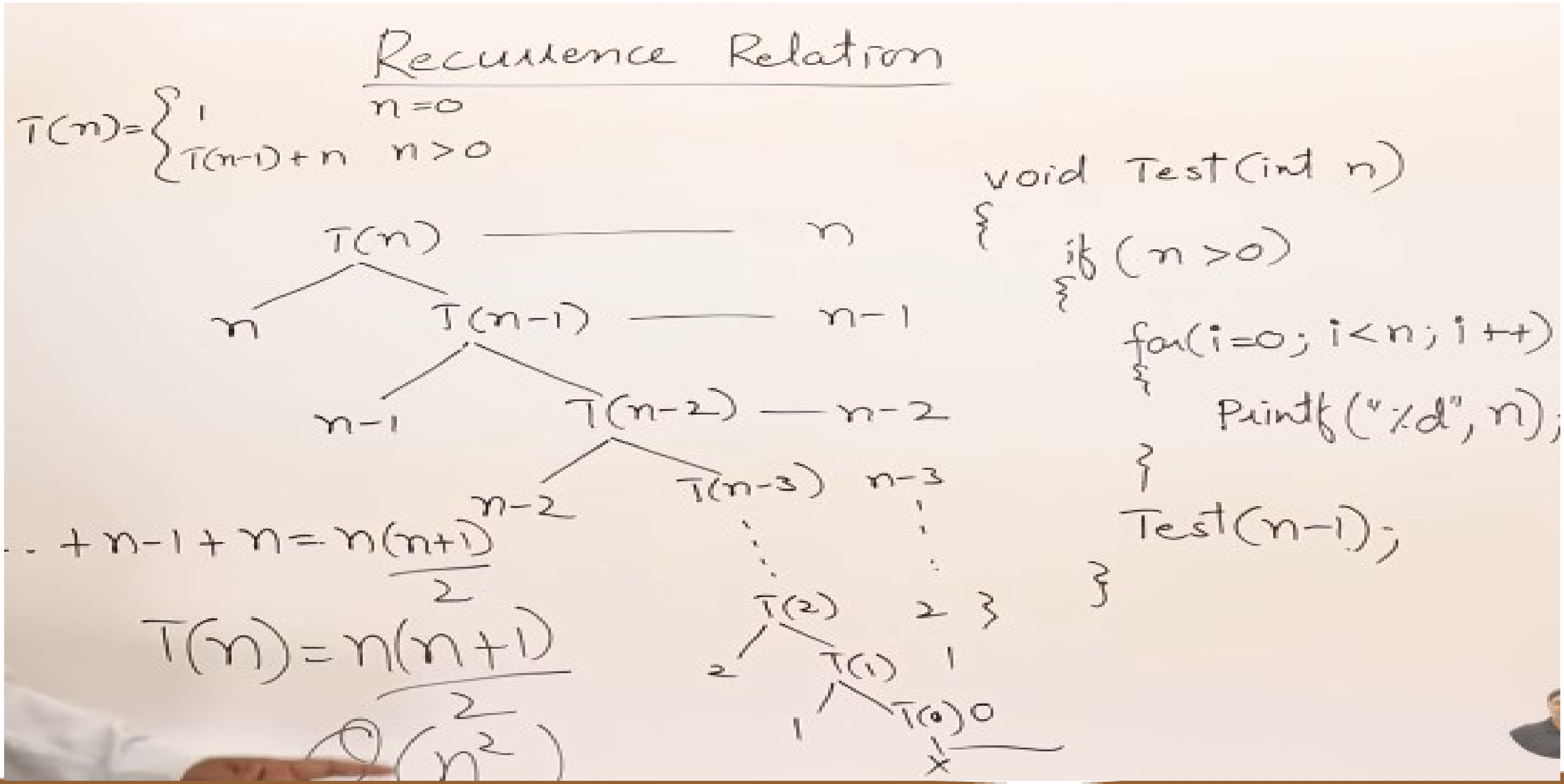
$$T(n) = \begin{cases} 1 & n = 0 \\ T(n-1) + n & n > 0 \end{cases}$$

```
void Test(int n)
{
    if(n>0)
    {
        for(i=0; i<n; i++)
        {
            Printf("%d", n);
        }
        Test(n-1);
    }
}
```

Unit-1: Foundation of Algorithm Analysis

Recursion Tree Method

- Examples:
- $O+1+2+3+\ldots\ldots+(n-1)+n = n(n+1)/2 = O(n^2)$


Recurrence Relation

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

```
void Test(int n)
{
    if (n>0)
    {
        for(i=0; i<n; i++)
        {
            Printf("%d", n);
        }
        Test(n-1);
    }
}
```

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

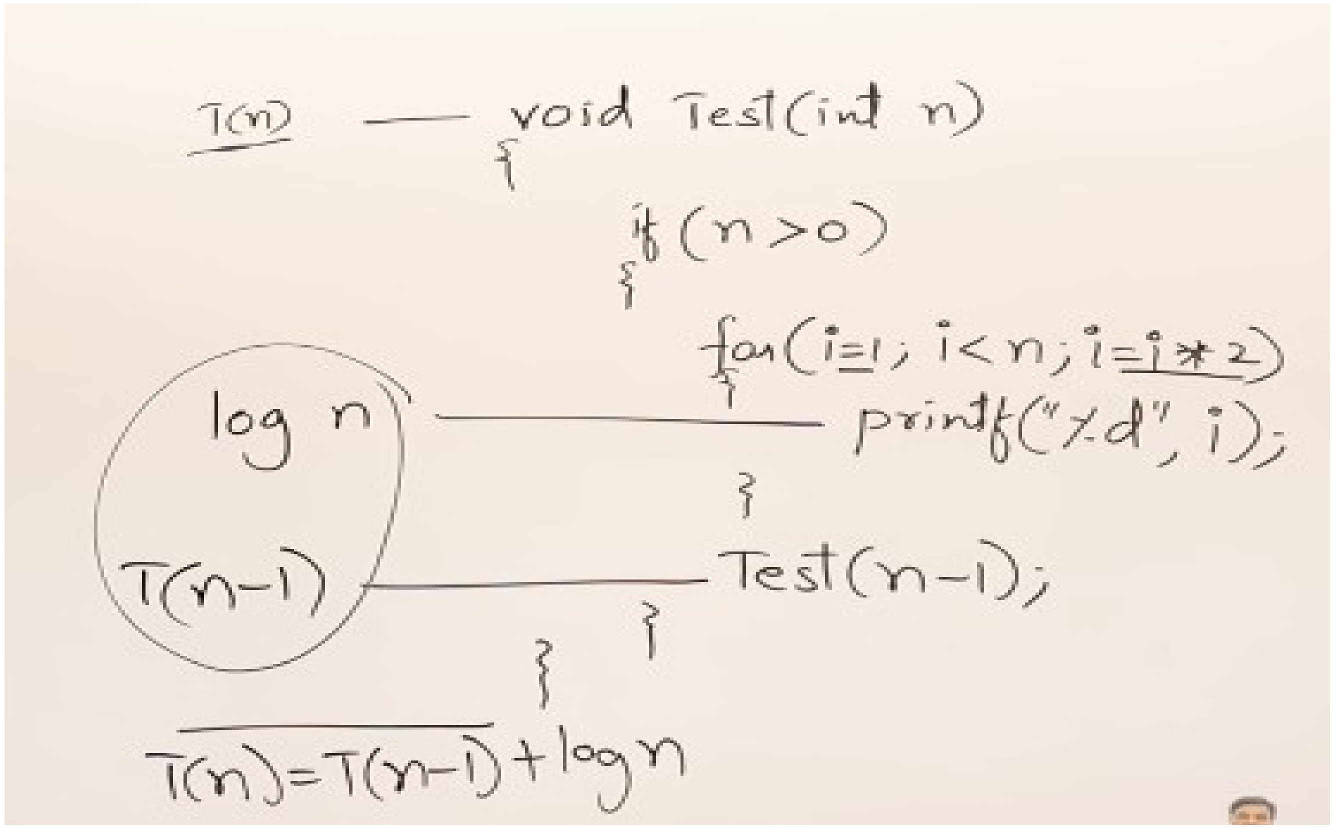
$$T(n)=\frac{n(n+1)}{2}$$

$$O(n^2)$$

Unit-1: Foundation of Algorithm Analysis

Recursion Tree Method

- Examples:



```
T(n) ——— void Test(int n)
         {
             if(n>0)
             {
                 for(i=1; i<n; i=i*2)
                 {
log n ———            printf("%d", i);
                 }
T(n-1) ———       Test(n-1);
             }
         }
```

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

Unit-1: Foundation of Algorithm Analysis

Recursion Tree Method

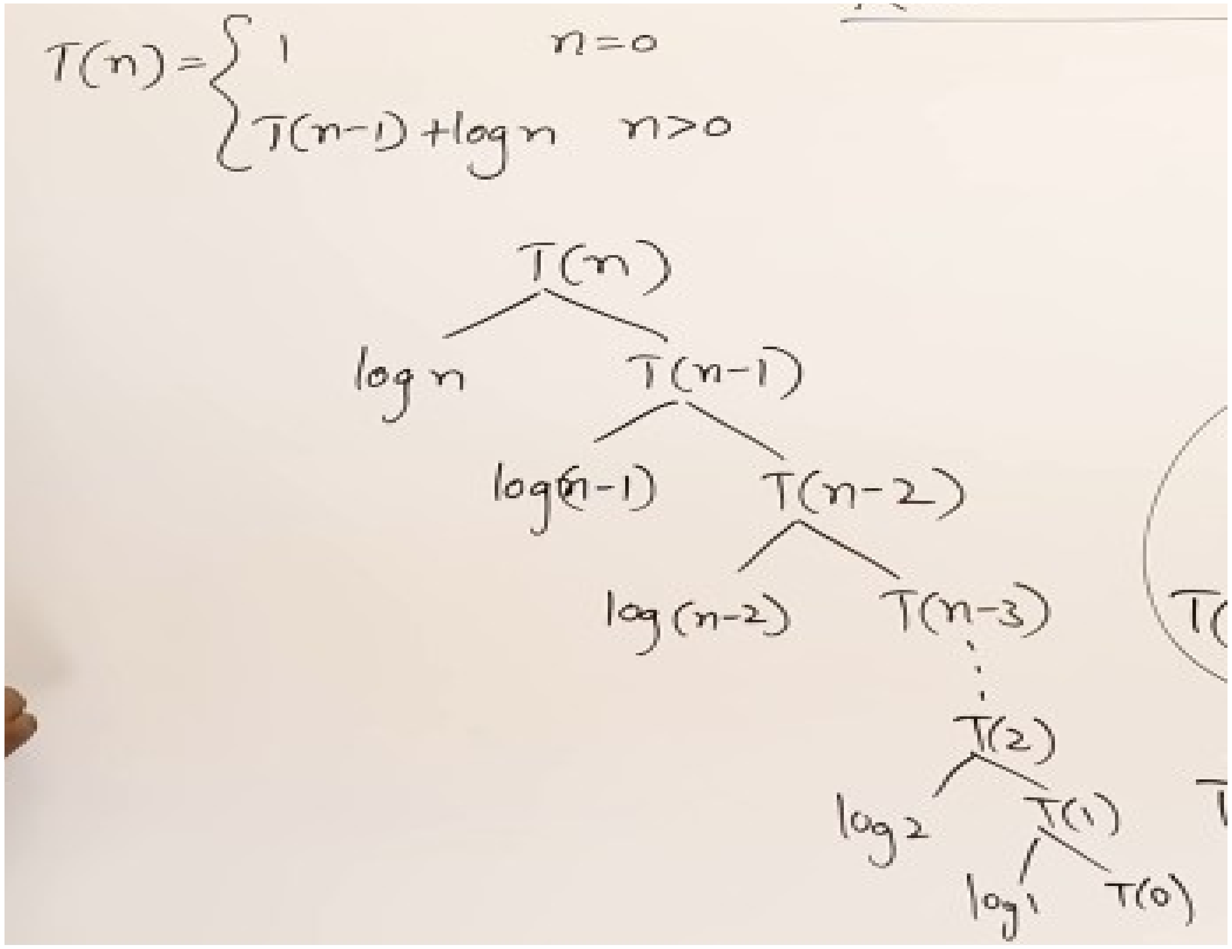
Examples:

T(n)= logn+log(n-1)+log(n-2)+...+log(2)+log(1)

= log[n*(n-1)*….*2*1)]

=log(n!)

