(Lecture#1)

DATA STRUCTURE AND ALGORITHM:

DATA:

* Data is defined as Raw facts and figures. (Material)

* It can also be referred as unarranged / unorganized information.

* Everything we see, we perceive is data. (Birds eye view)

* In simple words, Data is raw material.

* Any data also have some figures
  - → min length
  - → max length
  - → avg count.

There is no conclusion about these raw materials.

NOTE:

Computer is a data processor.

→ It takes data as input and process it.

Scanned with CamScanner

STRUCTURE:

* Form to manage data.
* Data is organized in any form i.e **structures**.
  These structures could be shape, Architecture, form,.

* After organizing the data (which is initially in scattered form), converts into well-organized form.

* Structure is used to organize data in well-organized form.

* Structure is blue-print.
* We choose structure wisely.

INFORMATION: Data after processing (organizing) becomes "**INFORMATION**".

* Information is processed data.
  (Basically processed facts and figures).

* Data and information are technically different.

* These structures are the storage of data.

Scanned with CamScanner

NOTE:

→ If we use good structure, then processing will be efficient.

→ To store data in memory, first we decide appropriate structure for data.
And overall organization of data would become efficient (efficient processing).

DATA STRUCTURES:

★ Arrangement of data in computer's memory (Main memory / RAM), for further processing.

★ Here, at this stage, data is just arranged but we need to process it using Algorithms.

ALGORITHMS:

We processes data by using efficient algorithm.

★ ~~too~~ Step by step execution of your program.

Scanned with CamScanner

* Sequence-wise processing of data.

* Algorithms are tools and techniques for processing data (as well as arrangement of data in computer's memory).

* Algorithms are those programs that stop / close / hold and gives an output.

* Infinite loops are not Algorithms.

* We follow well-defined Algorithms.

* We select efficient (both) algorithms and data-structure.

Scanned with CamScanner

DATA STRUCTURE TYPES/CATEGORIES:

Data structure can be divided into two categories:

1) Linear Data structure
2) Non-Linear Data structure.

1) Linear DS	2) Non-Linear DS
• Array (mostly used DS)	• Table
• Linked list	• Tree (It is also a graph)
• Stack	• Graphs
• Queue	• Sets

Scanned with CamScanner

Lecture #02

PERFORMANCE ANALYSIS OF ALGORITHM:

★ We will see after how many steps (time) our program will executes.

➡ Complexity of Algorithm:

★ Complexity of Algorithm is measured in terms of
- time complexity
- space complexity

• TIME COMPLEXITY: (NO. OF STEPS / STEPS COUNT)

The amount of time taken by an algorithm to execute itself.

★ This time is basically the number of steps or step count.

★ Whenever we analyse an algorithm we do not consider the hardware of it. (Independent of hardware)

★ Time complexity is independant of hardware, machine or any operating system.

★ It gives exact measure for its execution.

★ If we see about hardware it is called "PERFORMANCE MEASUREMENT".

DENOTION OF TIME COMPLEXITY:

$t_{program}(n)$

$n \rightarrow$ input (size)

Scanned with CamScanner

NOTE:

★ We assume;

$\boxed{\text{int a}}$ = 5 ; mostly it takes '2' steps to execute.

→ But we mostly consider Counts/Measures/ Steps per execution. (S/e)

★ STEP PER EXECUTION : (S/e)

- How many steps programs require to execute.

Thus,

We assume;

int a = 5; executes in 1 step.

This will be called Step per execution. (S/e)

★ FREQUENCY: (f)

- How many times program / specific step repeats.

★ TIME COMPLEXITY : (TOTAL)

- If we multiply 's/e' and 'f' then we get time complexity (t)

Scanned with CamScanner

(1) EXAMPLE: We have given a code and we need to count its time complexity.

~~void test () { → } NOT INCLUDED IN STEP COUNT.~~

NOT Included in step count as well as in frequency.

Statement No.		f	s/e
	void test ()		
1-	{ → Not execute	1	0
2-	{ int a, b; → Not matter how many variables declare in one line, we always takes its s/e 1.	1	1
3-	Cin >> a >> b;	1	1
4-	Cout << a+b;	1	1
5-	} → Brachets does not execute	1	0

- We have just considered that these steps are executing in one step only.

NOTE:

- Brackets and comments does not execute. So, there s/e is '0'.

Scanned with CamScanner

Now, To measure the time complexity (t_n) of test(), we will multiply 'S/e' AND 'f' and add the answers.

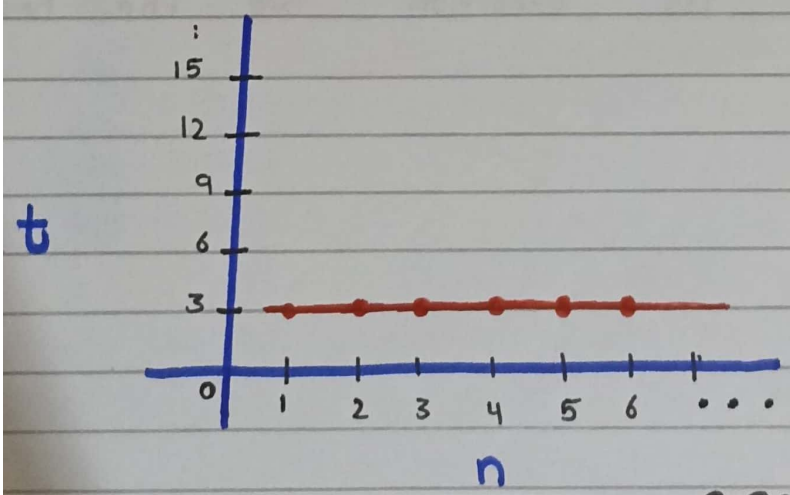
STATEMENT NO..	S/e	f	S/e x F = TOTAL
1-	0	1	0
2-	1	1	1
3-	1	1	1
4-	1	1	1
5-	0	1	0

$\boxed{t_{test}(n) = 3}$

This is time complexity of our program test. (step count of test()).

This time complexity means each time this (test) program executes, it takes '3' steps to give output / executes (stops)

GRAPH OF TIME COMPLEXITY OF test().



$\boxed{t_{test}(n) = 3}$

GROWTH RATE:

The "GROWTH RATE" of this function is "CONSTANT" for all inputs.

Scanned with CamScanner

② EXAMPLE :

→ If we are given a loop.

```
for (int i=0 ; i < 5; i++)
    temp;
```

	S/e	f	TOTAL
for (int i=0 ; i < 5; i++) } As statement is '1'	1	5 + 1 = 6 (TRUE time, False time)	6
temp;	1	5	5

$t_{(n)} = 11$

The growth rate of this function is "CONSTANT".

★ A loop will always executes '1' extra time bcz it will also check for (last) false comparision.

IN ABOVE CASE:

i	CHCK	RUN
0	✓	✓
1	✓	✓
2	✓	✓
3	✓	✓
4	✓	✓
5	✓	✗

★ An algorithm / program which takes constant time for its execution are the best one.

