Module 1

Algorithm

An algorithm is a finite set of instructions that if followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

- Input : Zero or more quantities are externally supplied
- output : Atleast one quantity is produced
- Definiteness : Each instruction is clear & unambigious
- finiteness : If we trace out the instructions of an algorithm, then for all cases, the algorithm terminates after a finite no. of steps
- Effectiveness : Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil & paper

Space / Time Complexity

- Space Complexity of an algorithm is the amount of memory it needs to run to completion.
- Time complexity of an algorithm is the amount of computer time it needs to run to completion.

Performance Evaluation can be loosely divided into two major phases :

1) A priori estimates & (Performance analysis)
2) A posteriori testing (Performance measurement)

→ Space Complexity

* The space needed by each algorithms is seen t

be the sum of the following components

1) A fixed part that is independent of the characteristics of the i/p & o/p. The part typically includes the instruction space (i.e., space for the code), space for simple variables & fixed-size component variables, space for constants & so on.

2) A variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables (to the extend that this depends on instance characteristics) & the recursion stack space.

3) The space requirement s(p) of any algorithm p may therefore be written as

$$S(P) = C + Sp \quad \text{(instance characteristics)}$$

where c is a constant.

How to Analyse algorithm:

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e, to estimate the complexity function for arbitrarily large i/p. The term "analysis of algorithms" was coined by Donald Knuth.

- Algorithm analysis provides theoretical estimation for the required resources of an algorithm.
- It is the determination of the amount of time

& Space resources required to execute it

Performance Evaluation consist of

* Priory estimate or Performance or asymptotic analysis
* posteriori testing or performance measurement

→ A priori analysis

- It is an absolute analysis
- Independent of language of compiler & types of h/w
- It will give approximate answer.
- Uses asymptotic notations to represent how much time algorithm will take in order to complete its execution.

→ A Posteriori analysis

* It is a relative analysis
* dependent on language of compiler & types of h/w
* give exact answer.
* don't use asymptotic notations to represent the time complexity.

Time efficiency is analysed by determining the number of repetitions of the basic operation as a function of input size.

↓

It contribute towards the running time of the algorithm.

$$T(n) \approx C_{op}\, C(n)$$

RUNNING TIME — EXECUTION TIME — NUMBER OF TIMES BASIC OPERATION IS EXECUTED

where n is the input size

Best-Case, Average Case, Worst-Case

For some algorithms, efficiency depends on form of input:

- Worst-case
- Best case
- average case

Worst-case

* $C_{worst}(n)$
* maximum over inputs of size n

Best-case

* $C_{best}(n)$
* minimum over inputs of size n

Average-case

* $C_{avg}(n)$
* "average" over inputs
* Number of times the basic operation will be executed on typical input
* NOT the average of worst & best case.
* Expected no. of basic operations considered as a random variable under some assumption about the probability distribution of all possible i/p.

MATHematical Analysis (Frequency Count Method)

TIME → Time Complexity → • as a func. of i/p. size
• running time of algorithm.

SPACE → Space Complexity → • space, needed by an algorithm
• represented as func. of i/p size 'n'

Q)

Algorithm Sum(A, n)	Cost	Frequency
{		1
s=0	c_1	1
for(i=0; i<n; i++)	c_2	n+1 (1+n+1+n = 2n+2 = 2(n+1))
{		
s = s + A[i];	c_3	n
}		
return s;	c_4	1
}		

$T(n) = c_1 + c_2(n+1) + c_3(n) + c_4$ — c_1, c_2, c_3, c_4 = constants

$= c_1 + c_2 n + c_2 + c_3 n + c_4$

$= c_1 + n(c_2 + c_3) + c_2 + c_4$

$= n(c_2 + c_3)$

$= O(n)$ (Linear Complexity)

$S(n)$ List out the variables

A ⟶ n (array)

s ⟶ 1

i ⟶ 1

n ⟶ 1

$n+3 \approx O(n)$

Q. writing algo for adding nxn matrices. Find its time & space complexity

Algorithm Add(A, B, n)	Cost	Frequency
{		
for(i=0; i<n; i++)	c_1	n+1 | (2n+2)
{ for(j=0; j<n; j++	c_2	n(2n+2)