=O(nlogn)

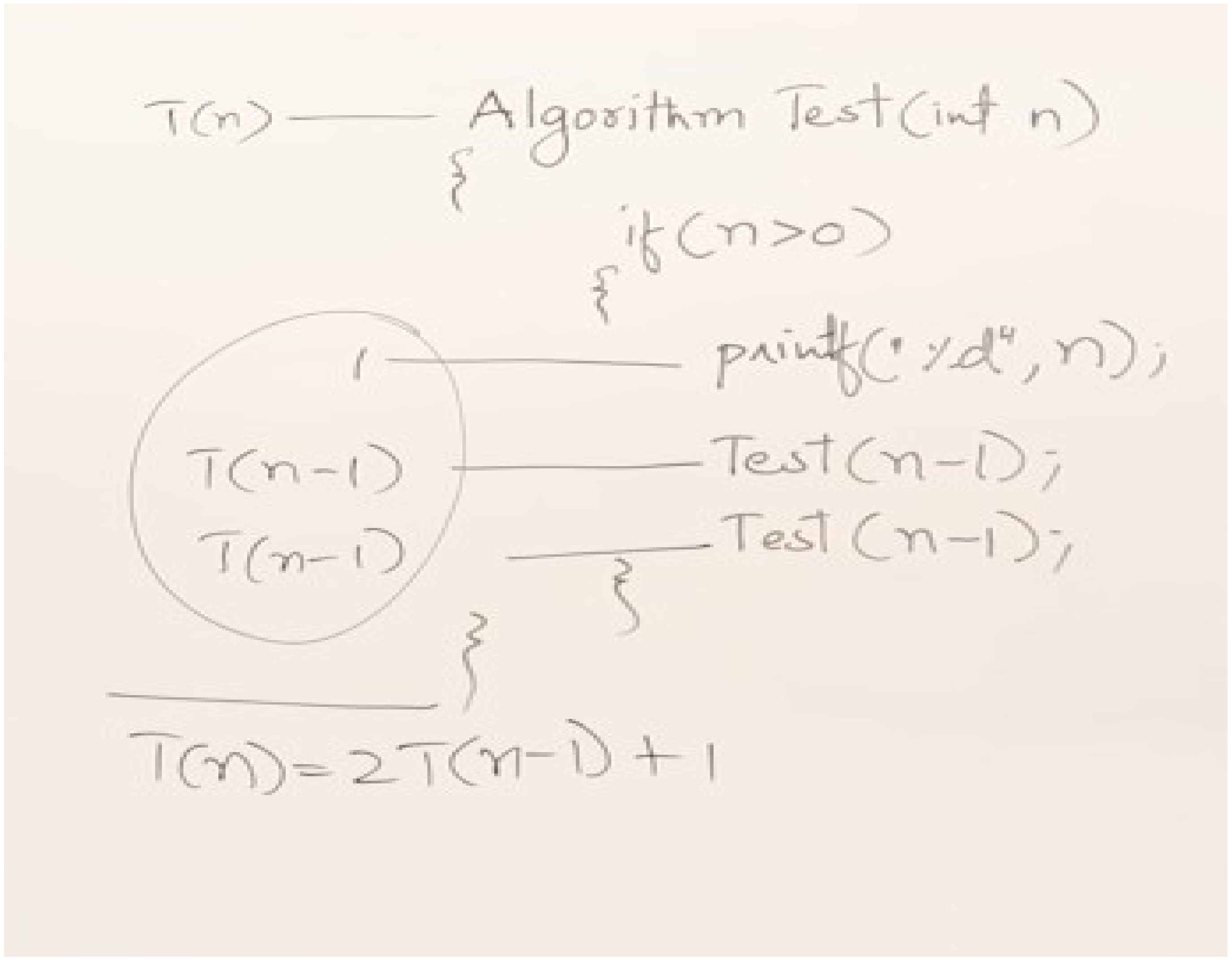


$T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+\log n & n>0 \end{cases}$
T(n)
log n
T(n-1)
log(n-1)
T(n-2)
log(n-2)
T(n-3)
T(2)
log 2
T(1)
log 1
T(0)

Unit-1: Foundation of Algorithm Analysis

Recursion Tree Method

- Examples:



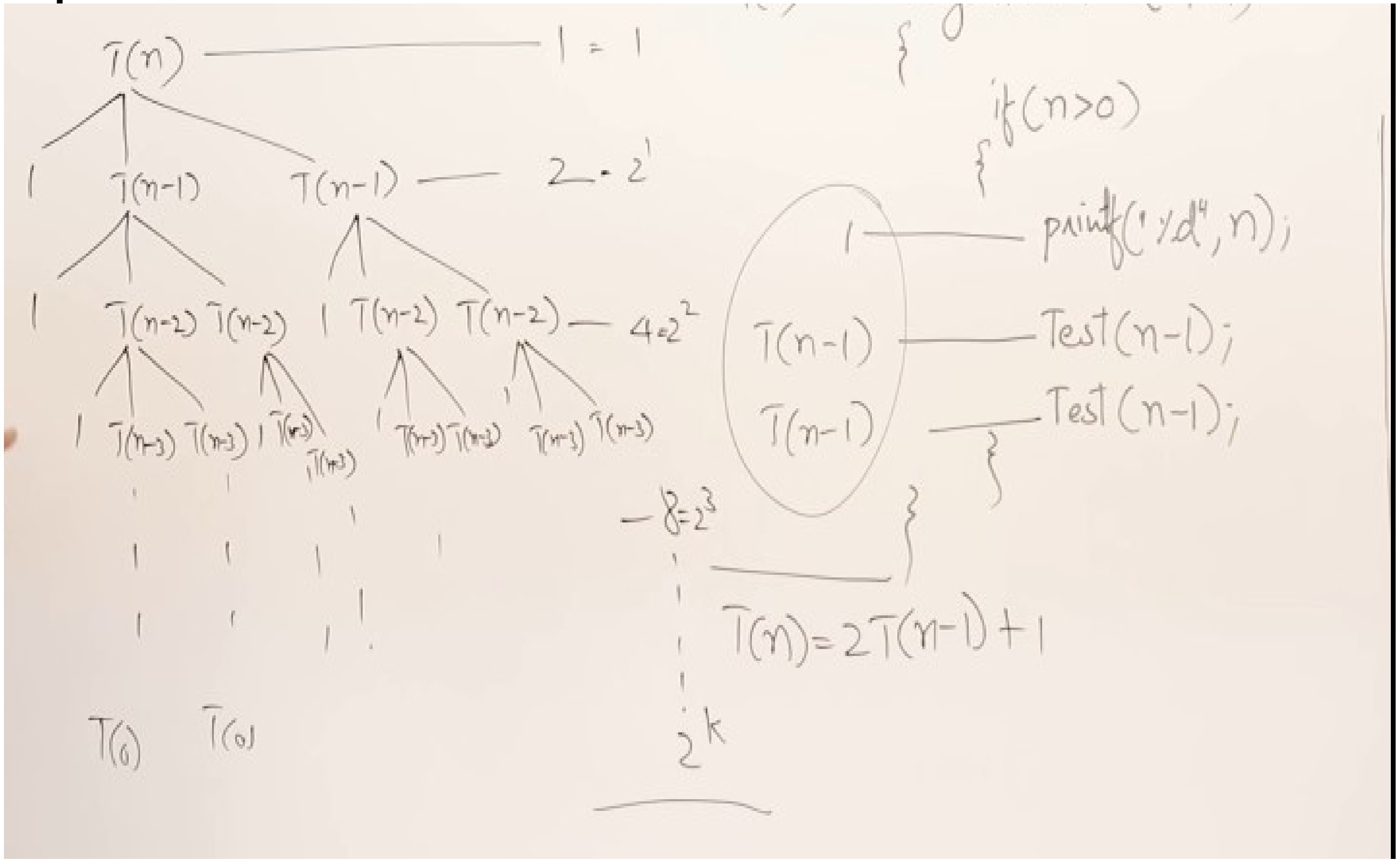
```
T(n) ———— Algorithm Test(int n)
          {
              if(n>0)
              {
   1   ——————     printf("%d", n);
   T(n-1) ————    Test(n-1);
   T(n-1) ————    Test(n-1);
              }
          }
```

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

Unit-1: Foundation of Algorithm Analysis

Recursion Tree Method

- Examples:



T(n) — 1 = 1
1 T(n-1) T(n-1) — 2 = 2^1
1 T(n-2) T(n-2) 1 T(n-2) T(n-2) — 4 = 2^2
T(n-3) T(n-3) T(n-3) T(n-3) T(n-3) T(n-3) T(n-3) T(n-3)
— 8 = 2^3
T(0) T(0)
2^k
{
if(n>0)
{
1 — printf("%d", n);
T(n-1) — Test(n-1);
T(n-1) — Test(n-1);
}
}
T(n) = 2T(n-1) + 1

Unit-1: Foundation of Algorithm Analysis

Recursion Tree Method

- Examples: $T(n)= \{1 \quad , n=0$

  $2T(n-1) +1, \quad n>1$

  - $T(n)= 1+2+2^2 +2^3 +\ldots\ldots..+2^k$
  - This pattern follow geometric series like,