Scanned with CamScanner

③ EXAMPLE:

	S/e	f	TOTAL
for(i=0; i<7; i++) —	1	8	8
temp; —	1	7	7
for (j=0; j<5; j++) —	1	6	6
start; —	1	5	5

Lower ~~upper~~ Limit ↓ ~~Lower~~ upper Limit ↓

$t(n) = 26$

↳ GROWTH RATE OF THIS FUNCTION IS ALSO "CONSTANT"

NOTE:

★ We can also use formula to calculate frequency of loop statement.

FORMULA: ⇒ U.L − L.L + 1

$7 - 0 + 1 = 8$

OR

$5 - 0 + 1 = 6$

Scanned with CamScanner

GENERALIZED TIME COMPLEXITY OF LOOP:

④ **EXAMPLE:** Consider a general loop.

```
for(i=0; i<n; i++)
    temp;
```

→ $n+1$

→ n

$t(n) = 2n+1$

HOW

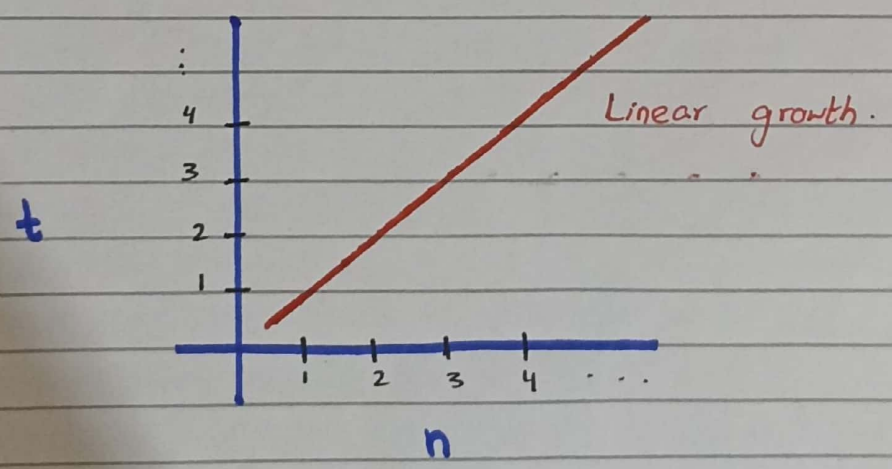
→ U.L − L.L + 1

$n - 0 + 1 = n+1$

$\boxed{t(n) = 2n+1}$ → This 1 program has linear growth rate.
Bcz 'n' (input)'s power is '1'. We mostly only see 'n' and its power.

- When $t = 1$
  $t(1) = 3$
- When $t = 4$
  $t(4) = 9$
- When $t = 100$
  $t(100) = 201$

*Output increases as value of input increases.



Scanned with CamScanner

⑤ **EXAMPLE:** Considering two loops.

```
for(i=0; i<n; i++)      →   n+1
    temp;               →   n

for(j=0; j<n; j++)      →   n+1
    stat;               →   n
```

$1 + t(n) = 4n+2$

$$\boxed{t(n) = 4n+2}$$

NOTE:

★ If we have function calls, we calculate the time complexity of functions (that are being called) separately, and then add it in overall (original time complexity of function)

LIKE:

```
Calculate () {
    add ();      t_add(n)
    sub ();      t_sub(n)
    mul ();      t_mul(n)
    div ();      t_div(n)
}
```

$$t_{calculate}(n) = t_{add}(n) + t_{sub}(n) + t_{mul}(n) + t_{div}(n)$$

Scanned with CamScanner

⑥ EXAMPLE: OF main() having a function call.

		s/e	f	TOTAL (s/e × f)
int main()				
1- {	→	0	1	0
2- int a=3, b=4;	→	1	1	1
3- int s= add (a,b);	→	$1 + t_{add}(n)$	1	$1 + t_{add}(n)$
4- cout << s ;	→	1	1	1
5- return 0;	→	1	1	1
6- }	→	0	1	0

↳ This '1' is for temp (OR here for 's') For storing the return value of function.

$$t_{main}(n) = \underbrace{4 + t_{add}(n)}$$

↳ It is not exact step count. Bc it has unknown $t_{add}(n)$. As, $t_{add}(n)$ is an unknown which we must have find, then it will be exact time complex

```
int add (int a, int b)
{
    return a+b;
}
```

→ 1

$$\boxed{t_{add}(n) = 1}$$

So,

$$t_{main}(n) = 4 + 1$$
$$= 5$$

$$\boxed{t_{main}(n) = 5}$$

↳ "The growth rate of this function is also constant

Scanned with CamScanner

NESTED LOOPS:

We will see how to calculate time complexity of nested loops.

① Example:

Consider a nested loop.

```
for (i=0; i<n ; i++)          → n+1
    for(j=0 ; j<n ; j++)      → n x (n+1) = n^2 + n
        temp ;                → n x n = n^2
```

* This is the main statement we mostly interested (calculate time complexity) of this term.

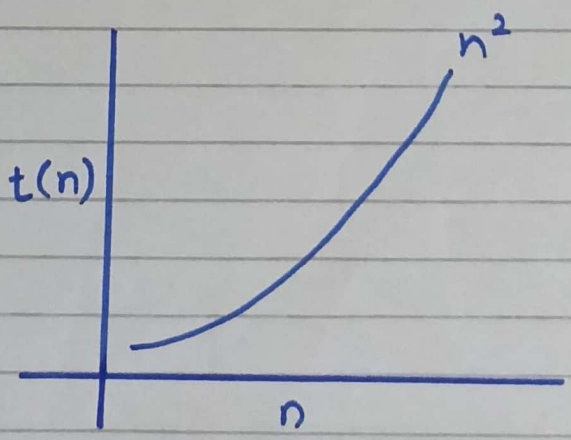
$$\boxed{t(n) = 2n^2 + 2n + 1}$$

* The growth rate of this algorithm is "quadratic"

CHECK:

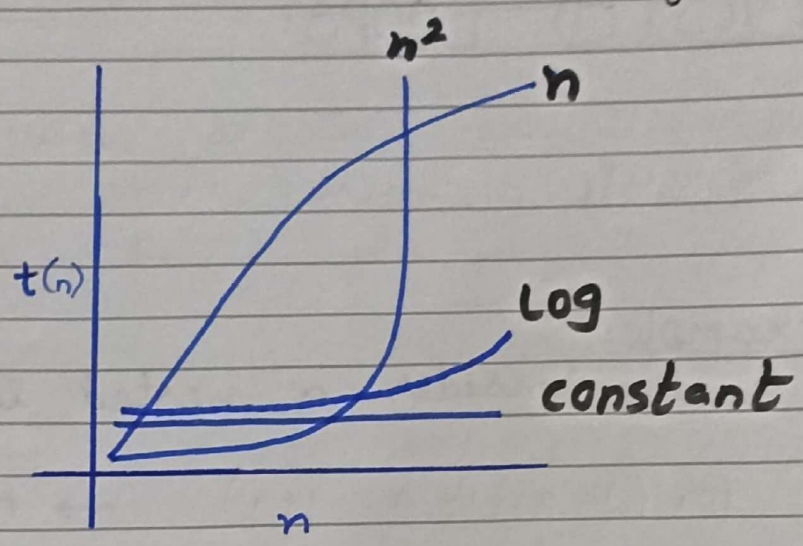
i	j
0	0 ... n+1
1	0 ... n+1
2	0 ... n+1
⋮	⋮
n	0 ... n+1

GRAPH:



Scanned with CamScanner

GRAPH OF TIME COMPLEXITIES (n, n^2, constant, log):



NOTE:

* Every statement inside loop executes one time less than the loop.

