Introduction

What is an Algorithm?

- Algorithm is a set of steps to complete a task.
- Its is a finite set of instructions that is to be followed to accomplish the task.
- Criteria's:
  - Input
  - Output
  - Definiteness
  - Finiteness
  - Effectiveness

Analysis of algorithm

- Task of determining how much computing time and storage an algorithm requires.

- Two approaches to determine performance of an algo
  - Time complexity: amount of time require to execute program
  - Space complexity: amount of memory required to run the execution

Space complexity

- ***Space Complexity*** of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input
- vector<**int**> **myVec**(n);

  for(**int** i = 0; i < n; i++)

  printf(myVec[i]);
- Space needed by program has following components:
  - Instruction space
  - Data space
  - Environment stack space

Time complexity

- The ***time complexity*** is the number of operations an algorithm performs to complete its task with respect to **input size** (considering that each operation takes the same amount of time).
- ***Input Size:*** *Input size is defined as total number of elements present in the input. For a given problem we characterize the input size* ***n*** *approproately. For example:*
  - *Sorting problem: Total number of item to be sorted*
  - *Graph Problem: Total number of vertices and edges*
  - *Numerical Problem: Total number of bits needed to represent a number*

Asymptotic notation

- we can't judge an algorithm by calculating the time taken during its execution in a particular system.
- We need some standard notation to analyze the algorithm.
- ***Asymptotic notation*** to analyze any algorithm and based on that we find the most efficient algorithm.
  - Here in Asymptotic notation, we do not consider the system configuration, rather we consider the **order of growth** of the input.
  - We try to find how the time or the space taken by the algorithm will increase/decrease after increasing/decreasing the input size.
- There are three asymptotic notations that are used to represent the time complexity of an algorithm:
  - **Big O Notation**
  - **Big Ω Notation**
  - **Big Θ Notation (theta)**

Best case, Average case, and Worst case

- An algorithm can have different time for different inputs. It may take 1 second for some input and 10 seconds for some other input.
- **Best case:** This is the lower bound on running time of an algorithm. We must know the case that causes the minimum number of operations to be executed. If array [1, 2, 3, 4, 5] and we are finding if "1" is present in the array or not. So here, after only one comparison, you will get that your element is present in the array. So, this is the best case of your algorithm.
- **Average case:** We calculate the running time for all possible inputs, sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) distribution of cases.
- **Worst case:** This is the upper bound on running time of an algorithm. We must know the case that causes the maximum number of operations to be executed. In our example, the worst case can be if the given array is [1, 2, 3, 4, 5] and we try to find if element "6" is present in the array or not. Here, the if-condition of our loop will be executed 5 times and then the algorithm will give "0" as output.

Asymptotic notation (Big O, Big **Ω, Big Θ)**

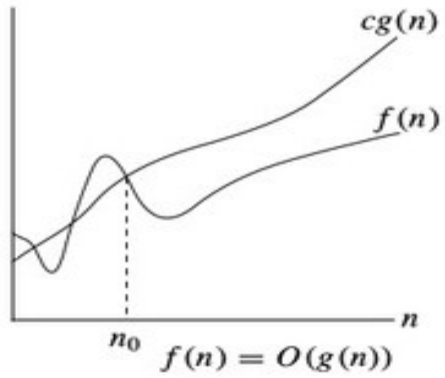
- In the asymptotic analysis, we generally deal with large input size.
- **Big O notation**
- defines the upper bound of any algorithm
- In other words, we can say that the big O notation denotes the maximum time taken by an algorithm or the worst-case time complexity of an algorithm.
- So, big O notation is the most used notation for the time complexity of an algorithm.
- So, if a function is g(n), then the big O representation of g(n) is shown as O(g(n)) and the relation is shown as:

```
O(g(n)) = { f(n): there exist positive constants c and n0
        such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n0 }
```

if f(n) = 2n² + 3n + 1

and **g**(n) = n²

then **for** c = 6 and n0 = 1, we can say that **f**(n) = O(n²)