    $a+ar+ar2+\ldots..+ar^k = a(r^{k+1}-1) /(r-1)$

  - Here a=1 and r=2, Then

    $T(n)= 1(2^{k+1}-1) /(2-1) = 2^{k+1}-1)$

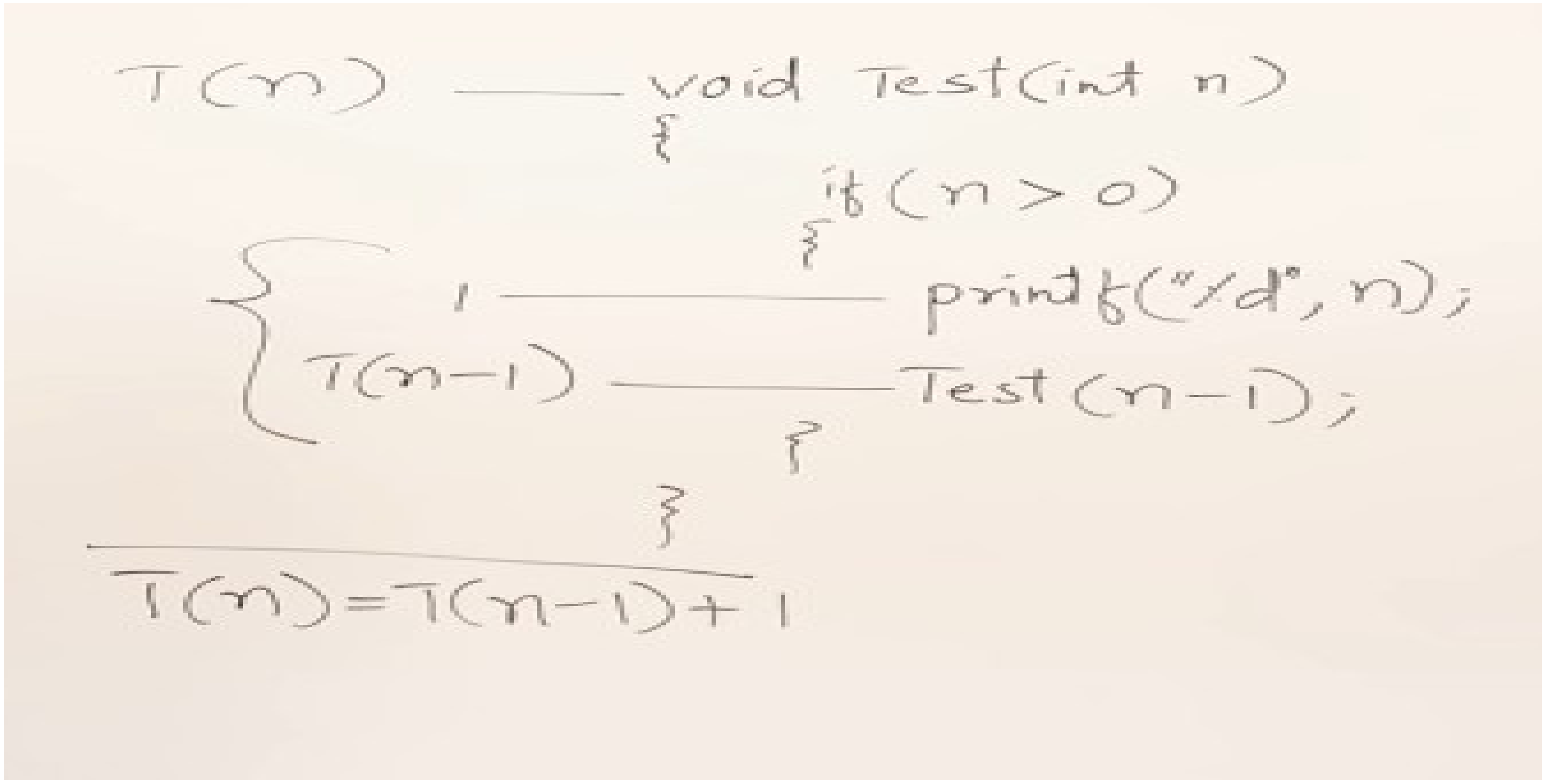
  - Assuming, (n-k)=0 then n=k gives,

    $T(n)= 2^{n-1}-1$ i.e. $O(2^n)$

Unit-1: Foundation of Algorithm Analysis

Substitution Method

- Example 1

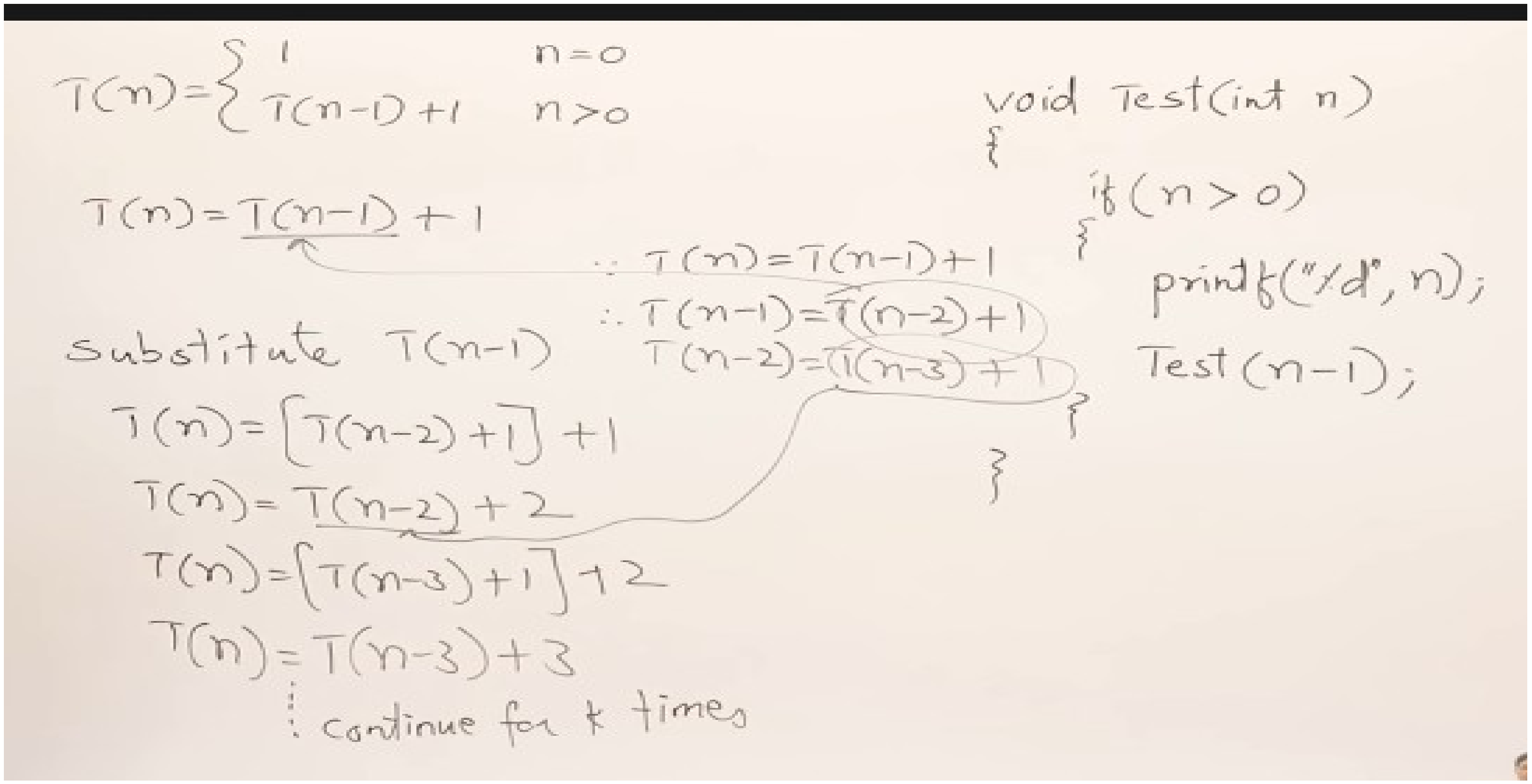


```
T(n) ——— void Test(int n)
         {
             if (n > 0)
             {
    1 ———————     printf("%d", n);
    T(n-1) ————   Test(n-1);
             }
         }
```

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

Unit-1: Foundation of Algorithm Analysis

Substitution Method

- Example 1
- 

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

```
void Test(int n)
{
    if(n>0)
    {
        printf("%d", n);
        Test(n-1);
    }
}
```