* Best time complexity is "LOGARITHMIC" because it increases slowly.

Scanned with CamScanner

② Example:

consider given loop.

```
for (i=0 ; i<n ; i++)
    for (j=0 ; j<i ; j++)
        temp;
```

★ We mostly interested in time complexity of this main term.

★ In these kind of algorithms, you should ignore the loops and only consider the deepest statement i-e temp (in given case)

★ For finding time complexity:

CHEKING:

i	j	no. of executions (temp)
0	0 (x)	(0) ✗ (does not run)
1	0	1
	1 (x)	
2	0	
	1	2
	2 (x)	
3	0	
⋮	1	3
	2	
	3 (x)	
⋮	⋮	⋮
n	0,1,...n	n

$$\boxed{= 0+1+2+3 \cdots n}$$

★ Now, we will only consider inner instruction:

Scanned with CamScanner

$t(n) = 1 + 2 + 3 + \ldots + n$ (Sum of Arithmetic Series)

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

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

* The growth rate of this algorithm is "quadratic."

NOTE:

In time complexity polynomial, we are only concerned with the highest degree of polynomial.

3) **Example:**

consider loop.

(1) for(i = 1 ; i < n ; i = i * 2) → $\log_2 n + 1$

temp; → $\log_2 n$

comparing it with another loop (with linear growth)

(2) for (i = 0 ; i < n ; i++) → $n + 1$

stmt; → n

* The above (1) written algorithm will be more efficient because it will take lesser steps to execute the input.

Scanned with CamScanner

★ **for example:** if we are given input 16, **loop no. (1)** will take only 5 steps to exeute it, while **loop no. (2)** will take 16 steps.

CHECK:

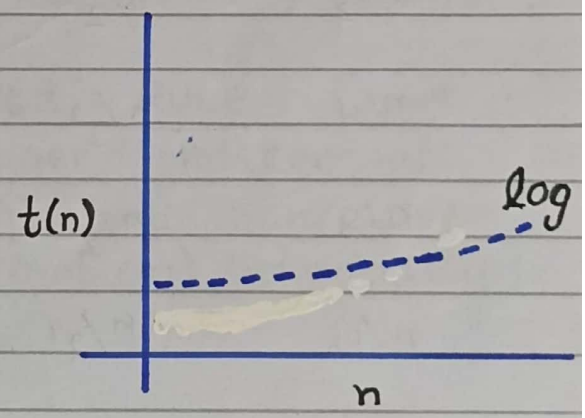
★ Now, To calculate time complexity:

```
for(i=1; i<n; i=i*2)   → log₂n+1
    temp;              → log₂n
```

i		
1	2^0	0
2	2^1	2
4	2^2	2×2
8	2^3	2×2^2
16	2^4	2×2^3
32	2^5	2×2^4
64	2^6	2×2^5
⋮	⋮	⋮
n	2^k	

★ if input is "32", this loop will execute completely in '6' steps.

GRAPH:



We will get:

$$2^k = n$$

$$\log_2 n = k$$

$$\boxed{t(n) = \log_2 n}$$

★ The growth rate of this algorithm is "LOGARITHMIC".

Scanned with CamScanner

④ **Example:**

consider the loop

$$\text{for}\left(i = n \;;\; i > 1 \;;\; i = \frac{i}{2}\right)$$

temp;

★ In this case, steps will be opposite of previous example.

★ To calculate time complexity:

CHECK:

i	
$n/1$	$n/2^0$
$n/2$	$n/2^1$
$n/4$	$n/2^2$
$n/8$	$n/2^3$
$n/16$	$n/2^4$
⋮	⋮
$n/n \boxed{= 1}$	$n/2^k$

we get:

$$1 = n/2^k$$

$$n = 2^k$$

$$\boxed{t(n) = \log_2 n}$$

★ The growth rate of this algorithm is "LOGARITHMIC".

Scanned with CamScanner

NOTE:

RULE OF THUMB:

* if problem is increasing by multiplying

OR

* if problem is decreasing by dividing

then the growth rate of that problem is "LOGARITHMIC".

NOTE:

* Logarithmic techniques/algorithms depends on

"DIVIDE AND CONQUER" RULE

(means to divide a bigger problem into smaller steps (problem) and then solve those smaller steps (problem) individually.)

⑤ **Example:**

Consider a loop.

```
for (i=1; i<n ; i = i*5)
    temp;
```

$\rightarrow \log_5 n + 1$

$\rightarrow \log_5 n$

$$\boxed{t(n) = 2\log_2 n + 1}$$

Scanned with CamScanner

⑥ Example:

Consider a loop.

for $(i = n;\ i > 1;\ i = i/7)$ $\rightarrow \log_7 n + 1$

temp; $\rightarrow \log_7 n$

$$\boxed{t(n) = 2\log_7(n) + 1}$$

Scanned with CamScanner

⑦ PRACTICE EXAMPLE:

consider a loop.

```
for (i=0; i<n; i++)          → n+1
    for(j=0; j<i; j++)       → n(n+1)/2
        for (K=0; K<j; K++)
            temp;
```

CHECK "FOR K LOOP":

i	j	K	temp(no. of execution)
0	0	X	X
1	0,1	0	X
2	0,1,2	0,1	1
3	0,1,2,3	0,1,2	3
4	0,1,2,3,4	0,1,2,3	6
5	0,1,2,3,4,5	0,1,2,3,4	10
⋮	⋮	⋮	⋮
n	0,1,...n	0,1,...(n-1)	n

$$\boxed{t(n) = (n-1) + d}$$

→ This is series of "Arithmetic Progression"

Scanned with CamScanner

RECURSION:

★ In recursion, we check/calculate complexities for two things:

1 → Base case complexity (where if condition is true)

2 → recursion function calling (where if condition is false)

① Example:

consider recursion algorithm.

```
void print (int n)
{
    if (n<=0)          → 1
        cout << n;     → 1
    else
        print(n-1);    → 1 + t_print(n-1)
}
```

CHECK:

1. if (n=0):

$t_{print}(0) = 2$

★ This function will stop calling itself when the statement of "if" will be true.

Scanned with CamScanner

2. else $(n>0)$ this step is to check condition of "if" (whether it is false)

$$t_{print}(n) = \overbrace{1}^{\text{this step is to check condition of "if" (whether it is false)}} + \underbrace{1}_{\text{This 1 is for return value (temp) of function calling.}} + \underbrace{t_{print}(n-1)}_{\text{Time of recursive function calling.}}$$

★ This statement is known as "RECURRENCE RELATION" (a statement in which there is a recursive function calling.)

