Data Structures and Algorithms

Reg #01

Before Mids

[redacted]

4th Sem

RAYYAN SHABBIR

BITF20M535

(Lecture#1)

DATA STRUCTURE AND ALGORITHM:

DATA:

* Data is defined as Raw facts and figures. (Material)
* It can also be refered 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.

STRUCTURE:

* Foum 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.

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.

* Sequence-wise procesing 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.

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

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:

$t_{program}(n)$ → input (size)

NOTE:

* We assume:

```
int a = 5;
```

mostly it takes '2' steps to execute.

→ But we mostly consider Counts / Measur 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 s

This will be called step per execution

* **FREQUENCY: (f)**
  - How many times program / spec step repeats.

* **TIME COMPLEXITY: (TOTAL)**
  - If we multiply 'S/e' and then we get time comp

(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
1-	void test() { → 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-	} → Brackets 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'.

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

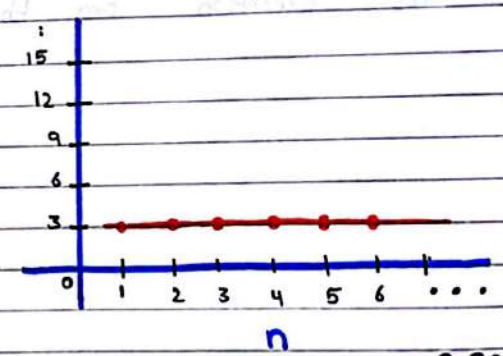
STATEMENT NO..	s/e	f	s/e x f = TOTAL
1-	0	1	0
2-	1	1	1
3-	1	1	1
4-	1	1	1
5-	0	1	0

$t_{test}(n) = 3$

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().



$t_{test}(n) = 3$

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

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)	6
temp;	1	5 (False time)	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	✓	X

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

③ **EXAMPLE:** (Lower limit, upper limit)

```
for(i=0; i<7; i++)
    temp;
for(j=0; j<5; j++)
    start;
```

	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
		$t(n) = 26$	

↳ GROWTH RATE OF THIS FUNCTION IS ALSO "CONST

NOTE:

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

FORMULA : ⇒ U.L − L.L + 1

$7 - 0 + 1 = 8$

OR

$5 - 0 + 1 = 6$

GENERALIZED TIME COMPLEXITY OF LOOP:

④ EXAMPLE: Consider a general loop.

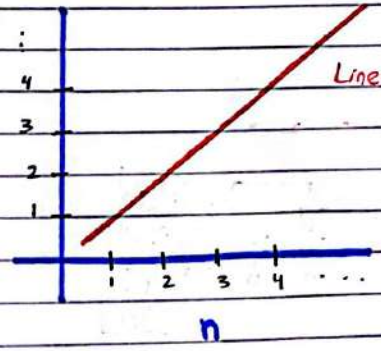
```
for(i=0; i<n; i++)   → n+1
    temp;            → n
                     t(n) = 2n+1
```

HOW → U.L − L.L + 1 → n − 0 + 1 = (n+1)

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 the value of input increases.



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
                     t(n) = 4n+2
```

t(n) = 4n+2

NOTE:

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

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

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

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

		s/e	f	TOTAL (s/e x 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.

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

↳ It is not

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)
```

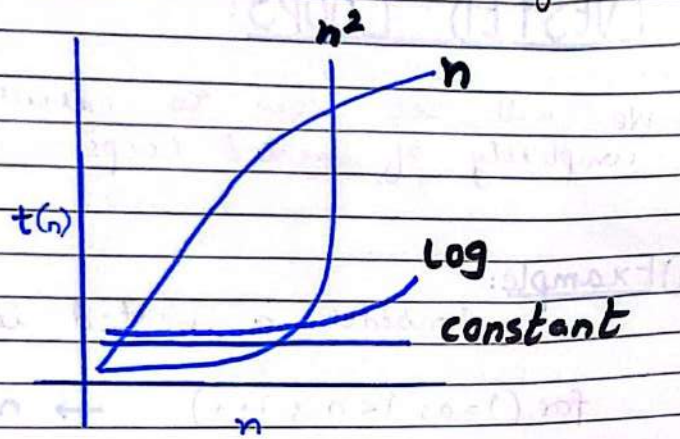
$t(n) = 2n^2 + 2n + 1$

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

* The growth rate of this algorithm is "quadratic"