$T(n)=\underline{T(n-1)}+1$

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

$\therefore T(n-1)=T(n-2)+1$

$T(n-2)=T(n-3)+1$

substitute $T(n-1)$

$T(n)=[T(n-2)+1]+1$

$T(n)=\underline{T(n-2)}+2$

$T(n)=[T(n-3)+1]+2$

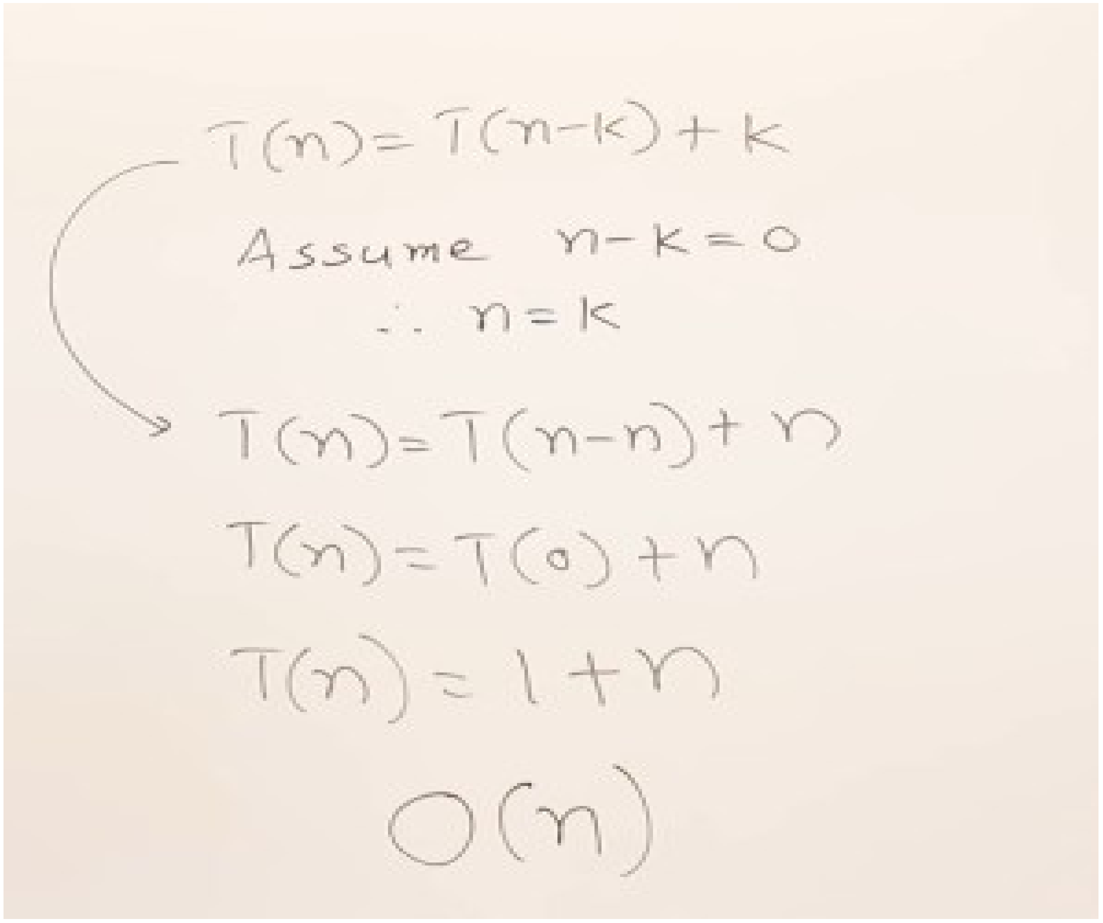
$T(n)=T(n-3)+3$

$\vdots$ continue for k times

Unit-1: Foundation of Algorithm Analysis

Substitution Method

- Example 1



$T(n) = T(n-k) + k$

Assume $n - k = 0$

$\therefore n = k$

$T(n) = T(n-n) + n$

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

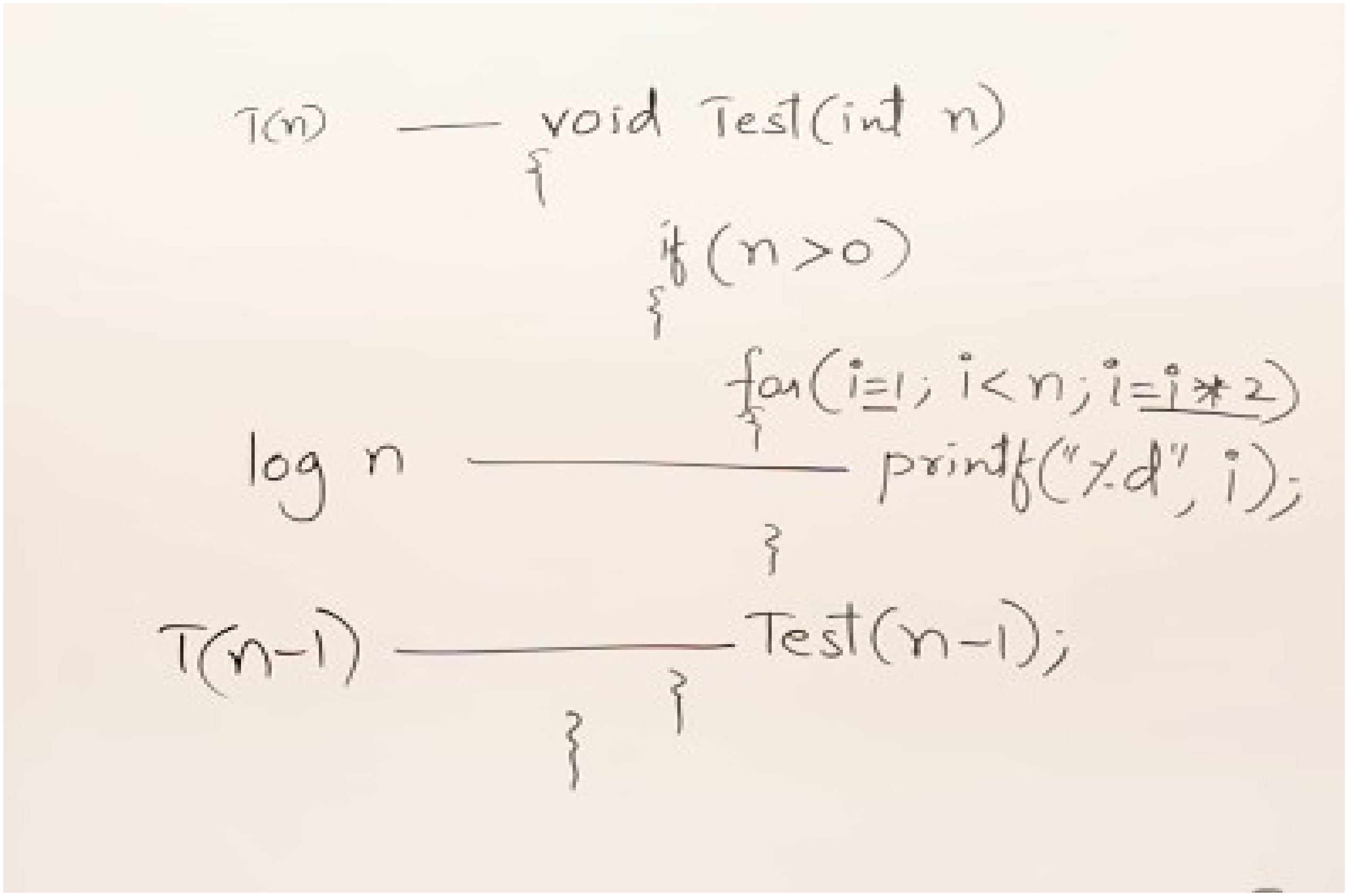
$T(n) = 1 + n$

$O(n)$

Unit-1: Foundation of Algorithm Analysis

Substitution Method

- Example 2

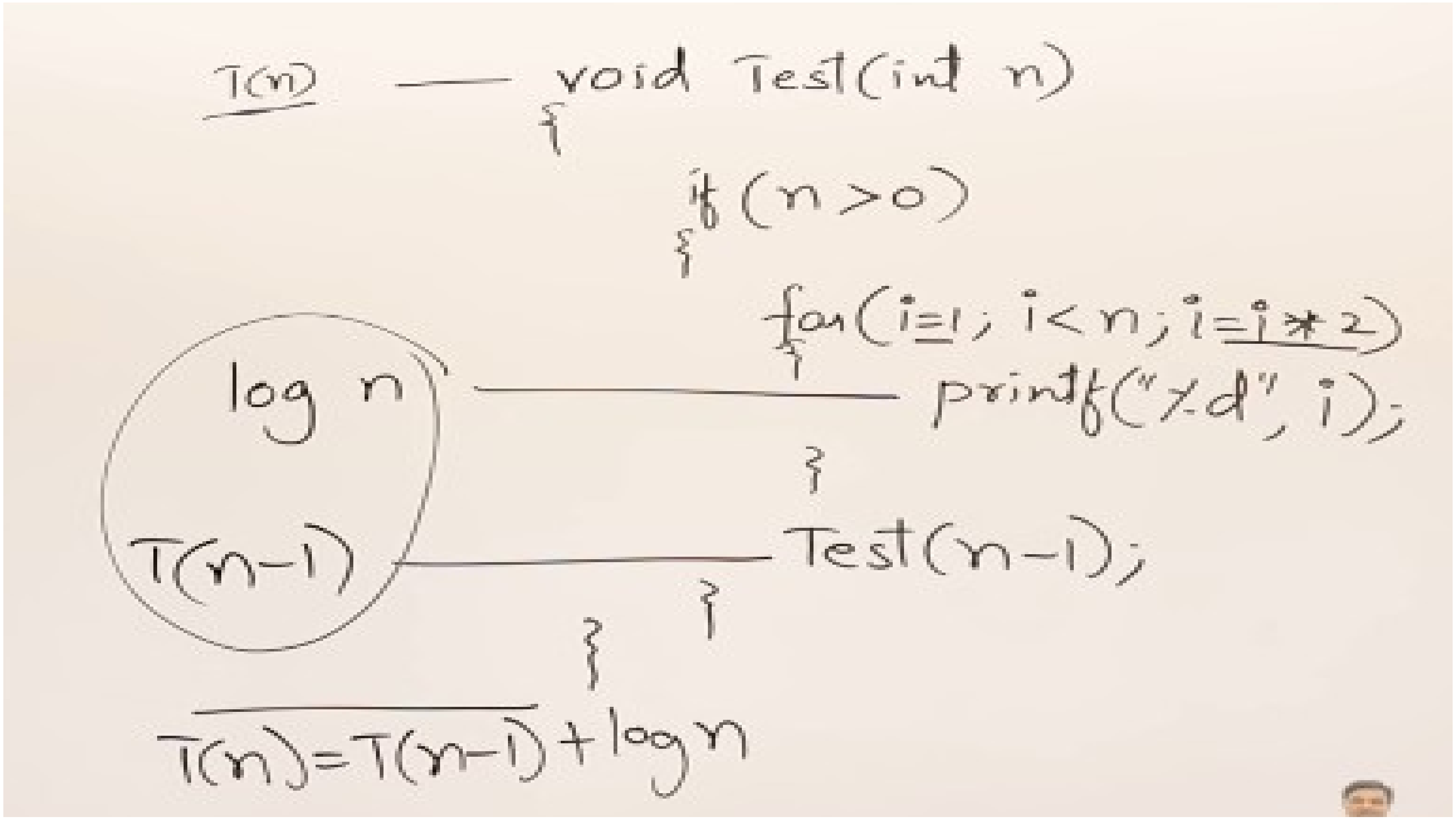


```
T(n)   —— void Test(int n)
          {
              if(n>0)
              {
                  for(i=1; i<n; i=i*2)
                  {
log n  ——             printf("%d", i);
                  }
T(n-1) ——         Test(n-1);
              }
          }
```

Unit-1: Foundation of Algorithm Analysis

Substitution Method

- Example 2



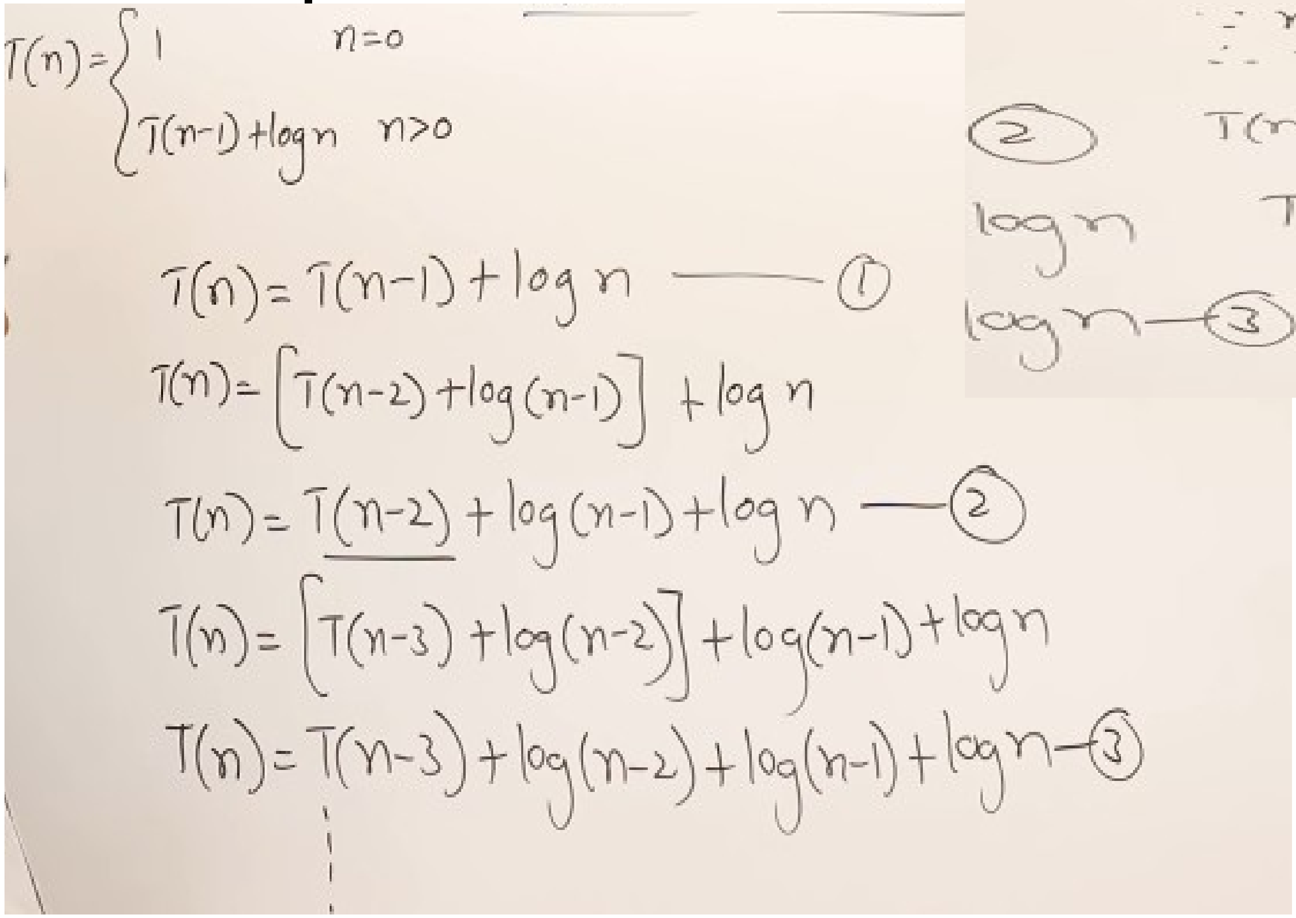
```
void Test(int n)
{
    if(n>0)
    {
        for(i=1; i<n; i=i*2)
        {
            printf("%d", i);
        }
        Test(n-1);
    }
}
```