RECURRENCE RELATION:

$$t_{print}(n) = \begin{cases} 2 & ; \; n<=0 \quad \text{(if statement)} \\ 1+1+t_{print}(n-1) \\ \quad \text{OR} & ; \; n>0 \quad \text{(else statement)} \\ 2+t_{print}(n-1) \end{cases}$$

★ Now we have to do subsitution of this. (RECURRENCE RELATION)

Scanned with CamScanner

SUBSITUTION: (We are only concerned with else statement complexity)

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

$$= 2 + (2 + t((n-1)-1)) \quad \therefore \text{for } n-1: \; n-1-1 = n-2$$

$$= 2 + [2 + t(n-2)]$$

$$= 2+2+(2+t((n-1)-2)) \quad \therefore \text{for } n-2: \; n-1-2 = n-3$$

$$= 2+2+2+t(n-3)$$

$$= 2+2+2+2+t(n-4) \quad \therefore n-1-3 = n-4$$

$$\vdots$$

$$= 2n + t(n-n)$$

$$= 2n + t(0) \quad (\therefore t(0) = 2)$$

$$\boxed{t_{print}(n) = 2n+2}$$

→ The growth rate of this algorithm is 'Linear'.

Scanned with CamScanner

② Example:

consider a recursive algorithm.

```
int func (int n) {
    if (n== 0)                    → 1
    {
        cout << n;                → 1
        return 0;                 → 1
    }
    return func(n-1)+n;           → 1 + t_func(n-1)
}
```

not compulsory to write its time complexity

not compulsory to write its time complexity

CHECK:

1. if (n==0)

$t_{func}(0) = 3$

2. (n != 0)

this step is to check condition of "if" (whether it is false)

$t_{func}(n) = 1 + 1 + t_{func}(n-1)$

★ This statement is known as "RECURRENCE RELATION" (a statement in which there is a recursive function calling)

Scanned with CamScanner

RECURRENCE RELATION:

$$t_{func}(n) = \begin{cases} 3 \; ; & n = 0 \\ \underbrace{1 + t_{func}(n-1) \; ; \; n > 0}_{\text{RECURRENCE RELATION}} \end{cases}$$

★ Now we have to do subsitution of this.

SUBSTITUTION: (Taking time complexity when "if" will be false (means function calling time complexity))

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

$$= 1 + (\overbrace{1 + t((n-1)}-1))$$

$$= 1 + 1 + t(n-2) \qquad (\therefore n = n-1)$$

$$= 1 + 1 (\overbrace{1 + t((n-1)}-2))$$

$$= 1 + 1 + 1 + t(n-3)$$

$$\vdots$$

$$= 1n + t(n-k)$$

$$= 1n + t(n-n) \qquad (\therefore k = n)$$

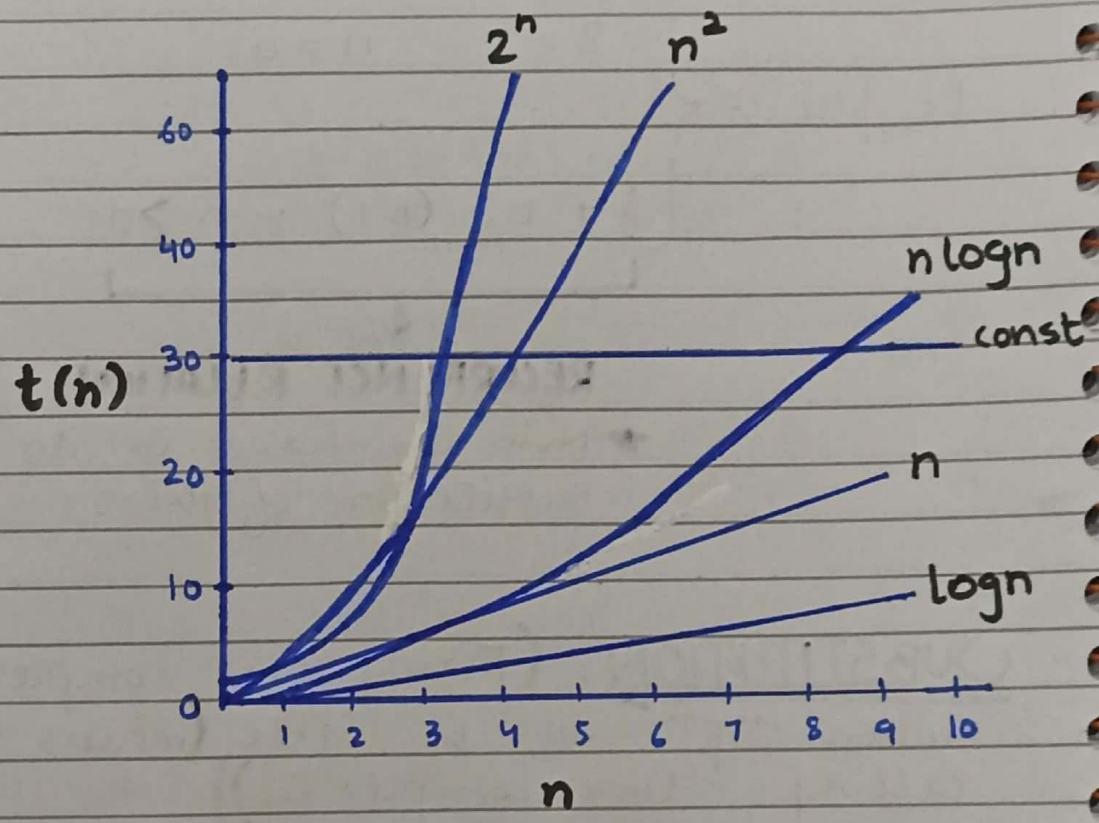
$$= 1n + t(0)$$

$$= n + 3 \qquad (\therefore t(0) = 3)$$

$$\boxed{t(n) = 3 + n}$$ → The growth rate of this algorithm is "linear".

Scanned with CamScanner

Performance Analysis of Algorithm
Comparison of Time Complexities



log n	n	n logn	n^2	n^3	2^n
0	1	0	1	1	2
1	2	2	4	8	4
2	4	8	16	64	16
3	8	24	64	512	256
4	16	64	256	4096	65,536
5	32	160	1024	32,768	4,294,967,296

Scanned with CamScanner

PERFORMANCE MEASUREMENT

★ Now, For time taken on machine.

- 1st thing we have to know is "**COMPUTER SPEED**".

COMPUTER SPEED = NO. OF $\underbrace{\text{STEPS}}_{\text{Instructions}}$ / SEC

★ The execution time of an algorithm running on a particular machine based on input can be calculated by following formula.

FORMULA: (For time taken by an algorithm on particular machine)

$$\text{time (taken by an algorithm on particular machine)} = \frac{\text{total step count of an algorithm}}{\text{speed of a machine (steps/sec)}}$$

NOTE:

★ "PERFORMANCE MEASUREMENT" is dependant of machine.

★ "PERFORMANCE ANALYSIS" is independent of machine.