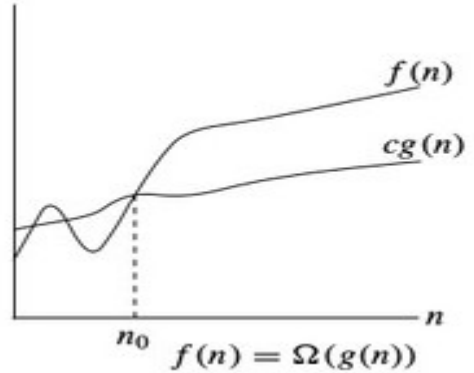


Asymptotic notation (Big O, Big **Ω, Big Θ)**

Ω Notation

- It denotes the lower bound of an algorithm i.e. the time taken by the algorithm can't be lower than this.
- In other words, this is the fastest time in which the algorithm will return a result.
- Its the time taken by the algorithm when provided with its best-case input.
- So, if a function is g(n), then the omega representation is shown as Ω(g(n)) and the relation is shown as:

```
Ω(g(n)) = { f(n): there exist positive constants c and n0
        such that 0 ≤ cg(n) ≤ f(n) for all n ≥ n0 }
```



Asymptotic notation (Big O, Big **Ω, Big Θ)**

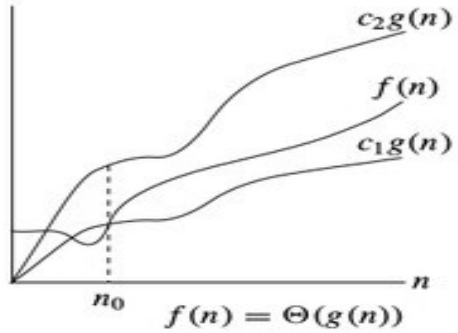
Θ Notation (theta)

- The Θ Notation is used to find the average bound of an algorithm i.e. it defines an upper bound and a lower bound, and your algorithm will lie in between these levels. So, if a function is g(n), then the theta representation is shown as Θ(g(n)) and the relation is shown as:

  Θ(g(n)) = { f(n): there exist positive constants c1, c2 and n0

  such that 0 ≤ c1g(n) ≤ f(n) ≤ c2g(n) **for** all n ≥ n0 }

- *The above expression can be read as* theta of g(n) is defined as set of all the functions f(n) for which there exists some positive constants c1, c2, and n0 such that c1*g(n) is less than or equal to f(n) and f(n) is less than or equal to c2*g(n) for all n that is greater than or equal to n0.



Growth rate functions

Function of Growth rate

Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

Examples

```
i=1
loop(i<=1000)
app code
i=i+1
end loop
n=1000
```

```
i=1
loop(i<10)
app code
i=i*2
end loop
```

```
i=1
loop(i<=n)
pf(i)
i=i+1
end loop
```

formula= no. of iterations proportional to loop factor
thus efficiency propotional to no. of iterations

1) i = 1
loop(i<=1000)
appln code
i = i+1
end loop
n=1000

(time)
∴ efficiency ∝ no. of iterations.
∴ efficiency = n.
∴ f(n) = n
∴ time complexity = O(n)

```
i=1
loop(i<10)
app code
i=i*2
end loop
```

iterations	value of i
1	1 2^0
2	2 2^1
3	4 2^2
4	8 2^3

thus $f(n)=Log_2 n$

3) i = 1

loop (i ≤ n)

pf (i)

i = i + 1

end loop

⇒ ∴ efficiency ∝ no. of iterat^{n}s.

∴ T. C. O(n).

Examples

the algortihm DOT has an eff of 5n. calculate the runtime efficiency for following program

i=1

loop(i<=n)

DOIT(....) 5n

i=i+1

end loop

Calculate runtime complexity for followi programm.

```
i = 1
loop (i ≤ n)
    DOIT (---) 5n
    i = i+1
end loop.
```

Iterations = Outerloop factor * innerloop fac

$= n * 5n$

$= 5n^2$

$\therefore T.C = O(n^2)$.

if the eff of an algo can be expressed as o(n)=n^2. calculate eff of the following program

=1

loop(i<n)

DOT(____) (DOIT=n^2)

i=i*2

end loop

Solution:

iterations=$Log_2 n$*$\mathbf{n^2}$

thus

efficiency=$Log_2 n$*$\mathbf{n^2}$

Recurrence relations

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs.

To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

A recurrence relation is an equation that recursively defines a sequence where the next term is a function of the previous terms

There are three methods for solving Recurrence:

1. Substitution Method
2. Recursion Tree Method
3. Master Method

Substitution method

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and shows that the guess is correct.

For example consider the recurrence T(n) = 2T(n/2) + n

We guess the solution as T(n) = O(nLogn). Now we use induction to prove our guess.