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

Unit-1: Foundation of Algorithm Analysis

Substitution Method

- Example 2



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

$$T(n)=T(n-1)+\log n \text{ —— } ①$$

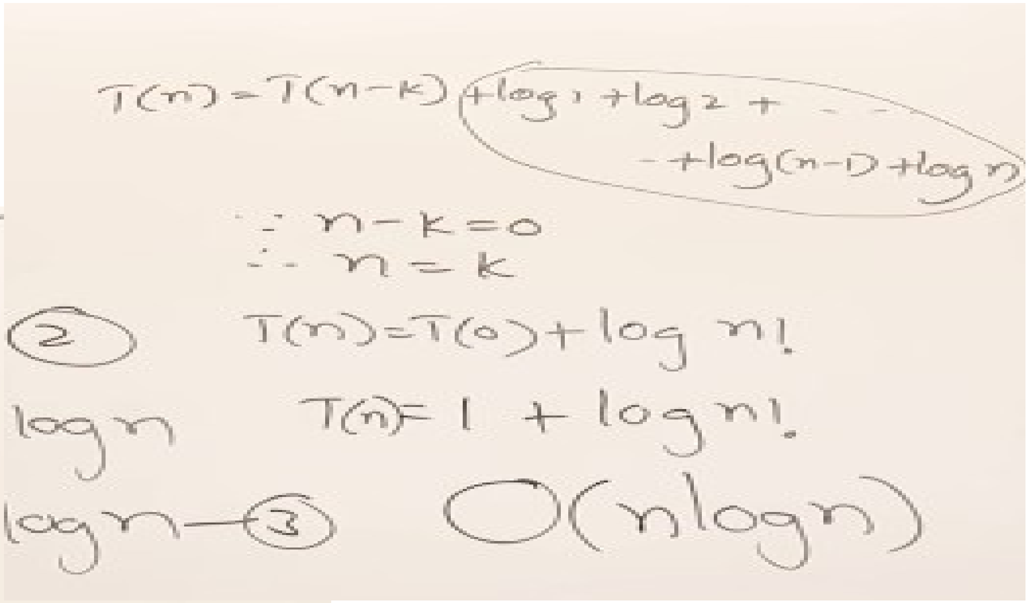
$$T(n)=[T(n-2)+\log(n-1)]+\log n$$

$$T(n)=\underline{T(n-2)}+\log(n-1)+\log n \text{ —— } ②$$

$$T(n)=[T(n-3)+\log(n-2)]+\log(n-1)+\log n$$

$$T(n)=T(n-3)+\log(n-2)+\log(n-1)+\log n \text{ — } ③$$

$$\vdots$$



$$T(n)=T(n-k)+\log 1+\log 2+\dots+\log(n-1)+\log n$$

$$\because n-k=0$$

$$\therefore n=k$$

② $T(n)=T(0)+\log n!$

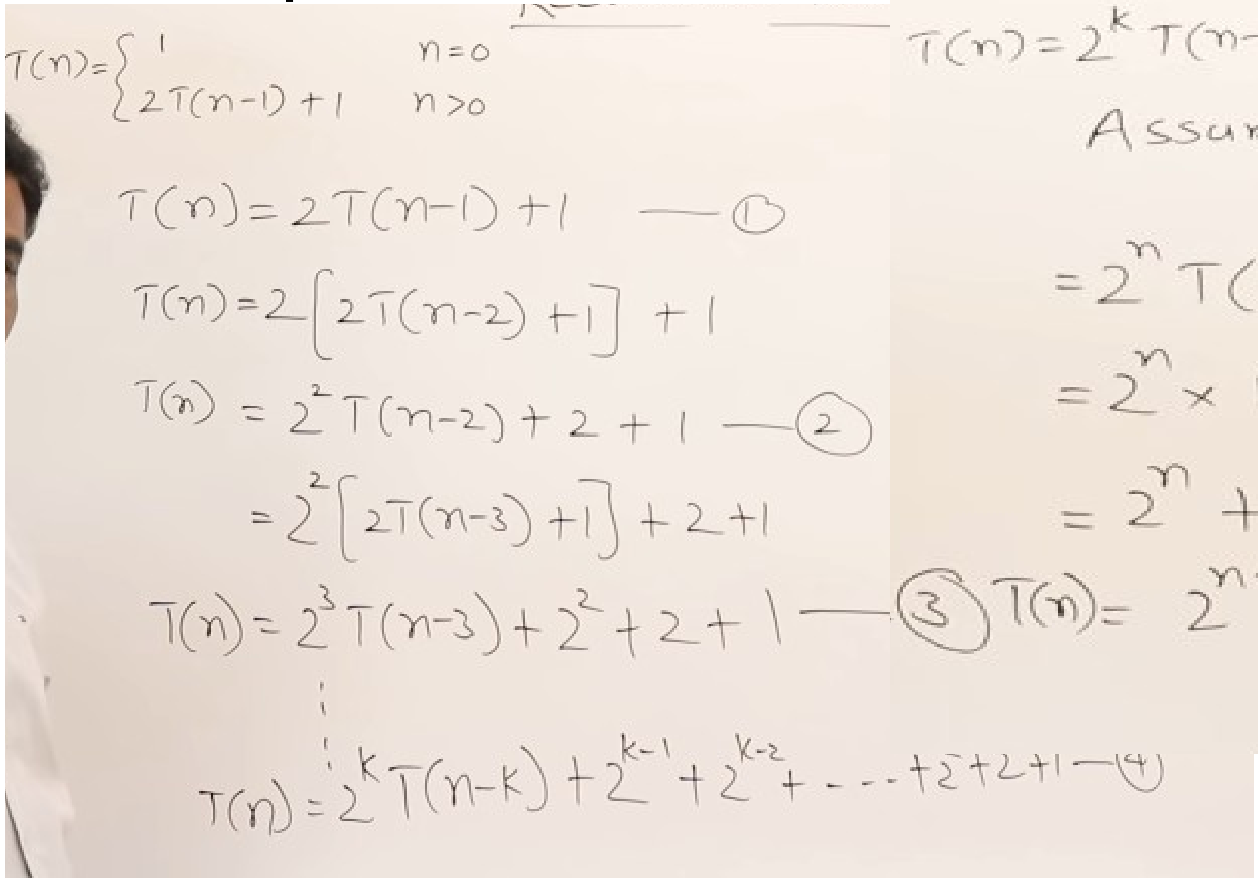
$\log n$ $\quad T(n)=1+\log n!$

$\log n$ — ③ $\quad O(n\log n)$

Unit-1: Foundation of Algorithm Analysis

Substitution Method

- Example 3



$$T(n)=\begin{cases}1 & n=0\\ 2T(n-1)+1 & n>0\end{cases}$$

$$T(n)=2T(n-1)+1 \quad \text{— } (1)$$

$$T(n)=2\left[2T(n-2)+1\right]+1$$

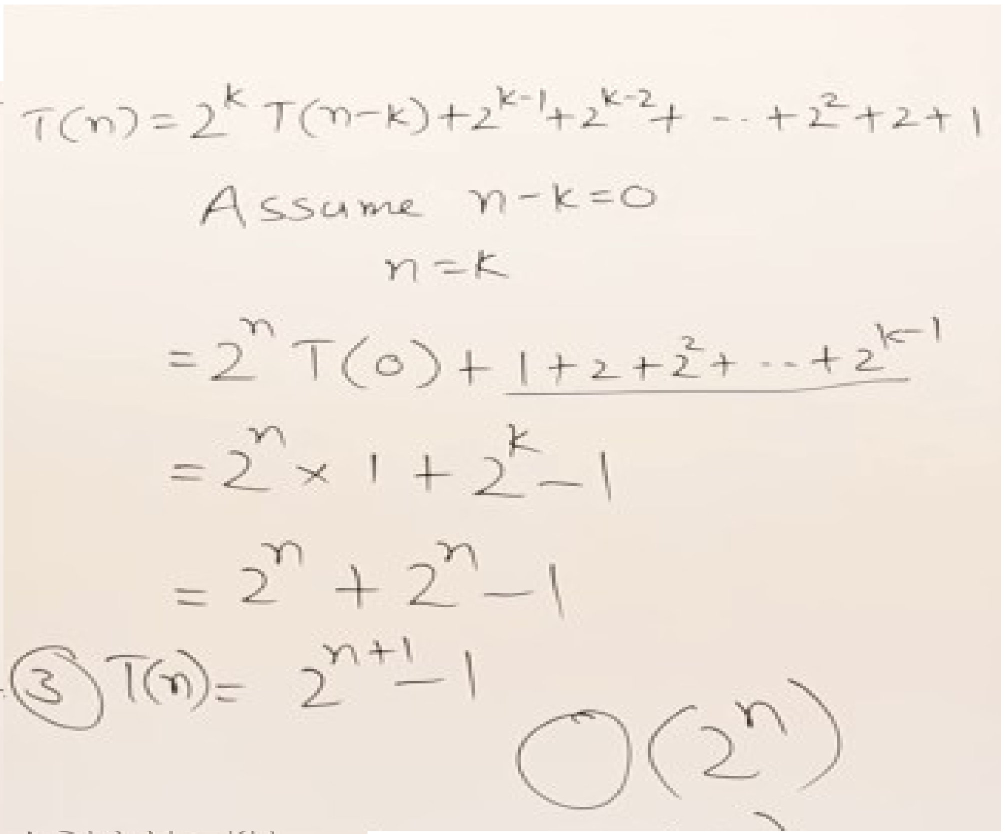
$$T(n)=2^2T(n-2)+2+1 \quad \text{— } (2)$$