Scanned with CamScanner

① **EXAMPLE:** (Consider an example for measuring the performance of an algorithm on specific machine)

Total no. of steps: $t(n) = 2n+3$

Computer speed $= 10$ Instruction/sec

FORMULA:

$$\text{time taken} = \frac{\text{total step count of an algorithm}}{\text{computer speed (steps/sec)}}$$

SOLUTION:

$\boxed{\text{AT INPUT } n=10}$

$$t(10) = \frac{2(10)+3}{10}$$

$$= \frac{23}{10}$$

$$t(10) = 2.3 \text{ second.}$$

$\boxed{\text{AT INPUT } n=50}$

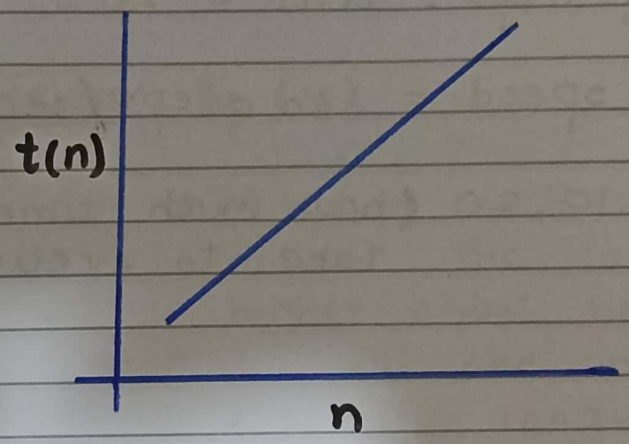
$$t(50) = \frac{2(50)+3}{10}$$

$$= \frac{103}{10}$$

$$t(50) = 10.3 \text{ second.}$$

Scanned with CamScanner

★ We can see that when we increase the size of input, the time taken by this algorithm on specific machine also increasing.



Scanned with CamScanner

② EXAMPLE:

Calculate the total time taken (performance measurement) of an algorithm whose

total no. of steps : $t(n) = 2n^2$

computer speed = 10 steps / sec

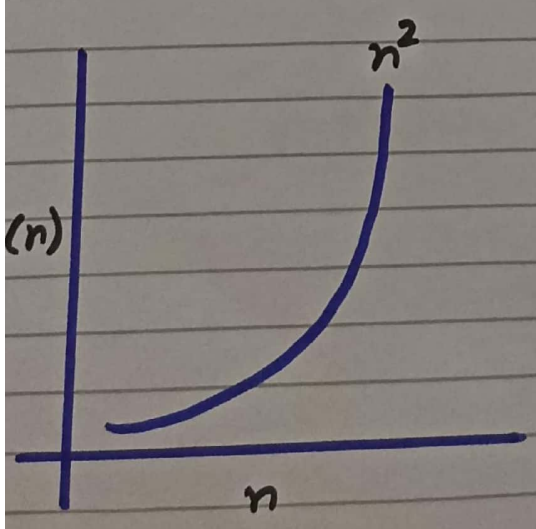
at $n = 5, 10, 20$ (how much time it will take to execute)

SOLUTION:

$$t(5) = \frac{2(5)^2}{10} = \frac{50}{10} = 5 \text{ second.}$$

$$t(10) = \frac{2(10)^2}{10} = 2(10) = 20 \text{ second}$$

$$t(20) = \frac{2(20)^2}{10} = \frac{800}{10} = 80 \text{ second.}$$



★ means its time complexity (growth rate) is "QUADRATIC" because whenever we increase its input size, the time taken by this algorithm on specific machine will be increasing rapidly by factor of square (^2)

Scanned with CamScanner

lecture #05

SPACE COMPLEXITY

main memory

★ The amount of "memory" (RAM) taken by an algorithm to execute itself.

DENOTION: $S_p(n)$

★ In space complexity, we didn't consider the language (don't bother about how many bytes it will take on memory for a specific language)

★ Here (in space complexity) we only see / take / use no. of words because int, float, etc. size may change from machine to machine and language to language (so we don't bother about their size, we only consider their size in words; 1, 2, etc.)

★ We only see about no. of variables which we declare by ourselves in our algorithm / program.

Scanned with CamScanner

ASSUMPTION:

★ If we assume that int takes '2' words in our machine.

EXAMPLE#1:

Consider an algorithm to find its space complexity

```
void test ()
{
    int a = 5, b = 6;      → 4

    int s = a+b;           → 2

    cout << 5;
}
```

$\boxed{S_{test}(n) = 6 \text{ words}}$ → The space complexity of this algorithm is **"CONSTANT"**.

Scanned with CamScanner

EXAMPLE#2:

n words

size of array

```
void test (int a[ ] , int n)
{
    for (int i=0; i<n ; i++)
        cout << a[i];
}
```

1 word

1 word

$$\boxed{S_{test}(n) = n+2}$$

→ The growth rate of this algorithm is "LINEAR".

EXAMPLE # 3:

5 words

1 word

```
void test (int a[ ] , int 5)
{
    for (int i=0; i<n ; i++)
        cout << a[i];
}
```

1 word

size of array

$$\boxed{S_{test}(n) = 7 \text{ words}}$$

→ The growth rate of this algorithm is "CONSTANT"

Scanned with CamScanner

NOTE:

* In space complexity, we are independant of language.

* If we have a function (algorithm) in which we have "**int**" as well as "**float**" variables (datatypes), we don't bother about their sizes on machine.
We only consider all separately declared variable are same; whether they are of '**1**' words or '**2**' words, so on.

Scanned with CamScanner

ASYMPTOTIC NOTATIONS

★ These are mathematical notations.

★ These are used to describe the time complexity of an algorithm when the input tends towards a particular value or a limiting value.

★ These notations help us to represent time complexities in simple way instead of writing lengthy functions.

(We use asymtotic notation so that, we don't need to write lengthy functions)

→ Asymtotic Notations generally have '3' Languages:

- Big-Oh O
- Big-omega Ω
- Big-theta θ

★ On machine, we calculate the time by

End time − start time

NOTE:

★ We only find growth rate / time of processing, not the time of input or output.

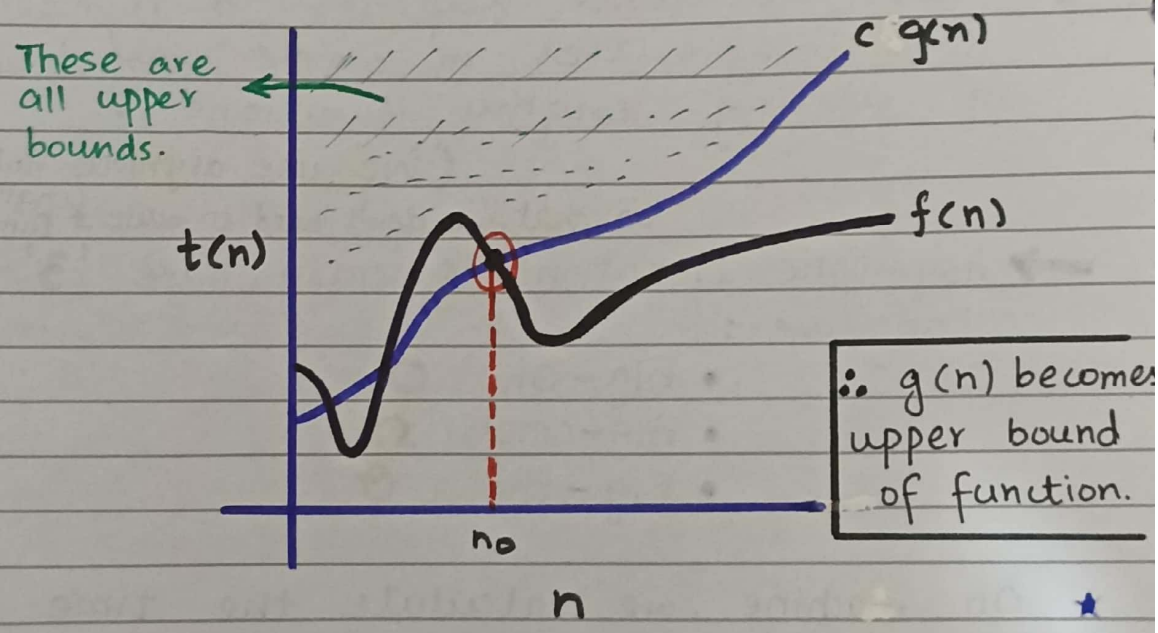
Scanned with CamScanner

① BIG-OH (O)

Denotion

This represent the upper bound of a function.