`c[i,j] = A[i][j] + B[i][j]`	c_3	$n \times n$
`}`		
`}`		

$T(n) = c_1(2n+2) + c_2(2n+2)n + c_3 n^2$

$= c_1 \cdot 2n + c_1 2 + c_2 \cdot 2n^2 + c_2 \cdot 2n + c_3 n^2$

$= n(2c_1 + 2c_2) + 2c_1 + n^2(2c_2 + c_3)$

$= n + n^2$

$= n^2 \approx O(n^2)$

$S(n)$

A $\longrightarrow n^2$
B $\longrightarrow n^2$
C $\longrightarrow n^2$
i $\longrightarrow 1$
j $\longrightarrow 1$
n $\longrightarrow 1$ $\Rightarrow 3n^2 + 3 = n^2 \approx O(n^2)$

Q. Multiplying two Square matrix

Algorithm multiply (A, B, n)	Cost	Freq.
{		
`for(i=0; i<n; i++)`	c_1	$2n+2$
`for(j=0; j<n; j++)`	c_2	$(2n+2)n$
`c[i,j] = 0`	c_3	$n \times n$
`for(k=0; k<n; k++)`	c_4	$(2n+2)n^2$
`c[i,j] = c[i,j] + A[i,k] + B[k,j]`	c_5	n^3
}		

$T(n) = c_1(2n+2) + c_2(2n+2)n + c_3 n^2 + c_4(2n+2)n^2 + c_5 n^3$

$= 2c_1 n + 2c_1 + 2c_2 n^2 + nc_2 + c_3 n^2 + 2c_4 n^3 + 2c_4 n^2 + c_5 n^3$

$= n(2c_1 + c_2) + 2c_1 + n^2(2c_2 + c_3 + 2c_4) + n^3(2c_4 + c_5)$

$= n + n^2 + n^3 = n^3 \approx O(n^3)$

S(n)

A ⟶ n^2
B ⟶ n^2
i ⟶ 1
j ⟶ 1
K ⟶ 1
C ⟶ n^2
n ⟶ 1

$= 3n^2 + 4$
$= n^2$
$\approx O(n^2)$

Q. 1)
```
for(i=0; i<n; i++)
  {
   stmt 1
  }
```
$= O(n)$

2)
```
for(i=n; i>0; ~~i++~~ i--)
  {
   stmt 2
  }
```
$= n+1 = O(n)$

3)
```
for(i=1; i<n; i=i+2)
  {
   stmt 3
  }
```
$= n/2 = O(n)$

4)
```
for(i=1; i<n; i=i+20)
  {
   stmt 4
  }
```
$= n/20 = O(n)$

Q.
```
for(i=1; i<n; i=i*2)
  {
   stmt;
  }
```

Suppose n = 8

1 ⟶ 2	3
2 ⟶ 4	
4 ⟶ 8	
8 ⟶ X	K

$= 2^{③K}$

$n = 2^K$

$n = 2^K$

$K = \log_2 n$

$\approx O(\log_2 n)$

$n = 10$

$n = K/2 = 10$

10 b/w 2^3 & 2^4 = $n = 2^4$

take ceil ⟶ $K = \log_2 10 = 4$

$\log_2 10 = 4$

$i = i * 3 \longrightarrow O(\log_3 n)$

$i = i * 16 \longrightarrow O(\log_{16} n)$

Q.
```
for (i=0; i<n; i++)
for (j=0; j<i+j++)
  { stmt;
  }
```

i =	j =	
0	0	0
1	0 1	1
2	0 1 2	2
3	0 1 2 3	3

$= 1+2+3+\cdots+n$

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

$= \frac{n^2+n}{2} \approx \underline{\underline{O(n^2)}}$

Q.
```
p=0
for (i=1; p<=n; i++)
  {
    p=p+i
  }
```

Worst, Best & Average case analysis of algorithm

Here we assume that element can be at any position & is equally likely.

Let 'p' be the probability that for Successful Search, '1-p' be the probability that for unsuccessful Search.

$p \longrightarrow$ Successful Search

$1-p \longrightarrow$ Unsuccessful Search

$\boxed{P_i = p/n}$ = successful search for every possible equal.