$$=2^2\left[2T(n-3)+1\right]+2+1$$

$$T(n)=2^3T(n-3)+2^2+2+1 \quad \text{— } (3)$$

$$\vdots$$

$$T(n)=2^kT(n-k)+2^{k-1}+2^{k-2}+\dots+2^2+2+1 \quad \text{— } (4)$$



$$T(n)=2^kT(n-k)+2^{k-1}+2^{k-2}+\dots+2^2+2+1$$

Assume $n-k=0$

$$n=k$$

$$=2^nT(0)+\underline{1+2+2^2+\dots+2^{k-1}}$$

$$=2^n\times 1+2^k-1$$

$$=2^n+2^n-1$$

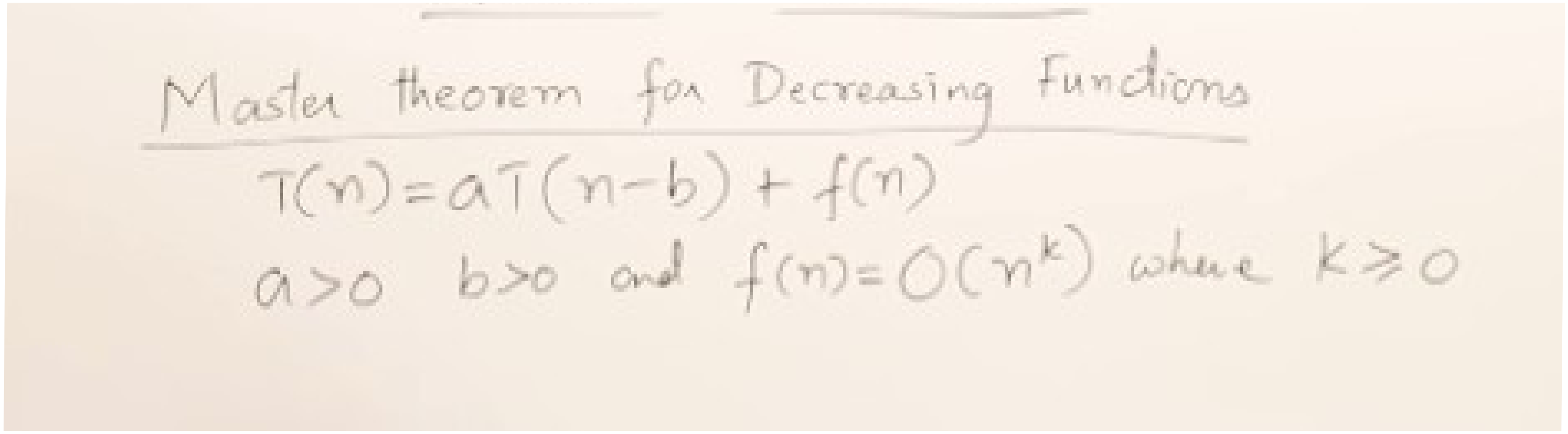
$$T(n)=2^{n+1}-1$$

$$O(2^n)$$

Unit-1: Foundation of Algorithm Analysis

Master Method

- Master method is a direct way of getting the solution.
-

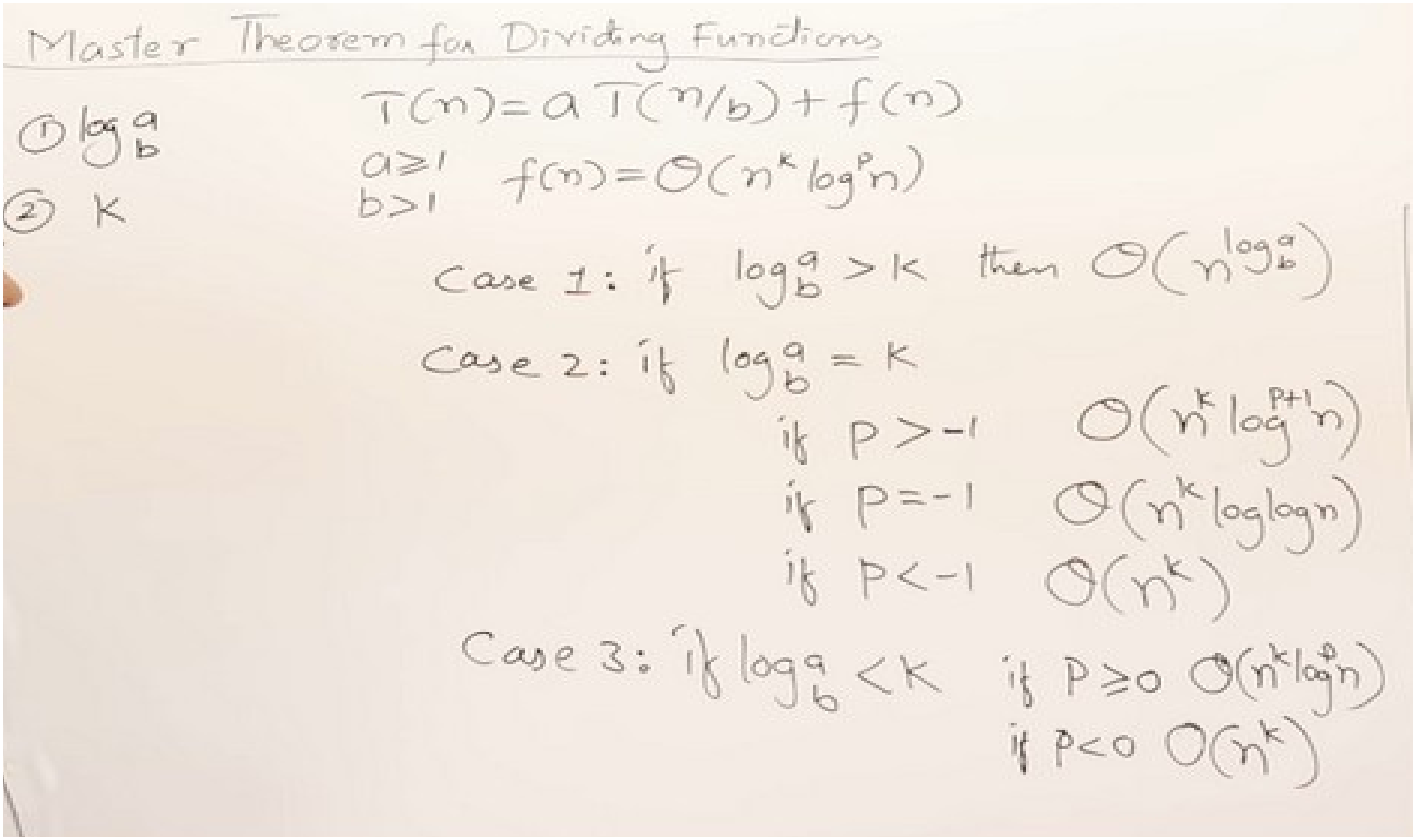


Master theorem for Decreasing Functions

$T(n) = aT(n-b) + f(n)$

$a>0 \quad b>0$ and $f(n) = O(n^k)$ where $k \geq 0$

Master Method : General form



Master Theorem for Dividing Functions

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

① $\log_b a$

② k

$a \geq 1$

$b > 1$

$f(n) = \Theta(n^k \log^p n)$

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

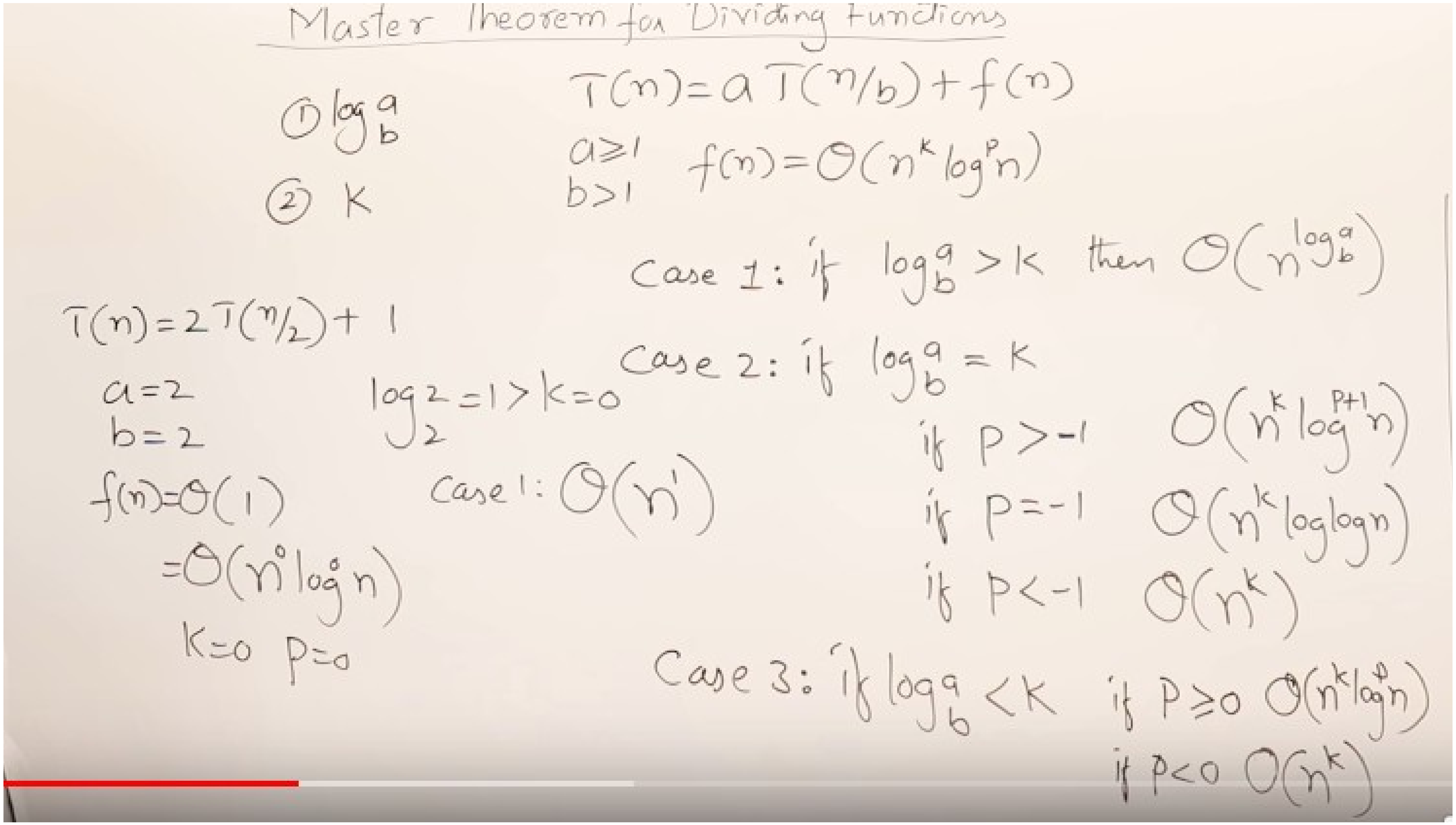
- if $p > -1$ $\Theta(n^k \log^{p+1} n)$
- if $p = -1$ $\Theta(n^k \log\log n)$
- if $p < -1$ $\Theta(n^k)$

Case 3: if $\log_b a < k$

- if $p \geq 0$ $\Theta(n^k \log^p n)$
- if $p < 0$ $O(n^k)$

Unit-1: Foundation of Algorithm Analysis

Master Method: Example



Master Theorem for Dividing Functions

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

$a \geq 1$

$b > 1$

$f(n) = \Theta(n^k \log^p n)$

① $\log_b a$

② k

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

- if $p > -1$ $\Theta(n^k \log^{p+1} n)$
- if $p = -1$ $\Theta(n^k \log\log n)$
- if $p < -1$ $\Theta(n^k)$