★ The function $f(n) = O(g(n))$ iff there exist some positive constant (s) 'c' and 'n_0' (both 'c' and 'n_0' positive) where $(n \geq n_0)$ such that $f(n) \leq c\,g(n)$.



∴ g(n) becomes upper bound of function.

Scanned with CamScanner

EXAMPLE: UPPER BOUND BIG-OH

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

→ Let's choose some function $c\,g(n)$ which is greater than from $f(n)$

$$c\,g(n) = 2n$$

n	$f(n) = 2n+3$	$cg(n) = 2n$	$f(n) \leq c\,g(n)$
1	5	2	NO (x)
2	7	4	NO (x)
3	9	6	NO (x)
4	11	8	NO (x)
5	13	10	NO (x)

★ The function '2n' ($c\,g(n)$) is not upper bound of $f(n)$ because it is not greater than or equal to ($\geq$) $f(n)$.

→ Let's choose some function $c\,g(n)$

$$c\,g(n) = 3n$$

n	$f(n) = 2n+3$	$cg(n) = 3n$	$f(n) \leq c\,g(n)$
1	5	3	NO (x)
2	7	6	NO (x)
3	9	9	YES (✓)
4	11	12	YES (✓)
5	13	15	YES (✓)

★ $2n+3 = O(n)$ with $c = 3$ and $n_0 = 3$

Scanned with CamScanner

→ Let's choose $f(n) = 2n+3$ and

$$c\,g(n) = 7n$$

n	$f(n)=2n+3$	$c\,g(n)=7n$	$f(n) \leq c\,g(n)$
1	5	7	✓
2	7	14	✓
3	9	21	✓
4	11	28	✓
⋮	⋮	⋮	✓
100	203	700	✓

★ $2n+3 = O(n)$ with $c = 7$ and $n_0 = 1$

→ If we choose $f(n) = 2n+3$ and

$$c\,g(n) = n^2$$

n	$f(n)=2n+3$	$c\,g(n)=n^2$	$f(n) \leq c\,g(n)$
1	5	1	✗
2	7	4	✗
3	9	9	✓
4	11	16	✓
5	13	25	✓

★ $2n+3 = O(n^2)$ with $c = 1$ and $n_0 = 3$

Scanned with CamScanner

→ If we choose $f(n) = 2n+3$ and

$$c\,g(n) = 2^n$$

n	$f(n) = 2n+3$	$c\,g(n) = 2^n$	$f(n) \leq c\,g(n)$
1	5	2	×
2	7	4	×
3	9	8	×
⋮	⋮	⋮	⋮
100	203	1.267650e+30	✓

★ $2n+3 = O(2^n)$ with $c=1$ and $n_0 = 4$

∴ Here '2^n' is a whole function. We don't consider it as '$c = 2$' and power 'n' as a function.

EXAMPLE:

$$f(n) = 3n\lg n + 2n + 4$$

$$c\,g(n) = 3n\lg n + 2n\lg n + 4n\lg n$$
$$c\,g(n) = 9n\lg n$$

$3n\lg n + 2n + 4 = O(n\lg n)$ at $c = 9,\ n_0 = 1$

Scanned with CamScanner

⇒ SIMPLEST WAY / EASIEST WAY TO FIND $cg(n)$ AS UPPER BOUND OF $f(n)$:

* As we can see that any functions like **$f(n)$** would have many upper bounds which will be called **$cg(n)$**
(Like we saw in previous examples; $2n, 3n, 7n, 2^n, n^2$, etc, are all upper bounds of **$f(n) = 2n+3$**)

* So, here we have a question that which upper bound function **($cg(n)$)** should be chosen.

* To choose the **$c\ g(n)$**:
we should put the highest degree polynomial with all the variables.

* So, easiest way of taking **$cg(n)$**:
For example:

$f(n) = 2n+3$ and its $cg(n) = 2n+3n = \mathbf{5n}$

$cg(n) = 5n$

* So,

$5n$ but $<5n$ means
$6n$ $4n, 3n, 2n, \ldots$
$7n$ are not upper bounds
$\vdots$ of $f(n)$
Kn are all upper bounds of $f(n)$

Scanned with CamScanner

① EXAMPLE:

If we choose $f(n) = 2n+3$ and

$$c\,g(n) = 2n+3n$$

$$c\,g(n) = 5n$$

n	$f(n) = 2n+3$	$c\,g(n) = 5n$	$f(n) \leq c\,g(n)$
1	5	5	✓
2	7	10	✓
3	9	15	✓
⋮	⋮	⋮	⋮
100	203	500	✓

★ $2n+3 = O(n)$ with $c = 5$ and $n_0 = 1$

② EXAMPLE: If

$$f(n) = 2n^2 + 3n + 1$$

$$\Rightarrow c\,g(n) = 2n^2 + 3n^2 + 1n^2$$

$$c\,g(n) = 6n^2$$

★ $2n^2 + 3n + 1 = O(n^2)$ with $n_0 = 1$, $c = 6$

Scanned with CamScanner

③ **EXAMPLE:** If

$$f(n) = 5n^3 + 2n^2 + 3n + 6$$

$$\hookrightarrow cg(n) = 5n^3 + 2n^3 + 3n^3 + 6n^3$$

$$cg(n) = 16n^3$$

★ $5n^3 + 2n^3 + 3n + 6 = O(n^3)$ with $n_0 = 1$, $c = 16$

★ By this method (of finding $cg(n)$), "n_0" will always be "1".

CHOOSING APPROPRIATE UPPER BOUND ($cg(n)$):

$$\boxed{\underbrace{n < n\lg n < n^2 < n^2\lg n < n^3 < 2^n < n^n}_{\text{all are upper bounds of 'n'.}}}$$

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

★ Which one to choose as upper bound of $f(n)$?

★ Although, they all are correct because all of them are upper bound of $2n+3$.

★ But always choose the meaningful $g(n)$

★ So, here $O(n)$ is the closest to $2n+3$ and meaningful (so, we choose it as Big-Oh of $f(n)$).