We need to prove that T(n) <= cnLogn. We can assume that it is true for values smaller than n.

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

<= 2cn/2Log(n/2) + n

= cnLogn - cnLog2 + n

= cnLogn - cn + n

<= cnLogn

Prove $T(n) = O(n \log n)$ $\quad T(n) \leq c \cdot n \log n$

$T(n) = 2T(n/2) + n \quad n > 1$

$T(n) \leq c \cdot n \log n$ put this in given recurrence relation equation

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

~~$\leq cn \log n - cn \log 2 + 1$~~

$\leq c \cdot 2\left(\frac{n}{2}\right) \log (n/2) + n$

$\leq c \cdot n \log (n/2) + n$

$\leq c \cdot n (\log n - \log 2) + n$

$\leq c\, n (\log n - 1) + n$

$\leq cn \log n - cn + n$

$\leq cn \log n - n(c-1)$

if $c \geq 1$

$\leq cn \log n + n(0)$

$\leq cn \log n$

Thus $T(n) = O(n \log n)$

Recurrence tree method

Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.

2. In general, we consider the second term in recurrence as root.

3. It is useful when the divide & Conquer algorithm is used.

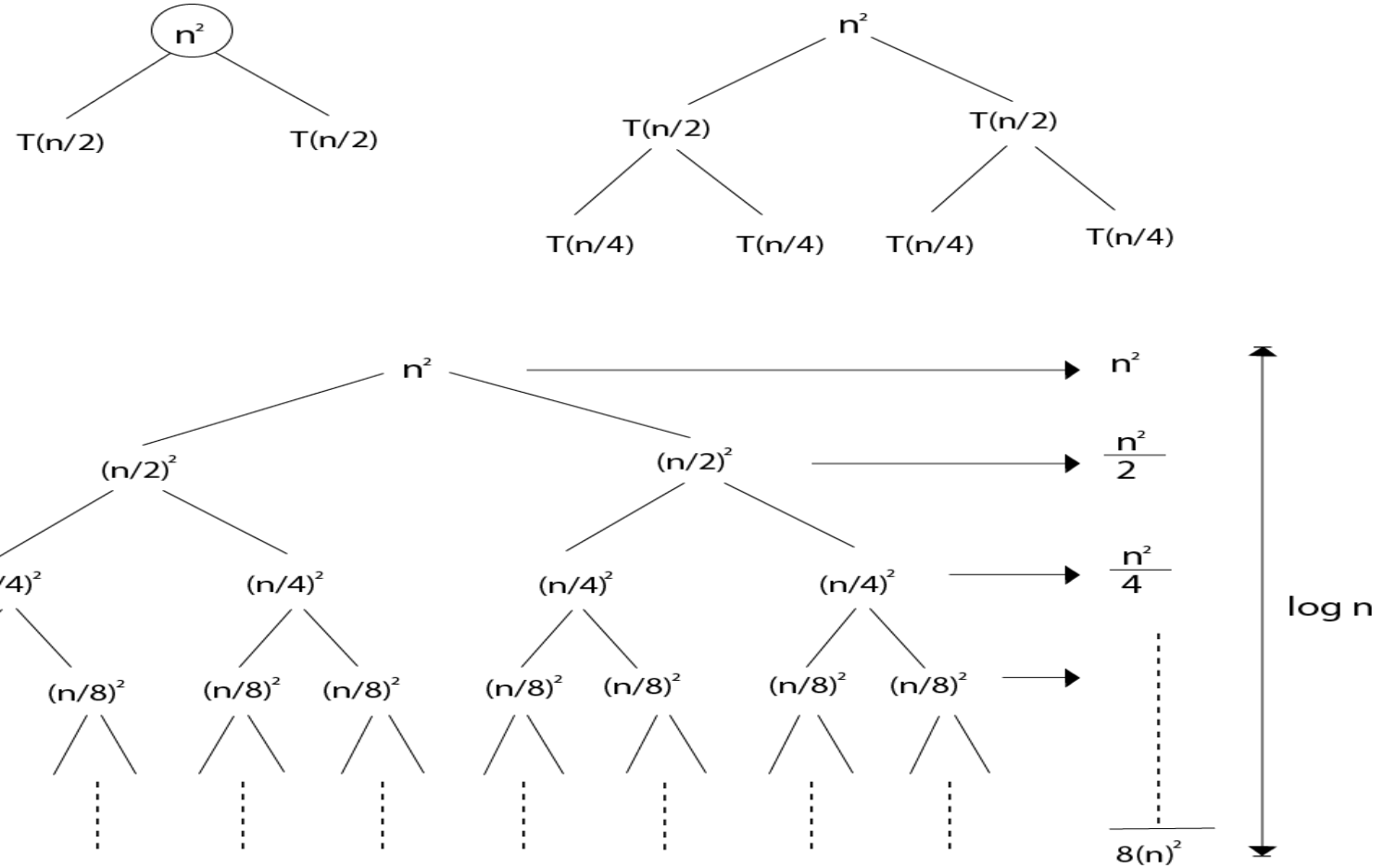
4. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.