Case 3: if $\log_b a < k$

- if $p \geq 0$ $\Theta(n^k \log^p n)$
- if $p < 0$ $O(n^k)$

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

$a = 2$

$b = 2$

$f(n) = \Theta(1)$

$= \Theta(n^0 \log^0 n)$

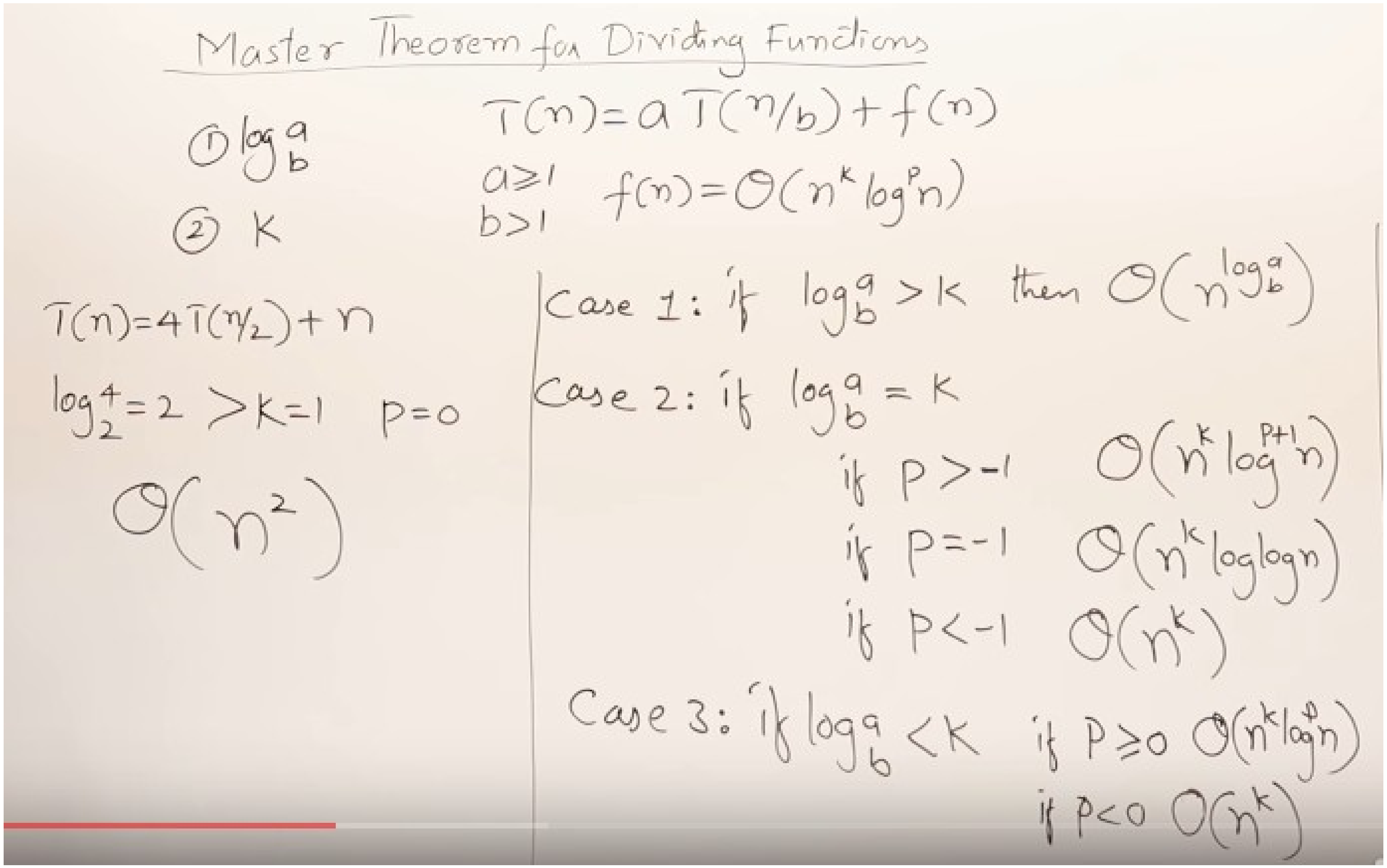
$k = 0$ $p = 0$

$\log_2 2 = 1 > k = 0$

case 1: $\Theta(n^1)$

Unit-1: Foundation of Algorithm Analysis

Master Method: Example



Master Theorem for Dividing Functions

① $\log_b a$

② k

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

$a \geq 1$

$b > 1$

$f(n) = \Theta(n^k \log^p n)$

$T(n) = 4T(n/2) + n$

$\log_2 4 = 2 > k = 1 \quad p = 0$

$\Theta(n^2)$

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

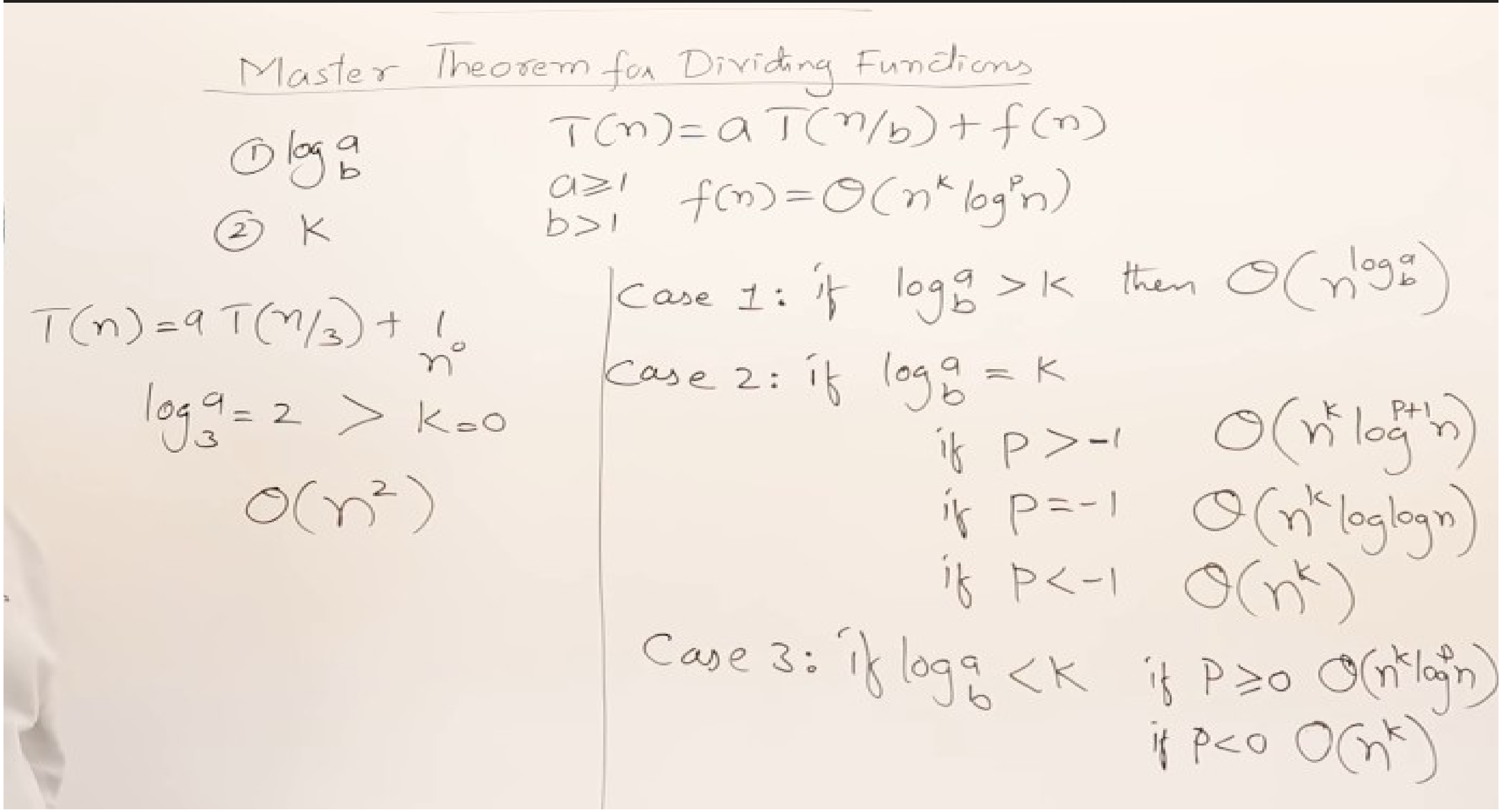
- if $p > -1$ $\quad \Theta(n^k \log^{p+1} n)$
- if $p = -1$ $\quad \Theta(n^k \log\log n)$
- if $p < -1$ $\quad \Theta(n^k)$