Scanned with CamScanner

lecture# 06

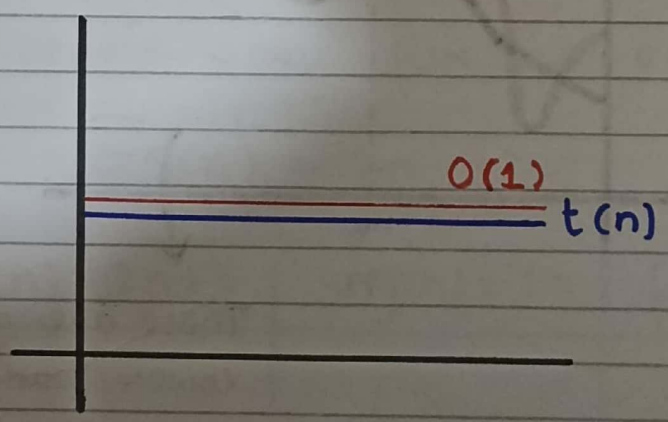
ASSYMTOTIC NOTATIONS REPRESENTING CONSTANT FUNCTION:

★ A constant function is represented asymtotically by digit '1' like 'O(1)', 'Ω(1)', 'Θ(1)'.

★ **For Example:**

$T(n) = 3$; this function has a constant growth.
So, we can simply represent it with **O(1)** instead of writing 2, 3, or any other term.

$T(n) = O(1)$ with $c = 1$, $n_0 = 1$.



★ When we have **O(1)** or **Ω(1)** or **Θ(1)** then we can conclude that the growth rate of that function is **"CONSTANT."**

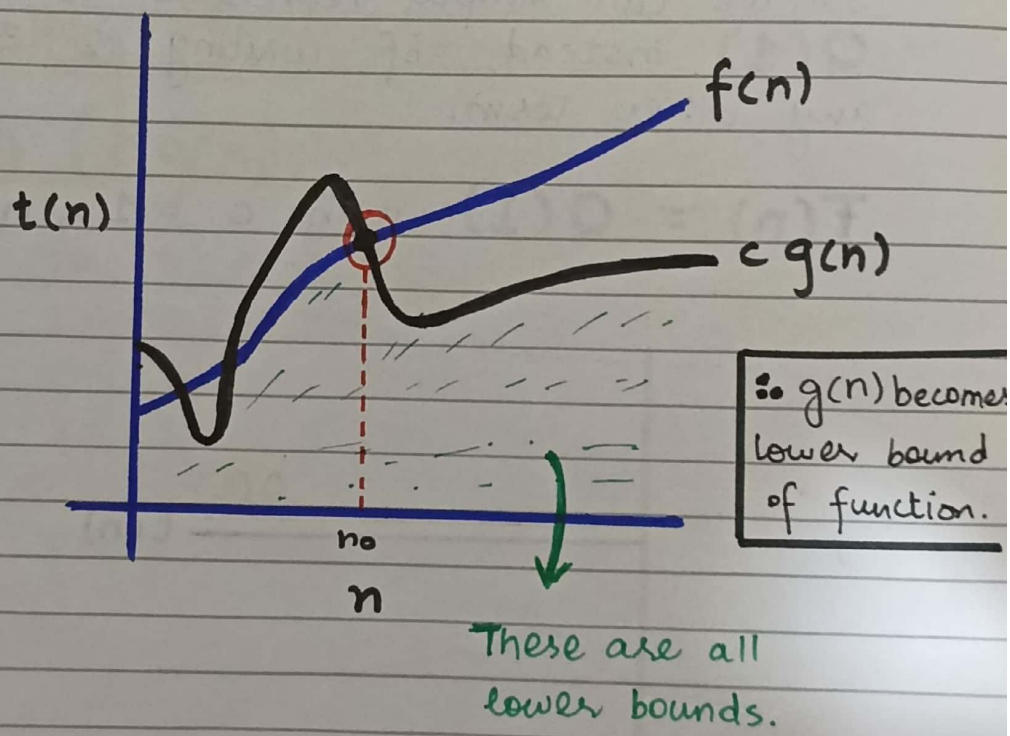
Scanned with CamScanner

② BIG-OMEGA (Ω)

Denotion

This represent the lower bound of a function.

★ The function $f(n) = \Omega(g(n))$ iff there exist some positive constant(s) 'c' and 'n_0' (both 'c' and 'n_0' positive) where $(n \geq n_0)$ such that $f(n) \geq c\,g(n)$



Scanned with CamScanner

EXAMPLE: LOWER BOUND BIG-OMEGA

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

→ Let's choose some function $c\,g(n)$ which is smaller than $f(n)$

$$c\,g(n) = 2n$$

n	$f(n) = 2n+3$	$c\,g(n) = 2n$	$f(n) \geq c\,g(n)$
1	5	2	✓
2	7	4	✓
3	9	6	✓
⋮	⋮	⋮	
100	203	200	✓

★ $2n+3 = \Omega(n)$ with $c = 2$ and $n_0 = 1$.

→ Let's choose some function $c\,g(n)$

$$c\,g(n) = 1n$$

n	$f(n) = 2n+3$	$c\,g(n) = 1n$	$f(n) \geq c\,g(n)$
1	5	1	✓
2	7	2	✓
3	9	3	✓
⋮	⋮	⋮	
100	203	100	✓

★ $2n+3 = \Omega(n)$ with $c = 1$ and $n_0 = 1$.

Scanned with CamScanner

→ Let's choose some function $c\,g(n)$

$$c\,g(n) = 4$$

n	$f(n) = 2n+3$	$c\,g(n) = 4$	$f(n) \geq c\,g(n)$
1	5	4	✓
2	7	4	✓
3	9	4	✓
⋮	⋮	⋮	
100	203	4	✓

★ $2n+3 = \underbrace{\Omega(1)}$ with $c = 1$ and $n_0 = 1$

because it is a constant function.

→ Let's choose some function $c\,g(n)$

$$c\,g(n) = 2\log n$$

n	$f(n) = 2n+3$	$c\,g(n) = 2\log n$	$f(n) \geq c\,g(n)$
1	5	0	✓
2	7	0.60	✓
3	9	0.95	✓
⋮	⋮	⋮	
100	203	4	✓

★ $2n+3 = \Omega(\log n)$ with $c = 2$ and $n_0 = 1$.

Scanned with CamScanner

EXAMPLE ②:

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

→ Let's choose some function $c\,g(n)$

$$c\,g(n) = \sqrt{n}$$

n	$f(n)=2n+3$	$cg(n)=\sqrt{n}$	$f(n) \geq cg(n)$
1	5	1	✓
2	7	1.41	✓
3	9	1.73	✓
⋮	⋮	⋮	
100	203	10	✓

★ $2n+3 = \Omega(\sqrt{n})$ with $c = 1$ and $n_0 = 1$.

∴ Here '$\sqrt{n}$' is a whole function. And the constant multiply with it is '1'.

Scanned with CamScanner

⇒ SIMPLEST WAY / EASIEST WAY TO FIND $cg(n)$ AS LOWER BOUND OF $f(n)$:

★ As we can see that any functions like $f(n)$ would have many lower bounds which will be called $cg(n)$ (Like we saw in previous examples; $1n$, $2n$, 4, $2\log n$, $\sqrt{n}$, etc are all lower bounds of $f(n) = 2n+3$)

★ So, here is a question that which lower bound function ($cg(n)$) should be chosen.