$P_i = P/n$

1 2 3 4 5

element to find is equal

in pos 1 → 1/5
2 → 1/5
3 → 1/5

$$T_{avg}(n) = \sum P_i * x_i$$

$$= \left[1 \times \frac{P}{n} + 2 \times \frac{P}{n} + 3 \times \frac{P}{n} + \dots\right]$$

$$= \frac{P}{n}\left[1 + 2 + 3 + \dots + n\right]$$

$$= \frac{P}{n} \times \frac{n(n+1)}{2} = \boxed{\frac{P(n+1)}{2}}$$

Successful search, $P = 1$, $= \frac{1(n+1)}{2} = \frac{n+1}{2}$ //

Unsuccessful search, $T_{avg}(n) = (1-P)n$

$P = 0$ $= (1-0)n = n$ //

n	n^2	2^n
1	1	2
2	4	4
4	16	16
8	64	256
16	256	65536
100	10000	2^{100}

ASYMPTOTIC NOTATIONS

To enable us to make meaningful statements about time & space complexities of an algorithm, asymptotic notations are used.

Execution time of an algorithm depends on the

1) Instruction set
2) Processor Speed
3) Disk I/O Speed

Hence, we estimate the efficiency of an algorithm asymptotically.

- Asymptotic notations are mathematical tool to represent the time complexity of algorithms for asymptotic analysis.

↓

To measure the efficiency of algorithms that
- don't depend on machine specific constants
- don't require algorithms to be implemented
- time taken by programs to be compared

- Important asymptotic notations include

1) Big Oh (O)
2) Omega (Ω)
3) Theta (Θ)

* Big Oh Notations

- Let $f(n)$ & $g(n)$ be two functions mapping integers to real numbers.
- A function $f(n) = O(g(n))$ iff $f(n) \leq c \cdot g(n)$ for some $n \geq n_0$ & c.

where n_0 & c are constants which is greater than 0

Eg: $f(n) = 2n + 3$

we know for $O(n)$, $f(n) \leq c \cdot g(n)$

$2n + 3 \leq c \cdot g(n)$

$2n + 3 \leq 5n \Rightarrow n = 1$ & $c = 5$

if $n=1$, $5 = 5$

→ $f(n) = 2n^2 + 3n + 5$

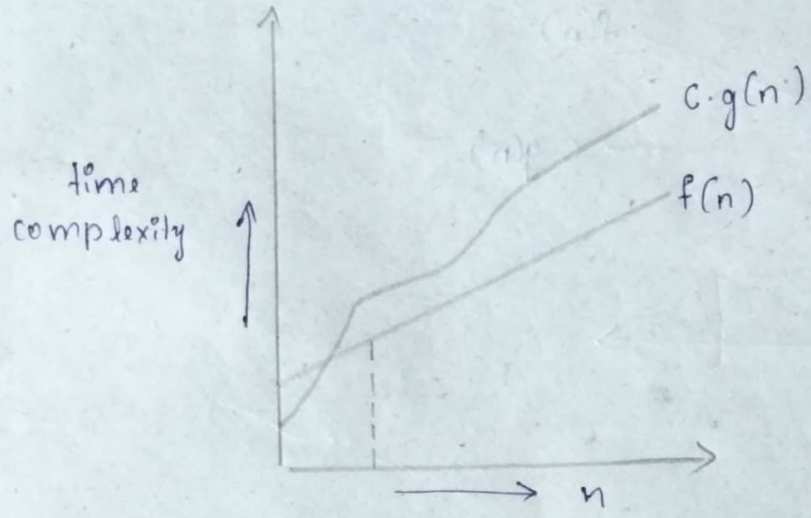
$2n^2 + 3n + 5 \leq c \cdot g(n)$

$2n^2 + 3n + 5 \leq 10n^2$

$n = 1$

$10 \leq 10$

Then $n_0 = 1$ & $c = 10$.



- function g(n) is an **upper bound** for function f(n), as g(n) grows faster than f(n) for all $n >= n_0$.

* Big-Omega(Ω) notation

- The function $f(n) = \Omega(g(n))$ iff $f(n) \geq c \cdot g(n)$ for all $n >= n_0$.