CHECK:

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



NOTE:

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

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	(0) ✗ (does not run)
	1	(1) ✓
1	0	(1) ✓
	1	(0) ✗
2	0	(1) ✓
	1	(1) ✓
	2	(0) ✗
3	0	(1) ✓
	1	(1) ✓
	2	(1) ✓
	3	(0) ✗
n	0	(1) ✓
	1	(1) ✓
	2	(1) ✓
	3	(1) ✓
	...	...
	n-1	(1) ✓
	n	(0) ✗

④ **Example:**

consider the loop.

```
for (i=n ; i>1 ; i = i/2)
    temp;
```

* In this case i steps will be opposite of previous example.

* To calculate time complexity:

CHECK:

i

$n/1$ $n/2^0$

$n/2$ $n/2^1$

$n/4$ $n/2^2$

$n/8$ $n/2^3$

$n/16$ $n/2^4$

⋮ ⋮

$n/n = 1$ $n/2^k$

we get:

$1 = n/2^k$

$n = 2^k$

$t(n) = \log_2 n$

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

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)   → log5 n + 1
    temp;                   → log5 n
```

$t(n) = 2 \log_5 n + 1$

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.

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

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

This 1 is for return value (temp) of function calling.

Time of recursive function calling.

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

RECURRENCE RELATION:

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

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

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

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

$= 2 + (2 + t((n-1)-1))$ ∴ for n-1: n-1-1 = n-2

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

$= 2+2+(2+t((n-1)-2))$ ∴ for n-2: n-1-2 = n-3

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

$= 2+2+2+2+t(n-4)$ ∴ n-1-3 = n-4

:

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

$= 2n + t(0)$ (∴ t(0) = 2)

$t_{print}(n) = 2n+2$ → The growth rate of this algorithm is 'linear'.

② Example:

consider a recursive algorithm.

```
int func (int n) {
    if (n == 0)                    → 1
    {
        count++;                   → 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)

RECURRENCE RELATION:

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

RECURRENCE RELATION

★ Now we have to do substitution of this.

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

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

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

$$= 1 + 1 + t(n-2) \quad (\because n = n-1)$$

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

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

$$\vdots$$

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

$$= 1n + t(n-n) \quad (\because k = n)$$

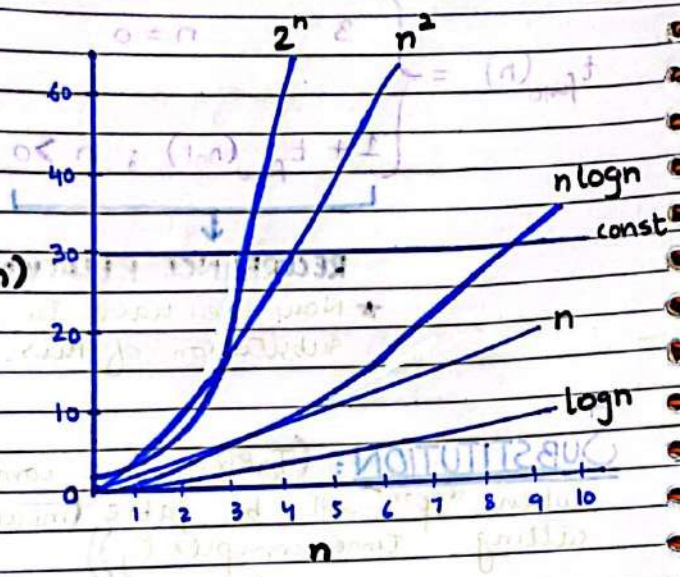
$$= 1n + t(0)$$

$$= n + 3 \quad (\because t(0) = 3)$$

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

Performance Analysis of Algorithm

Comparison of Time Complexities



log n	n	n log n	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

Performance Measurement

★ Now, For time taken on machine.