★ To choose the $cg(n)$:
We take highest degree polynomial of $f(n)$ with its constant and neglect all other variable or constant, etc.

Like:
for $f(n) = 2n+3$, we take
$cg(n) = 2n$

Scanned with CamScanner

① **EXAMPLE:** If

$$f(n) = 2n^2 + 3n + 4$$
$$\Rightarrow c\,g(n) = 2n^2$$

* $2n^2 + 3n + 4 = \Omega(n^2)$ is meaningful one.

② **EXAMPLE:** If

$$f(n) = 3n\lg n + 2n + 4$$
$$\Rightarrow c\,g(n) = 3n\lg n$$

* $3n\lg n + 2n + 4 = \Omega(n\lg n)$ is meaningful one

* By this method (of finding $c\,g(n)$); "n_0" will always be '1'.

CHOOSING APPROPRIATE LOWER BOUND ($c\,g(n)$):

$$\underbrace{1 < \lg(n) < \sqrt{n}}_{\text{all are lower bounds of 'n'}} < n$$

$$2n+3 = \Omega(n)$$
$$2n+3 = \Omega(\log n)$$
$$2n+3 = \Omega(\sqrt{n})$$
$$2n+3 = \Omega(1)$$

* Which one to choose as lower bound of $f(n)$?
* Although, they are all correct because all of them are lower bound of $2n+3$.

Scanned with CamScanner

* But always choose the meaningful g(n).

* So, here $\Omega(n)$ is the closest to $2n+3$ and meaningful (so, we choose it as Big-Omega of f(n)).

NOTE:

* In asymptotic notations, there is no need to write base of **log** and **ln**.

IMP:

* Whenever base is not mentioned, we assume:
  ln to be **base 2** (ln = base 2)
  and **log** to be **base 10** (log = base 10)

* Every algorithm must have some upper bound and lower bound.

* Upper bound and lower bound can be mostly same because there is also equality sign in their definition.

Scanned with CamScanner

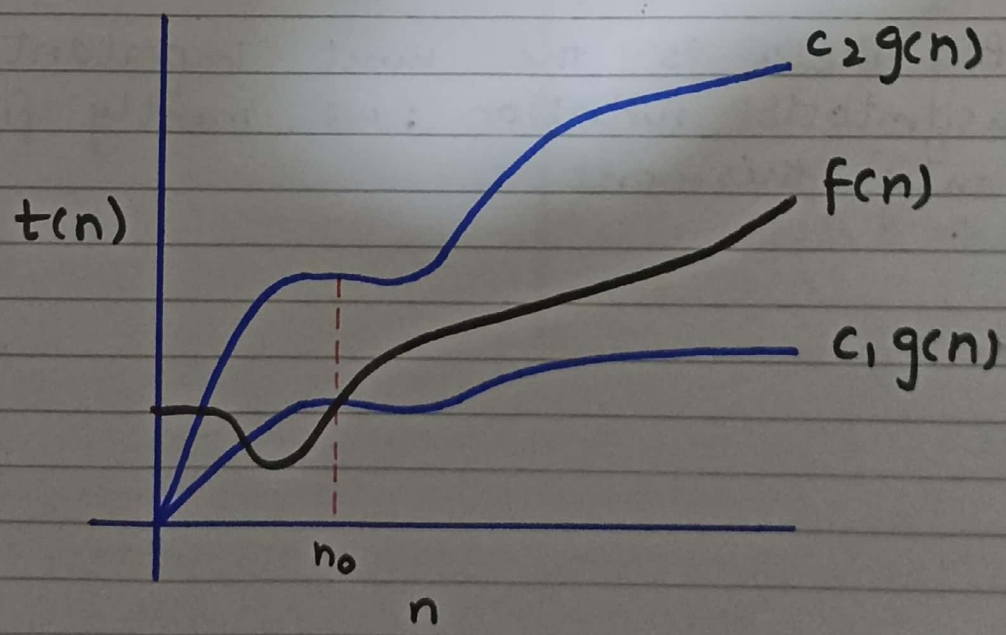
③ BIG-THETA (Θ)

Denotion

* This represent the average bound of a function.
* It is also known as 'tight bound'.

→ The function $f(n) = \Theta(g(n))$ iff there exist some constant 'c_1', 'c_2' and 'n_0' ('c_1', 'c_2' and 'n_0' positive) where $(n \geq n_0)$ such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$

$$\underbrace{c_1 g(n) \leq f(n)}_{\text{Lower Bound}} \underbrace{\leq c_2 g(n)}_{\text{upper Bound}}$$



* Big Theta covers both upper and lower Bound

Scanned with CamScanner

NOTE:

* Here, in Big Theta, '$g(n)$' is exactly same in lower as well as in upper bounds; but 'c_1' and 'c_2'
  - can also be same $(c_1 = c_2)$
  - or may not be same $(c_1 \neq c_2)$

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

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

here $(c_1 = c_2)$

* Big Theta only exist for that function whose upper and lower bound are same.

* For most of the functions, **big theta** (Θ) does not exist (because for that function upper and lower bounds are not same).

* Big Theta is the most important asymtotic notation; we mostly find only this one.

Scanned with CamScanner

EXAMPLE 1:

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

$$c_1\, g(n) = 2n$$
$$c_2\, g(n) = 5n$$

$$2n \leq 2n+3 \leq 5n \quad (\therefore \text{here } c_1 \neq c_2)$$

n	$f(n)=2n+3$	$c_1 g(n)=2n$	$c_2 g(n)=5n$
1	5	2	5
2	7	4	10
3	9	6	15
⋮	⋮	⋮	⋮
100	203	200	500

* $2n+3 = \Theta(n)$ with $c_1 = 2$, $c_2 = 5$ and $n_0 = 1$

$\therefore$ * constant 'c_1' is with lower bound of $g(n)$
* constant 'c_2' is with upper bound of $g(n)$.

Scanned with CamScanner

EXAMPLE ②: If (we take '$c_1 g(n)$' and '$c_2 g(n)$' which are not same)

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

$$c_2 g(n) = 1n^2$$

★ $2n+3 = O(n^2)$ with $c=1$, $n_0=1$

AND

$$c_1 g(n) = 1n$$

★ $2n+3 = \Omega(n)$ with $c=1$, $n_0=1$

- It does not have big theta because $g(n)$ for upper bound and lower bound is not same.

- This $f(n)$ only have **Big-Oh** or **Big-Omega**. It does not have **Big-Theta**, because

$$c_1 g(n) \neq c_2 g(n)$$

$$1n^2 \neq 1n$$

not same.

Scanned with CamScanner

NOTE:

* We cannot find Big Theta (θ) of '$\sqrt{n}$' because its upper and lower bound are always different
(means $c_1 g(n) \neq c_2 g(n)$
OR
big-Oh $\neq$ big-Omega)

MCQ

* Big Theta does not exists for every / all the algorithms (functions).
  ↳ if big-theta 'θ' does not exist for a function, they will go (find) for Big-Oh 'O' or Big-Omega 'Ω'.

Scanned with CamScanner