- Ω notation represents the **lower bound** of the running time of the algorithm. Thus, it provides the best case complexity of an algorithm.

- for 'Ω' notation,

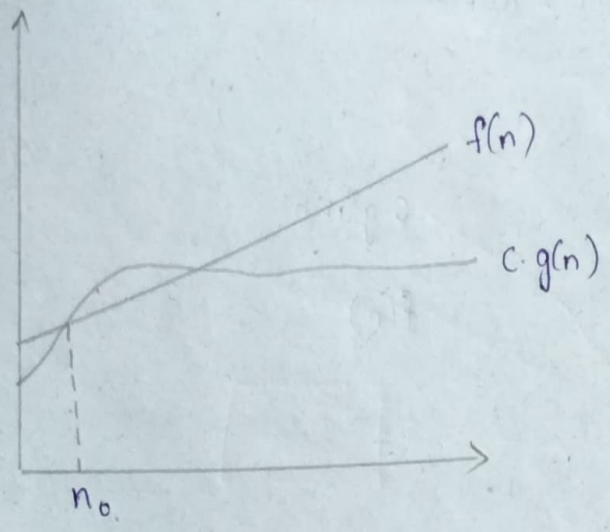
$f(n) \geq c \cdot g(n)$ is a function f(n) belongs to the set $\Omega(g(n))$ if there exist a +ve constant c such that it lies above $c \cdot g(n)$ for sufficiently large n.

- For any value of n, the minimum time required by the algorithm is given by $\Omega(g(n))$.

Eg: $f(n) = 3n + 2$

$f(n) \geq c \cdot g(n)$

$3n + 2 \geq 3n$ for all $n >= 1$, $c = 3$ $g(n) = n$ $n_0 = 1$

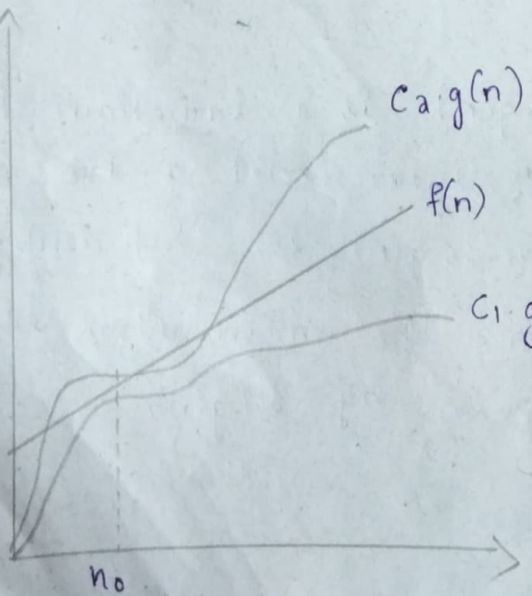


* Big-Theta (Θ) Notation

- Θ notation encloses the func. from above & below.
- Used for analyzing the average case complexity.
- Function $f(n) = \Theta(g(n))$ iff there exist a positive constants c_1, c_2 & n_0 such that

$$c_1 g(n) <= f(n) <= c_2 g(n) \quad \text{for all } n >= n_0$$

- $f(n) = \Theta(g(n))$ states that $g(n)$ is both upper & lower bound on the value of $f(n)$ for all $n >= n_0$.