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

COMPUTER SPEED = No. OF STEPS/SEC
(STEPS: Instructions)

★ 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)

time (taken by an algorithm on particular machine) = total step count of an algorithm / speed of a machine (steps/sec)

NOTE:

★ "PERFORMANCE MEASUREMENT" is dependant of machine.

★ "PERFORMANCE ANALYSIS" is independent of machine.

SPACE COMPLEXITY

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

main memory ↑

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.

ASSUMPTION:

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

EXAMPLE#1:

find its space complexity
Consider an algorithm to

```
void test()
{
    int a = 5, b = 6;   → 4
    int s = a + b;      → 2
    cout << 5;
}
```

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

EXAMPLE#2:

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

- int a[] → n words
- int n → size of array → 1 word
- int i = 0 → 1 word

$S_{test}(n) = n+2$ → The growth rate of this algorithm is **"LINEAR"**.

EXAMPLE # 3:

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

- int a[] → 5 words
- int 5 → 1 word → size of array
- int i = 0 → 1 word

$S_{test}(n) = 7$ words → The growth rate of this algorithm is **"CONSTANT"**

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.

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 notatio so that, we don't need to write lengthy function

➡ 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:

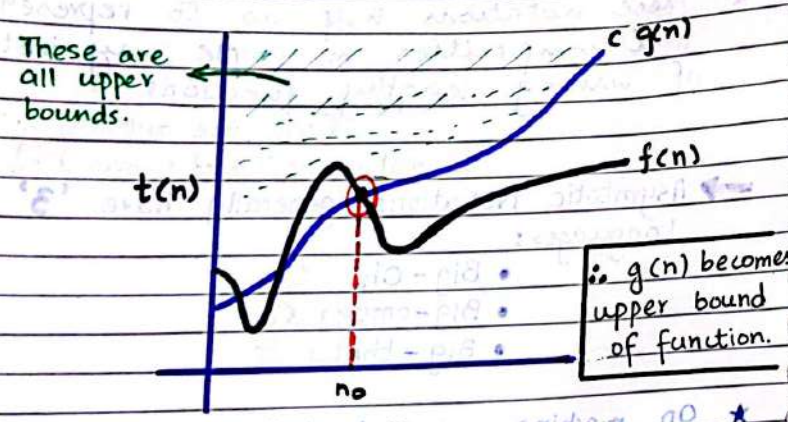
* We only find growth rate / time of processing, not the time of input or output.

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 $f(n) \leq cg(n)$.



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) ≤ cg(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' (cg(n)) is not upper bound of f(n) because it is not greater than or equal to (≥) f(n).

→ Let's choose some function c g(n)

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

n	f(n) = 2n+3	cg(n) = 3n	f(n) ≤ cg(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 $n_0 = 3$

→ Let's choose $f(n) = 2n+3$ and $cg(n) = 7n$

n	$f(n)=2n+3$	$cg(n)=7n$	$f(n) \leq cg(n)$
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 $cg(n) = n^2$

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

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

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

n	$f(n)=2n+3$	$cg(n)=2^n$	$f(n) \leq cg(n)$
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:

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

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

$cg(n) = 9n\lg n$

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

⇨ 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 $cg(n) = 2n+3n = 5n$
$cg(n) = 5n$

* So,

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

① EXAMPLE:

If we choose $f(n) = 2n+3$ and
$cg(n) = 2n+3n$
$cg(n) = 5n$

n	$f(n) = 2n+3$	$cg(n) = 5n$	$f(n) \leq cg(n)$
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

$f(n) = 2n^2 + 3n + 1$
$\Rightarrow cg(n) = 2n^2 + 3n^2 + 1n^2$
$cg(n) = 6n^2$

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

③ EXAMPLE: If

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

↳ $cg(n) = 5n^3 + 2n^3 + 3n^3 + 6n^3$

$cg(n) = 16n^3$

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

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

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

$n < n\lg n < n^2 < n^2\lg n < n^3 < 2^n < n^n$

↳ 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 $f(n)$).