5. 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.

6. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

Recursion tree method

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



$$T(n) = n^2 + \frac{n^2}{2} + \frac{n^2}{4} + \ldots\ldots \text{log n times.}$$

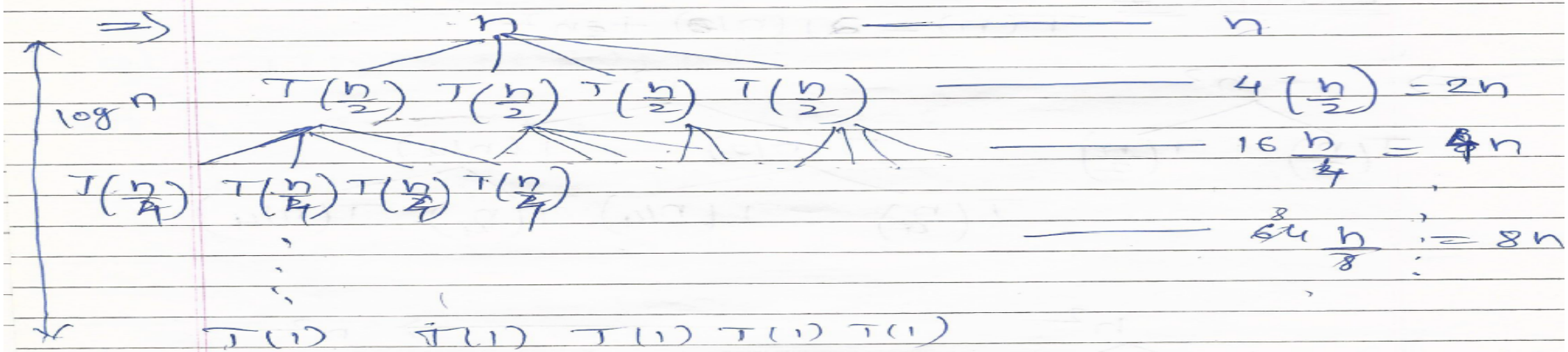
$$\leq n^2 \sum_{i=0}^{\infty} \left(\frac{1}{2^i}\right)$$