Q. Suppose $f(n) = \frac{1}{2}n^2 - 3n$. S.T $f(n) = \Theta(n^2)$

- we know, for Θ notation,

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)$$

Here $f(n) = \frac{1}{2}n^2 - 3n$ $\quad g(n) = n^2$

$$f(n) \geq c_1 \cdot g(n)$$

$$f(n) \leq c_2 \cdot g(n)$$

$$\frac{1}{2}n^2 - 3n \leq c_2 \cdot n^2$$

Suppose $n = 1$, $\quad \frac{1}{2} - 3 \leq c_2$

$$-\frac{5}{2} \leq c_2 \quad \times$$

$n = 2$, $\quad \frac{1}{2} \times 4 - 3 \times 2 \leq c_2 \cdot 4$

$$2 - 6 \leq c_2 \cdot 4$$

$$-1 \leq c_2 \quad \times$$

$n = 7$, $\quad \frac{1}{2} \times 7^2 - 21 \leq c_2 \cdot 49$

$$\frac{\cdot 1}{14} \leq c_2 \quad \checkmark$$

Q. $f(n) = 3n^2 + 4n - 2$ S.T $f(n) = \Theta(n^2)$

- $f(n) \leq c_2 \cdot g(n)$ & $f(n) \geq c_1 \cdot g(n)$

$$f(n) \geq c_1 g(n)$$

$$3n^2 + 4n - 2 \geq c_1 (n^2)$$

$n = 1$, $\quad 3 + 4 - 2 \geq c_1$

$$3n^2 + 4n - 2 \leq c_2 (n^2)$$

$$5 \leq c_2$$

3/12

Mathematical Analysis of Recursive algorithm.

All recursive algorithms must obey three laws :

i) A recursive algo. must have a base case

ii) It must change its state & move towards base ca

iii) It must call itself, recursively.

Stopping condition or base case occur when :

> there has been no improvement in the popula for x iteration.

> we reach an absolute no of generations.

> objective func. value has reached a certain pre-defi value.

General plan for analyzing time efficiency of recursiv algorithm

1) decide on a parameter indicating input size
2) Identify algorithm's basic operation
3) Check whether no of times basic operation is executed
4) Set up a recurrence relation.
5) Solve the recurrence

General algorithm should ~~contain~~

* Identify the base ~~the~~ case
* write base case in 'IF' condition
* else part in recursive case.

Eg : n!

Algorithm fact (int n)

if (n == 0)

return 1;

```
else
    return n x fact(n-1)
```

Recurrence relation / Recurrence Equation.

- A recurrence is an equation or inequality that describes a function in terms of its values on smaller i/p.
- Useful for expressing the running time of recursive algorithms.

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + cn & n > 1 \end{cases}$$

- General form:

$T(n) = 0$ if $n = n_0$ → Base Case

$T(n) = aT(n) + g(n)$ for $n > n_0$

- a → no. of times recursive func. is called.
- $g(n)$ → time required for doing other operation

Recurrence eqn for factorial

$T(n) = 0$ if $n = 0$

$T(n) = 1 \cdot T(n-1) + b$ for $n > 0$

(b → constant)

Three different methods for Solving recurrences:

> Substitution method
> Iteration method
> Master method.

(1)

Iteration method :

Here, we expand the recurrence by backward substitution & express it as summation of terms dependent only on 'n' & initial conditions.

Steps

1. Expand the recurrence k times
2. Work some algebra to express as a summation
3. Evaluate the summation

$$\begin{cases} T(n) = c & \text{for } n = 0 \quad \text{---------- (1)} \\ T(n) = T(n-1) + b & \text{for } n > 0 \quad \text{---------- (2)} \end{cases}$$

* We use method of backward substitution in this method to solve the problem efficiently.

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

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

Sub the value of $T(n-1)$ in this eqn

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

$= T(n-2) + 2b$ //

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

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

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

General formula : $\boxed{T(n) = T(n-k) + kb}$ ---------- (3)

In the above Equation, when base condition is reached, put $n - k = 0$

$n = k \Rightarrow \boxed{k = n}$

Substitute the value of k in (3)

$T(n) = T(n-n) + nb = T(0) = nb$ ——— (4)

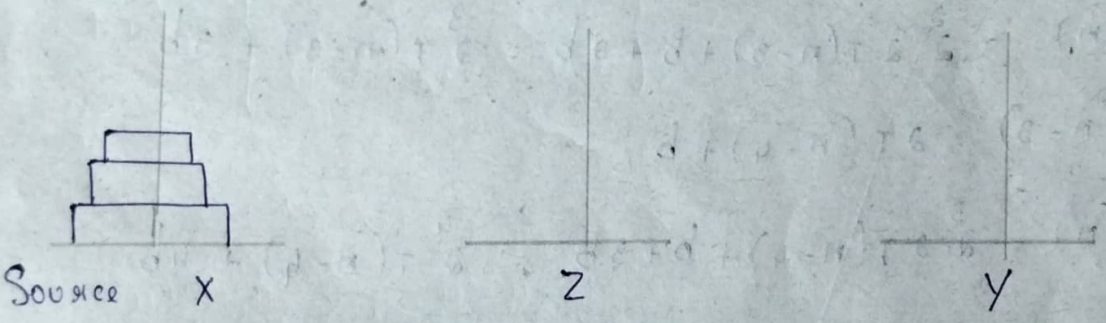
From (1), $T(0) = c$.