Case 3: if $\log_b a < k$

- if $p \geq 0$ $\quad \Theta(n^k \log^p n)$
- if $p < 0$ $\quad \Theta(n^k)$

Unit-1: Foundation of Algorithm Analysis

Master Method: Example



Master Theorem for Dividing Functions

① $\log_b a$

② k

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

$a \geq 1$

$b > 1$

$f(n) = \Theta(n^k \log^p n)$

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

if $p > -1$ $\Theta(n^k \log^{p+1} n)$

if $p = -1$ $\Theta(n^k \log\log n)$

if $p < -1$ $\Theta(n^k)$

Case 3: if $\log_b a < k$ if $p \geq 0$ $\Theta(n^k \log^p n)$

if $p < 0$ $O(n^k)$

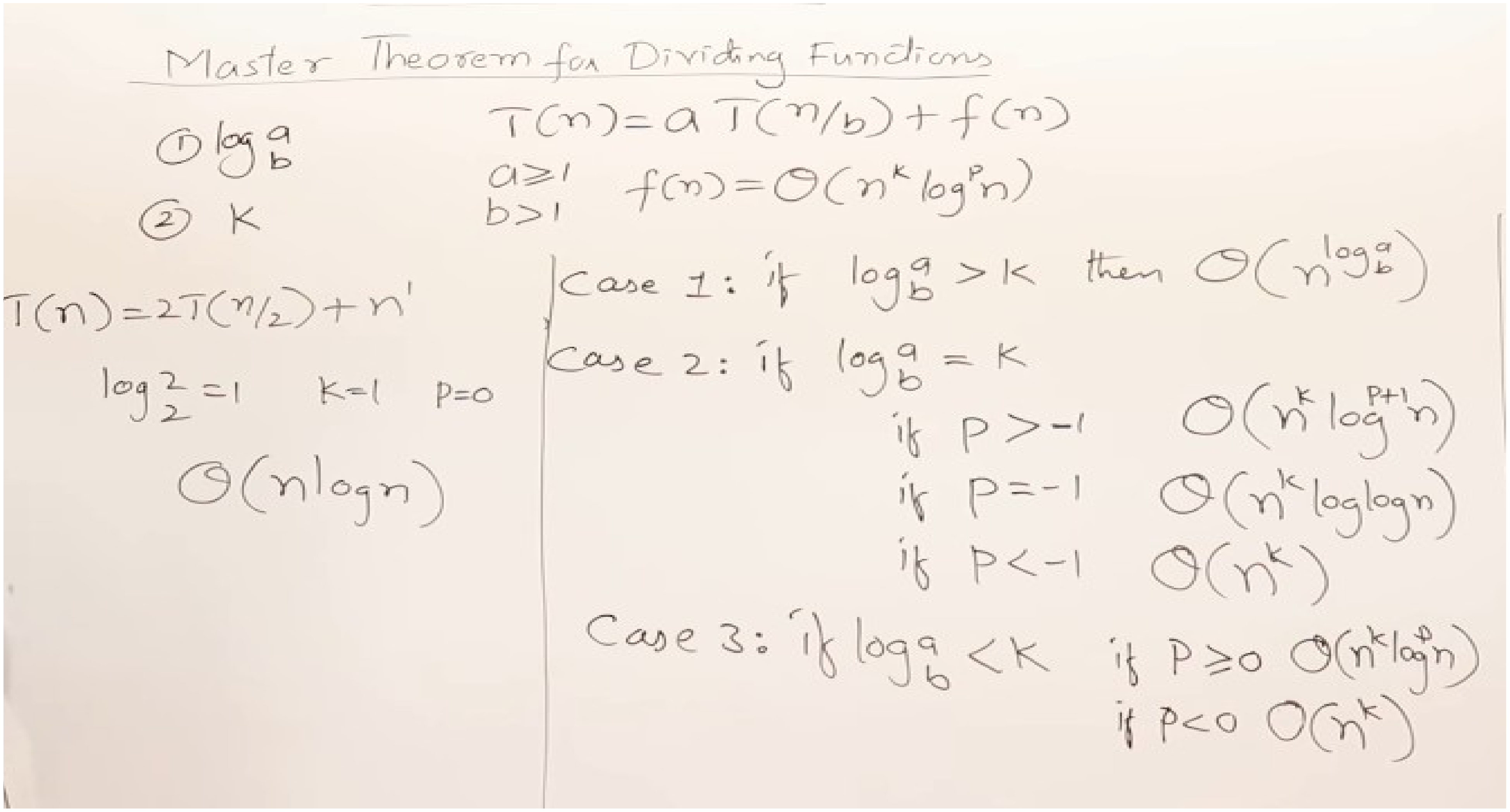
$T(n) = 9T(n/3) + \underset{n^0}{1}$

$\log_3 9 = 2 > k = 0$

$\Theta(n^2)$

Unit-1: Foundation of Algorithm Analysis

Master Method: Example



Master Theorem for Dividing Functions

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

$a \geq 1$ $\quad f(n) = \Theta(n^k \log^p n)$

$b > 1$

① $\log_b a$

② k

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

if $p > -1$ $\quad \Theta(n^k \log^{p+1} n)$

if $p = -1$ $\quad \Theta(n^k \log\log n)$

if $p < -1$ $\quad \Theta(n^k)$

Case 3: if $\log_b a < k$ $\quad$ if $p \geq 0$ $\quad \Theta(n^k \log^p n)$

if $p < 0$ $\quad \Theta(n^k)$

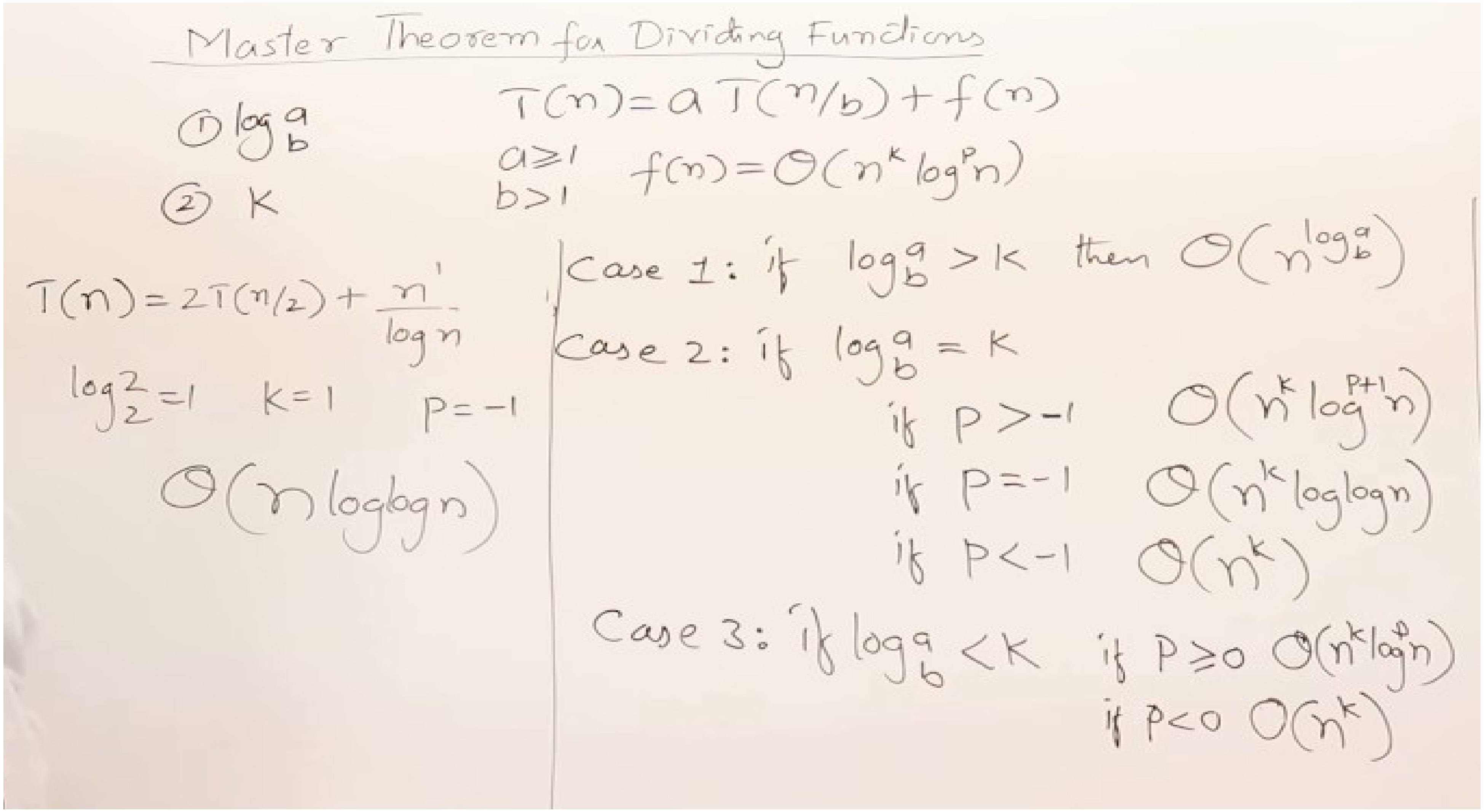
$T(n) = 2T(n/2) + n^1$

$\log_2 2 = 1 \quad k = 1 \quad p = 0$

$\Theta(n \log n)$

Unit-1: Foundation of Algorithm Analysis

Master Method: Example



Master Theorem for Dividing Functions

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

① $\log_b a$

② k

$a \geq 1$
$b > 1$ $f(n) = \Theta(n^k \log^p n)$

$T(n) = 2T(n/2) + \frac{n^1}{\log n}$

$\log_2 2 = 1$ $k = 1$ $p = -1$

$\Theta(n \log\log n)$

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

if $p > -1$ $\Theta(n^k \log^{p+1} n)$

if $p = -1$ $\Theta(n^k \log\log n)$

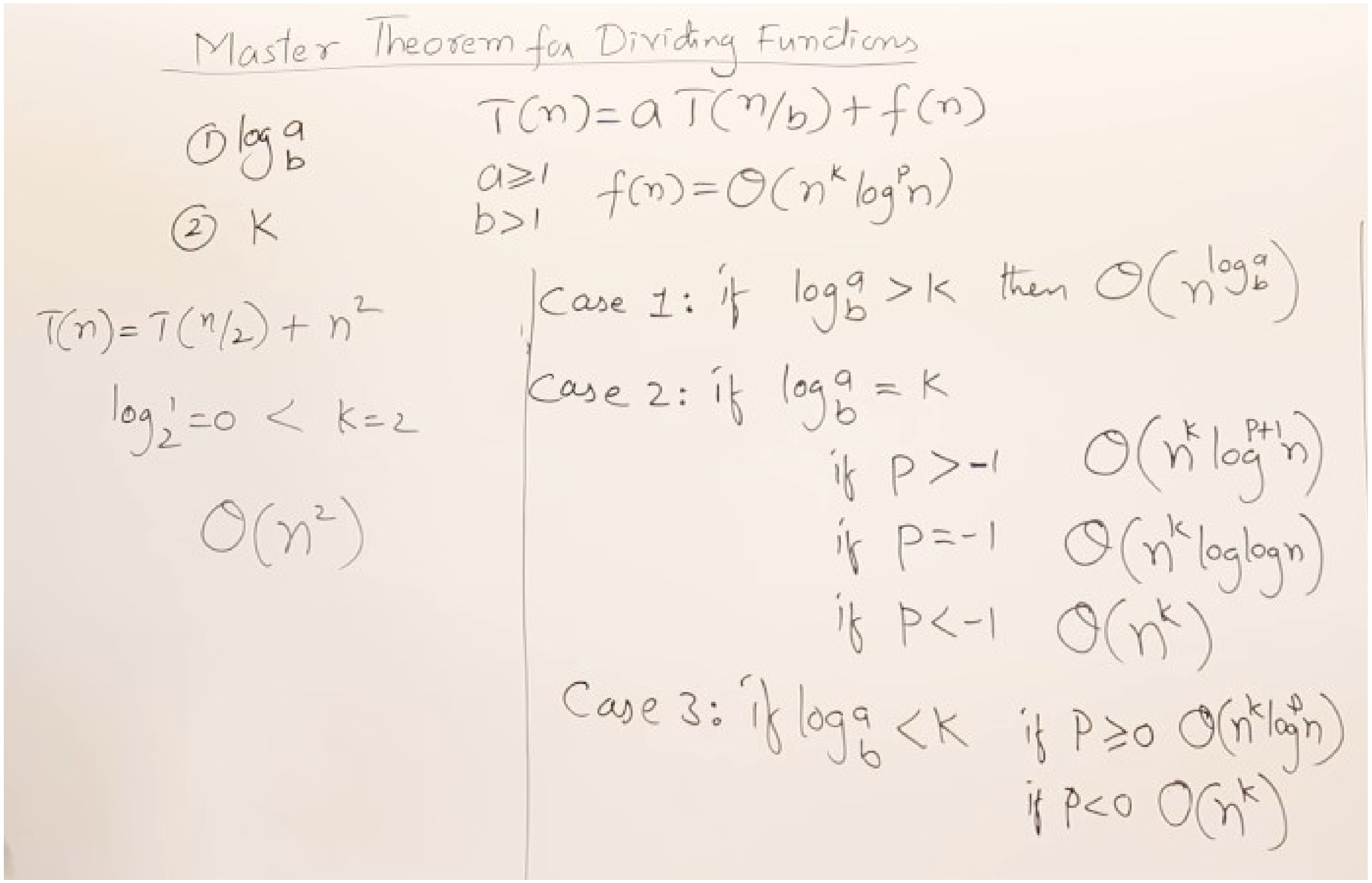
if $p < -1$ $\Theta(n^k)$

Case 3: if $\log_b a < k$ if $p \geq 0$ $\Theta(n^k \log^p n)$

if $p < 0$ $O(n^k)$

Unit-1: Foundation of Algorithm Analysis

Master Method: Example



Master Theorem for Dividing Functions

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

① $\log_b a$

② k

$a \geq 1$

$b > 1$

$f(n) = \Theta(n^k \log^p n)$

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

if $p > -1$ $\quad \Theta(n^k \log^{p+1} n)$

if $p = -1$ $\quad \Theta(n^k \log\log n)$

if $p < -1$ $\quad \Theta(n^k)$

Case 3: if $\log_b a < k$ $\quad$ if $p \geq 0$ $\Theta(n^k \log^p n)$

if $p < 0$ $\quad \Theta(n^k)$

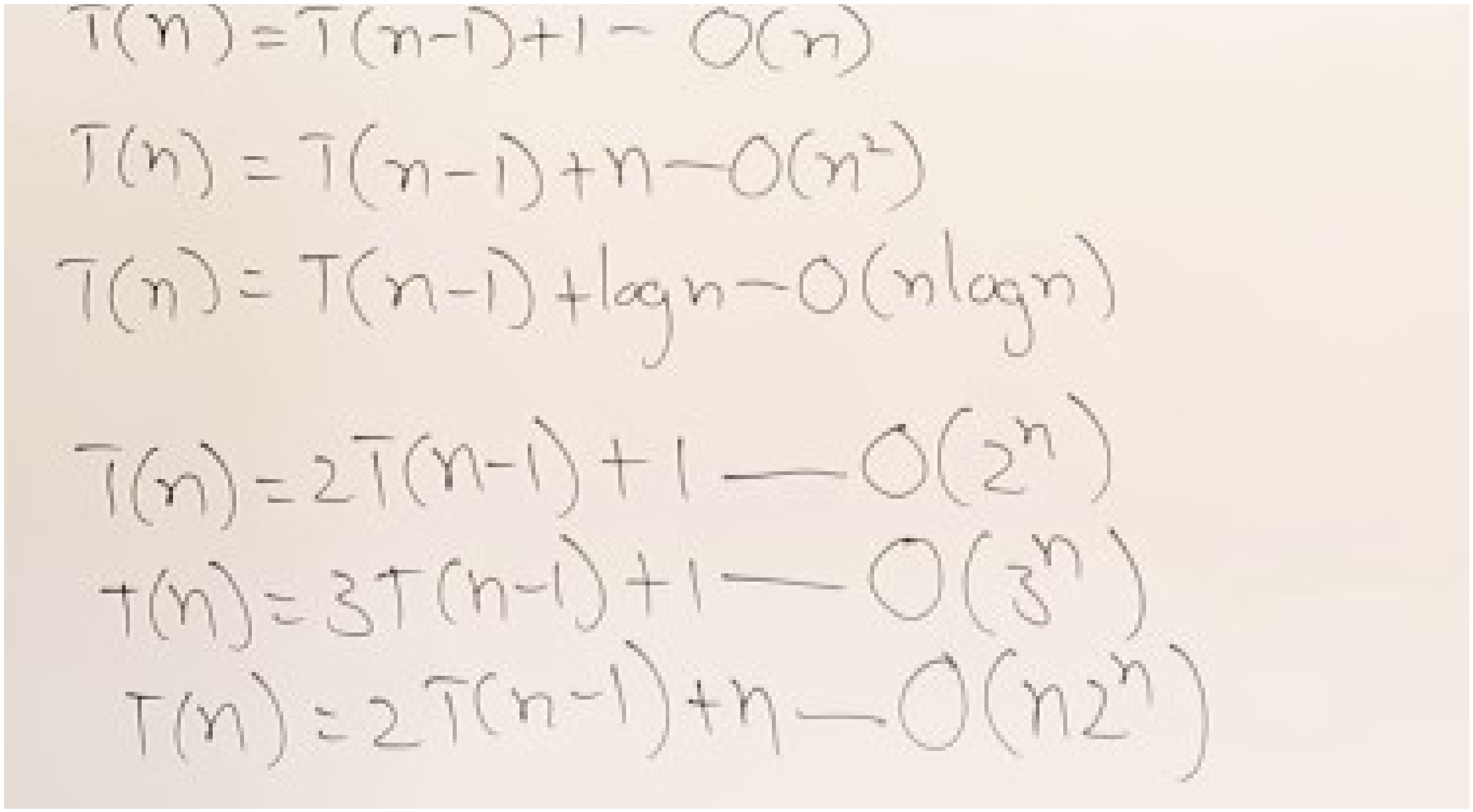
$T(n) = T(n/2) + n^2$

$\log_2 1 = 0 < k = 2$

$\Theta(n^2)$

Unit-1: Foundation of Algorithm Analysis

Summary of some Recurrence functions



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

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

$T(n) = T(n-1) + \log n - O(n \log n)$

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

$T(n) = 3T(n-1) + 1 - O(3^n)$

$T(n) = 2T(n-1) + n - O(n2^n)$

Unit-1: Foundation of Algorithm Analysis

Class Assignments

- How algorithm is a technology ? Explain.
- What do you understand by Analysis of Algorithm? Briefly explain the use of RAM Model in algorithm analysis.
- Briefly explain the Aggregate Analysis with example.
- What is Asymptotic Analysis? Does Asymptotic Analysis always work?
- Discuss some general properties of Asymptotic Notations.

Unit-1: Foundation of Algorithm Analysis

Class Assignments

- Solve the following Recurrence relations using master methods.
  - $T(n)= 2T(n/4)+\sqrt{n}$
  - $T(n)= 2T(2n/3)+1$
  - $T(n)= 9T(n/3)+n$
  - $T(n)= 4T(n/2)+n^2$
  - $T(n)= 3T(n/2)+n$
  - $T(n)= 3T(n/4)+n \log n$
  - $T(n)= 4T(n/4)+n^2/ \log n$
-

Unit-1: Foundation of Algorithm Analysis

Class Assignments

- Solve the following Recurrence relations using Recursion tree methods.
  - $T(1)=1$ when $n=1$

    $T(n)= T(n-1)+1$ when $n>1$
  - $T(1)=1$ when $n=1$

    $T(n)= 2T(n/2)+1$ when $n>1$
  - $T(1)=1$ when $n=1$

    $T(n)= 2T(n/2)+n$ when $n>1$

Unit-1: Foundation of Algorithm Analysis

Class Assignments

- Solve the following Recurrence relations using substitution methods.
  - $T(1)=1$ when $n=1$

    $T(n)= T(n-1)+1$ when $n>1$
  - $T(1)=1$ when $n=1$

    $T(n)= 2T(n/2)+1$ when $n>1$
  - $T(1)=1$ when $n=1$

    $T(n)= 8T(n/2)+n^2$ when $n>1$
  - $T(1)=1$ when $n=1$

    $T(n)= 4T(n/2)+n$ when $n>1$

Unit-1: Foundation of Algorithm Analysis

Thank You