$$\leq n^2 \left(\frac{1}{1-\frac{1}{2}}\right) \leq 2n^2$$

$$\mathbf{T(n) = \theta n^2}$$

Example 2:-

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



$$\therefore T(n) = n + 2n + 4n + 8n \cdots \cdots + \log_2 n$$

$$= n(1 + 2 + 4 + 8 + \cdots \log n)$$

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

$$= n^2 - n$$

$$\therefore T(n) = \underline{\underline{\Theta(n^2)}}$$

Size of sub-problem at level-i $= n/2^i$

Suppose at last level size be

$n/2^i = 1$

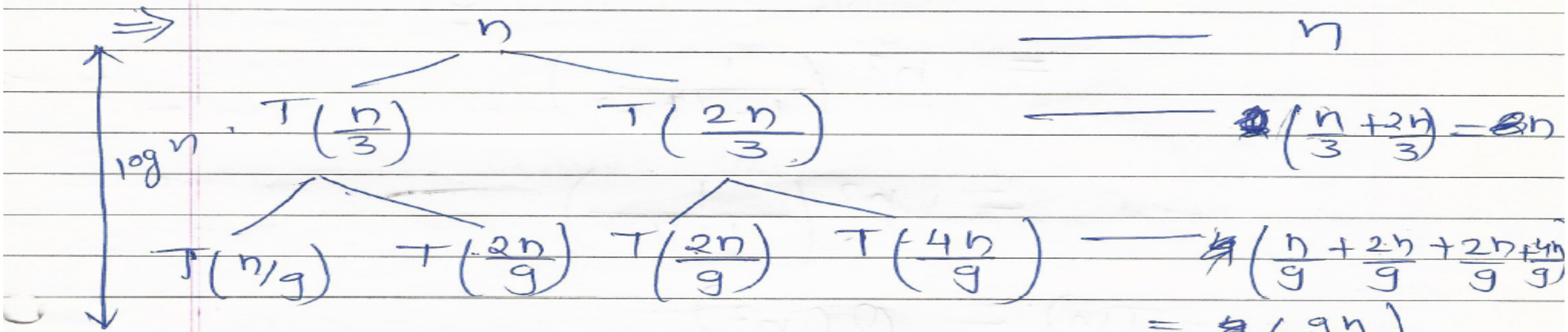
$n = 2^i$ $\quad i = \log$

* The subproblem size for a node at depth i is $n/2^i$.

Thus subproblem size hits $n=1$ when $n/2^i = 1$ or when $i = \log_2 n$

Example 3 :-

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



$$T(n) = n + n + n + n + \cdots + \log n.$$

$$= n(1 + 1 + 1 + 1 + \cdots + \log n).$$

$$\therefore T(n) = \underline{\Theta(n \log n)}$$

Master method

Master Method is a direct way to get the solution. The master method works only for following type of recurrences or for recurrences that can be transformed to following type.

$$T(n) = aT(n/b) + f(n) \text{ where } a >= 1 \text{ and } b > 1$$

- n is the size of the problem.
- a is the number of subproblems in the recursion.
- n/b is the size of each subproblem. (Here it is assumed that all subproblems are essentially the same size.)
- f (n) is the sum of the work done outside the recursive calls, which includes the sum of dividing the problem and the sum of combining the solutions to the subproblems.
- It is not possible always bound the function according to the requirement, so we make three cases which will tell us what kind of bound we can apply on the function.

Master Method

3 cases

$$T(n)=\begin{cases}\Theta\left(n^{\log_b a}\right) & f(n)=O\left(n^{\log_b a-\varepsilon}\right)\\ \Theta\left(n^{\log_b a}\log n\right) & f(n)=\Theta\left(n^{\log_b a}\right)\\ \Theta(f(n)) & f(n)=\Omega\left(n^{\log_b a+\varepsilon}\right)\text{AND}\\ & af(n/b)<cf(n)\ \text{forlargen}\end{cases}\begin{matrix}\\ \\ \varepsilon>0\\ c<1\end{matrix}$$

Master th Method

→ 3 cases

Cases 1 : running time dominated by cost at leaves

if $f(n) = O(n^{\log_b a - \epsilon})$, for $\epsilon > 0$

then $T(n) = \Theta(n^{\log_b a})$ for $\epsilon > 0$

In this case when we compare

$f(n)$ with $n^{\log_b a}$

then if $f(n) < n^{\log_b a}$

then case 1 is used.

Case 2 : Running time evenly distributed

If $f(n) = \Theta(n^{\log_b (a)})$

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

In this case if $f(n) = n^{\log_b a}$. if then apply case 2.