Lecture #05

ASSYMTOTIC NOTATIONS REPRESENTING CONSTANT FUNCTION:

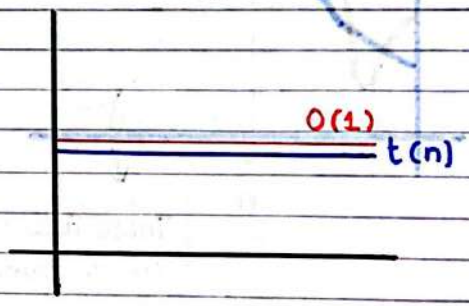
★ A constant function is represented asymtotically by digit '1' like '$O(1)$', '$\Omega(1)$', '$\Theta(1)$'.

★ 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 $\Omega(1)$ or $\Theta(1)$ then we can conclude that the growth rate of that function is "CONSTANT."

⇒ 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
$cg(n) = 2n$

① EXAMPLE: If

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

② EXAMPLE: If

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

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

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

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

$1 < \lg(n) < \sqrt{n} < n$ *
↳ all are lower bound of 'n' *

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

* 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 Ω because there is also equality (sign) in their definition.

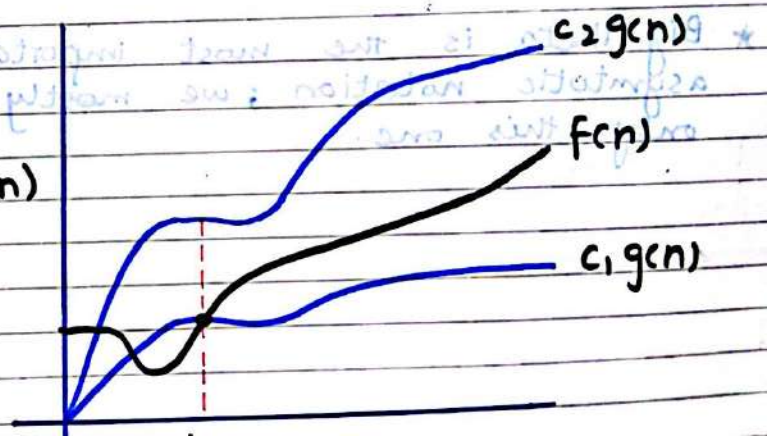
③ BIG-THETA (Θ)

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, these 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 $c_1 g(n) \leq f(n) \leq c_2 g(n)$

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



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.

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 ≠ 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'.

Lecture #7

LIST

→ It is a Data Structure.

→ It is ADT (Abstract Data Type)

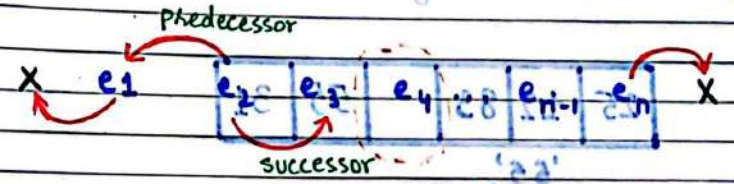
→ A set of values written down one below the other.

→ A number of connected items or names written or printed consecutively, typically one below the other.

→ List is a contagious memory allocation of homogeneous elements.

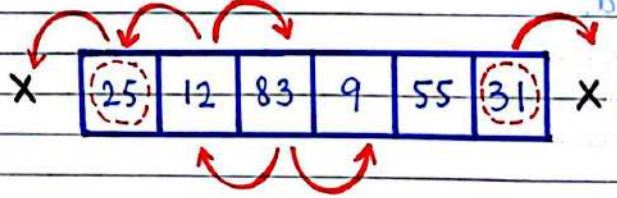
→ TO MAKE A LIST (FOR A LIST)

* The elements of the list must have predecessor and successor relationship.



* The first element of the list has no predecessor therefore it is marked as the beginning of the list.

* The last element of the list has no successor therefore it is marked as the end of the list.