Substitute in (4)

$T(n) = c + nb$, where c & b are constants.

$T(n) = O(n)$

Recursive soln for Tower of Hanoi



Hanoi (x, y, z)

x → source, y → dest^n, z → temporary dest^n

1. Move (N-1) disk from Source (x) to temp (z)
2. Move N^{th} disk from Source (x) to dest^n (z)
3. Move (N-1) disk from temp (z) to dest^n (y)

```
Algorithm hanoi (n, x, y, z)
                 ↓ no. of disk
{
  if (n = 1)
    print (" move disk 1 from x to y)
  else
    hanoi (n-1, x, z, y)
    print (" move disk n from x to y)
    hanoi (n-1, z, y, x)
}
```

Recurrence Eqn :

$T(n) = c$ for $n = 1$ ——— (1)

$T(n) = 2T(n-1) + b$ for $n > 1$ ——— (2)

↳ hanoi() is used 2 times

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

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

$T(n) = 2 \cdot 2T(n-2) + b + b = 2^2 T(n-2) + 2b$

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

$T(n) = 2^2 \cdot 2T(n-3) + b + 2b = 2^3 T(n-3) + 3b$

$T(n-3) = 2T(n-4) + b$

$T(n) = 2^3 \cdot 2T(n-4) + b + 3b = 2^4 T(n-4) + 4b$

General form :

$T(n) = 2^k T(n-k) +$

✓ $T(n) = 2T(n-1) + b$

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

✓ $T(n) = 2(2T(n-2) + b) + b = 4T(n-2) + 2b + b$

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

✓ $T(n) = 2(2(2T(n-3) + b) + b)) + b =$

$= 2(2(4(2T(n-3) + b) + 2b + b$

$= 8T(n-3) + 4b + 2b + b$

$T(n-3) = 2T(n-4) + b$

✓ $T(n) = 8(2T(n-4) + b) + 4b + 2b + b$

$= 16T(n-4) + 8b + 4b + 2b + b = 2^4 T(n-4) + b[1+2+4+8]$

General form:

$$2^k T(n-k) + b(2^k - 1) \longrightarrow (3)$$

when base case is reached, $n = 1$

$$n - k = 1$$

$$k = n-1 \rightarrow \text{sub. in } (3)$$

$T(1) = c$

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

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

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

$$= c \cdot 2^{n-1} + b \cdot 2^{n-1} - b$$

$-1 = 1/2$

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

$$= \frac{2^n}{2}(b+c) - b$$

$$= 2^n/2 \approx \boxed{T(n) \approx O(2^n)}$$

Q. Algorithm Bin(n)

HW

```
{
  if(n = 1)
  return 1
  else
  return bin((n/2) + 1)
}
```

- Recurrence eqn:

$T(n) = c$ for $n = 1$ ——— (1)

$T(n) = T(n/2) + b$ for $n > 1$ ——— (2)

The presence of $(n/2)$ in the function's argument makes backward substitution stumble on values of n that are not powers of 2.

Therefore, the standard approach to solve such recurrence is to solve by taking,

$$n = 2^k$$

& use Smoothness rule.

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

$$T(2^k) = T(2^{k-1}) + 1 \quad \text{for } k > 0 \quad \text{---(1)}$$

$$T(2^0) = 0 \quad \text{for } k = 0 \quad \text{---(2)}$$

By backward Substitution :

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

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

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

$$T(2^{k-2}) = T(2^{k-3}) + 1$$

$$T(2^k) = T(2^{k-3}) + 3$$

general form,

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

$$n - k = 1$$

$$k = n - 1$$

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

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

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

we know $n = 2^k$

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

$$k = \log_2 n \approx O(\log n)$$

(2) Master's Method

Theorem : Let 'a' be a positive integer, let 'b' be an integer greater than 1, & let f be a real-valued func. defined on perfect powers of b. for all perfect powers of n of b, define T(n) by the recurrence

$$T(n) = aT(n/b) + f(n) \quad \text{---------- (1)}$$

with a non-negative initial value T(1).