Case 3: Running time dominated by cost at root

if $f(n) = \Omega(n^{\log_b (a) + \epsilon})$

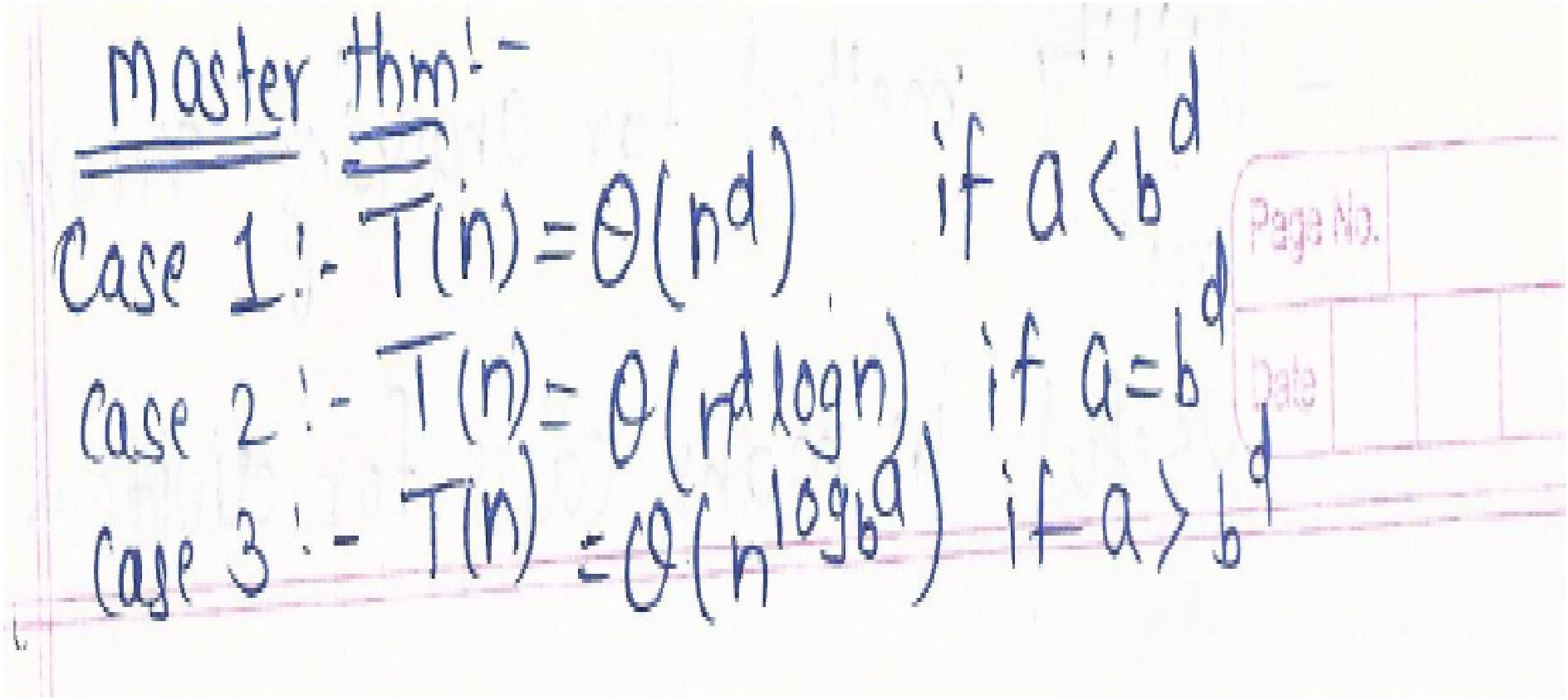
and $a \cdot f(n/b) < c \cdot f(n)$ for large n

then $T(n) = \Theta(f(n))$

So. if $f(n) > n^{\log_b a}$

then apply case 3.

Master Thm-Shortcut method



How to apply Master Method :-

1. Extract a, b & f(n) from given recurrence.

2. Determine $n^{\log_b(a)}$

3. Compare f(n) and $n^{\log_b(a)}$ asymptotically.

4. Determine appropriate master method case & apply it.

Example :-

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

1. $a = 2$, $b = 2$, $f(n) = n$.

2. Determine $n^{\log_b(a)} = n^{\log_2(2)}$

$= n^1 = n$.

3. Compare $n^{\log_b(a)} = n$

$f(n) = n$

both are equal.

4. Thus case 2: evenly distributed

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

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

$= \underline{\Theta(n \log n)}$

Short cut method for master theorem.

~~Case 1: $T(n) = \Theta(n^d)$ if $a < b^d$.~~

$T(n) = a \cdot T(n/b) + n^d$

Case 1: $T(n) = \Theta(n^d)$ if $a < b^d$

Case 2: $T(n) = \Theta(n^d \log n)$ if $a = b^d$

Case 3: $T(n) = \Theta(n^{\log_b a})$ if $a > b^d$.

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

$a = 2 \quad b = 2 \quad d = 1$

$a = 2 \equiv b^d = 2^1 = 2$

$\therefore$ case 2

$T(n) = \Theta(n^d \log n)$

$= \underline{\Theta(n^1 \log n)}$

$T(n) = 9T(n/3) + n.$

1. $a = 9, \quad b = 3, \quad f(n) = n$

2. $n^{\log_3 9} = n^{\log 3} = n^2$

3. $n^{\log_3 9} = n^2$

   $f(n) = n$

4. The $n^{\log_b a}$ is larger.

∴ Case 1.

∴ $T(n) = \Theta(n^{\log_3 9 - \epsilon})$

$= \Theta(n^{2-\epsilon})$

$= \Theta(n^2)$.