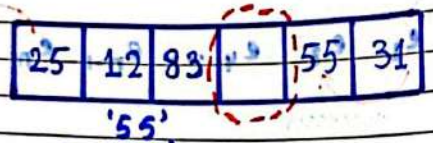
* Element '25' does not have any predecessor therefore indicates the beginning of the list.

* Element '31' does not have any successor therefore indicates the end of the List.

* All other elements have their predecessor and successor to maintain the list property.

→ INVALID LIST

* A list having vacant / empty space in the mid of the elements.



* Element '' does not have any predecessor so it will be considered as the last element which is logically wrong because there are two more elements exist after the empty cell.
* This list will be considered as "invalid".

TYPES OF LIST

(1) Unordered List

(2) Ordered List

(1) UNORDERED LIST :

* Elements are placed in the list in no particular order.
* Unordered list does not have in symentic or logical relation.
* It is only a list.
* Here, only relationship of predecessor and successor holds.
* We mostly maintains unordered list.
* In unordered list, the given list could be sorted as well as unsorted.

(2) ORDERED LIST:

* Elements maintain certain order either numerically or alphabetically.
* Elements are ordered according to some rule (relationship)
* Each element followed by one another.
* In ordered list, there have any relationship according to which data is arranged.
* In ordered list, we are always given sorted list.

NOTE:

* It is not necessary that ordered list will always be sorted and it is also not necessary that unordered list will always be unsorted.

→ UNSORTED LIST

RAYAN
SHABIR
ALI
SHAHMEER

→ SORTED LIST

* Names are sorted alphabetically.

ALI
RAYAN
SHABIR
SHAHMEER

PROPERTIES / TERMINOLOGIES OF A LIST:

→ To maintain a list of elements, following things should be considered

• Max Size

* Every list must have some "Max Size" → total available sort (memory)
  ↳ represent total capacity (which it maximum holds data)

• Current Size

* Every list must also have "Cur Size" → sorts (memory) which we use
  ↳ current number of elements a list holds.

* Initial curSize is '0'.

NOTE:

* We mostly deals with "curSize".

If maxSize is '10' and curSize is '5', then traverse / processing will be done upto curSize (i.e. 5th location).

GENERAL OPERATIONS PERFORMED ON A LIST:

(1) **INSERTION:** Adds an element in the list.

(2) **DISPLAY:** Display the complete data of list

(3) **SEARCH:** Searches an element using the given key

(4) **UPGRADE:** Upgrades an element with new one

(5) **DELETION:** Deletes an element using the given key.

(6) **MERGE:** It merges two lists

(7) **SPLIT:** Dividing list into two parts.

(8) **COMPARISON:** compares a list with another list.

(1) INSERTION IN UNORDERED LIST:

* It inserts / adds an element in the list.

* To perform insertion in the list, we first have to check if either the list is full or not:

curSize == MaxSize?

Is curSize equal to MaxSize?

* If at start curSize = 0 and user wants to insert element '5' then this new element will be inserted at the position **curSize + 1** (i.e. 1st location)

curSize ++

➡ **curSize = 1**

* Then if user again want to insert element '3' then again check **curSize == maxSize**

Then this new element will be inserted at the position **curSize + 1**

cursize ++

➡ **cursize = 2**

* Then again check repeating by same above process.

5	3	4	7	1	2

curSize = 6

* Whenever we insert an element in an unordered list, we insert it at the last (curSize + 1) index, because it is an unordered list, so we don't need to write data in order, we always insert the new element at the end.

1	2	3	4	5	6
25	12	83	9		

➡ The max size of this is '6' and current size is '4'.

➡ A new element '66' is inserted in the list on position **current size + 1** i.e **5** in this case.

1	2	3	4	5	6
25	12	83	9	66	

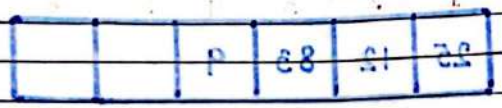
➡ The current size raises to '5'.

➡ Whenever we do insertion, we will increment the curSize by '1'.

➡ Insertion takes constant time to evaluate so, its time complexity is **O(1)**.