~~(1) can be~~ Master method is used to solve the recurrence of the form $T(n) = aT(n/b) + f(n)$. $a \geq 1$ & $b \geq 1$.

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

f(n) is asymptotically smaller or larger than $n^{\log_b a}$ by a polynomial factor n^2

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

Three cases:

(1) Case 1 :

If $f(n) = O(n^{\log_b a - \epsilon})$ for some $\epsilon > 0$, then $T(n) = \Theta(n^{\log_b a})$

(2) case 2 :

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

(3) case 3 :

If $f(n) = \Omega(n^{\log_b a + \epsilon})$ for some $\epsilon > 0$, and if

REGULARITY CONDITION ← $af(n/b) = c \cdot f(n)$ for $c < 1$ and all sufficiently large n, then $T(n) = \Theta f(n)$

Q. Given that $T(n) = 9T(n/3) + n$ using master's theorem.

We know, $T(n) = aT(n/b) + f(n)$

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

$n^{\log_b a} = n^{\log_3 9} = n^{\log_3 (3)^2} = n^{2 \log_3 3} = \underline{n^2}$

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

$n = n^2$ (case 1)

$O(n^{\log_b a - \epsilon}) = \Theta(n^{\log_b a})$

$\Rightarrow \Theta(n^2)$ //

Q. $T(n) = T(2n/3) + 1$

$a = 1 \quad b = \frac{3}{2} \quad f(n) = 1$

$n^{\log_b a} = n^{\log_{\frac{3}{2}} 1} = n^0 = 1$ //

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

$1 = 1$

case 2

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

Q. $T(n) = 3T(n/4) + n \log n$

$a = 3 \quad b = 4 \quad f(n) = n \log n$

$n^{\log_b a} = n^{\log_4 3} = n^{0.793}$

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

$n \log n \geq n^{0.793}$

case 3 $f(n) = \Omega(n^{\log_b a + \epsilon})$

Regularity condition: $af(n/b) = c \cdot f(n)$ for $c < 1$

$f(n) = n \log n \Rightarrow f(n/b) = (n/4) \log (n/4)$

3. $\left(\frac{n}{4} \log (n/4)\right) = c(n \log n)$

$3\left(\frac{n}{4} \log \left(\frac{n}{4}\right)\right) \leq \frac{3}{4}(n \log n)$ $\left(c = \frac{3}{4}\right)$ ✗

$\frac{a}{b^k} < 1$ $= \frac{3}{4^1} < 1$

$= \frac{3}{4} < 1$ ✓

$= \theta(f(n)) = \theta(n \log n)$

k = power of log

$f(n) = n \log^{1} n$

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

$a = 4 \quad b = 2 \quad f(n) = n$

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

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

$n < n^2$

case 1 $\theta(n^{\log_b a - \omega}) = \theta(n^{\log_b a}) = \theta(n^2)$

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

$a = 4 \quad b = 2 \quad f(n) = n^2$

$f(n) = n \log^{b} a$

$n^2 = n^2$

case 2 $= f(n) = \theta(\log_b^a \cdot \log n) = \theta(n^2 \log n)$

Q) $T(n) = 4T(n/2) + n^3$

$a = 4 \quad b = 2 \quad f(n) = n^3$

$f(n) = n \log_b a$

$n^3 > n^2$

Case 3

$f(n) = n(n^{\log_b a} + \omega)$

$a \cdot f(n/b) = c \cdot f(n)$

$4 f(\frac{n}{2}) = c \cdot n^3$

$4 \cdot (\frac{n}{2})^3 = c \cdot n^3$

$\Rightarrow c = 1/2$

$T(n) = \cancel{\Theta f \log}\ \Theta(f(n)) = \underline{\Theta(n^3)}$

(HW)

1) $T(n) = 5T(n/2) + \Theta(n^3)$

2) $T(n) = 27\, T(n/3) + \Theta(n^3/\log n)$