(2) DISPLAY IN UNORDERED LIST:

* To display a list, we print data upto the **CurSize** only. We don't print upto maxSize because we want to print data which is in use and we only print those location/addresses which are in use.



So, that's why, we execute our loop upto "less than CurSize" to print the data which is in use.

2) DISPLAY IN UNORDERED LIST:

* Whenever we print data of a list, we print the data upto **curSize**.
* We don't print upto **maxSize** because we want to print the data which is present on the locations we are using.
* So, that is the reason, we execute our loop from **0** to less than curSize (< curSize) that data is in use.

(3) SEARCHING IN UNORDERED LIST:

* Searching of an element in the list simply be done by comparing the required **Key** element **one by one** with the elements of the list.
* Checks each element with a **"Key"** element.
* But here we first have to check "if the list is empty or not"

curSize = 0? (if list is empty)

if curSize > 0 (if list is not empty)

→ If it is return true that List is not empty, then traverse (loop will be evaluated to process data upto curSize)

→ So, here, all elements (i.e **n** elements) are being checked one by one. So, its time complexity will be **O(n)** or **Θ(n)**.

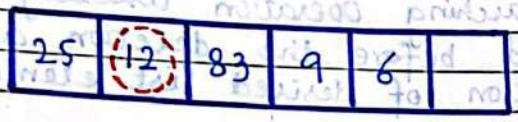
* The Searching operation usually performed before the deletion and/or upgradation of desired list element.

(4) DELETION IN UNORDERED LIST:

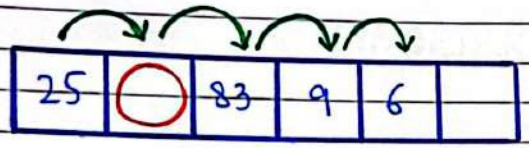
* "Deletion" operation cannot perform without "Search" operation.

* Whenever we want to delete in an unordered list, first we have to search.

If we have a list

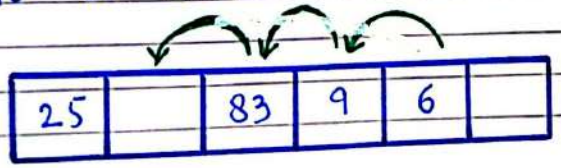


→ And we want to delete '12'.



→ After removal an empty (vacant) space is left, so here rule of predecessor and successor violates. It becomes invalid list.

* → To maintain the property of the list (predecessor and successor), after removing/deleting an element from unordered list and still makes it a list.



* Here, we shift an element towards the empty cell and decrement the curSize by one.

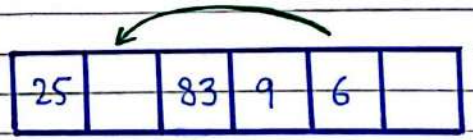
→ But the shifting here is not valid because it requires 'n-1' (or 'n') shiftings.

* Shifting element like this will take O(n)

→ As list in unordered; So, best way to remain it as a list (maintain property of the list)

* To make it efficient, simply replace the deleted element with the last element of list and decrement the **curSize** by one.

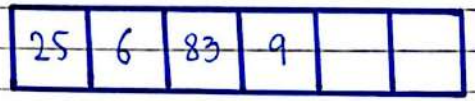
Like:



also

curSize --;

So, list will be:

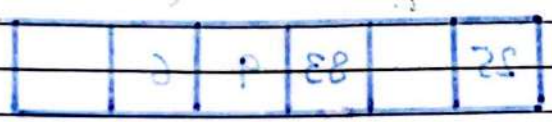


The time complexity for deletion for unordered list is also 'n' → O(n).

* It is because deletion also first search the element, for which it's time complexity will become 'n' (O(n))

* Then, after searching, the element will be deleted and last element will be replaced to empty cell, for this, it's time complexity will be O(1).

* So, orignally the time complexity for deleting will be O(1) and for overall searching will be O(n). The time complexity for deletion operation will be 'n+1' i-e O(n).



(5) UPGRADE IN UNORDERED LIST:

* First check if list is not empty;

  curSize = 0 ?

* "Upgrade" operation also cannot perform without "Search" operation.

* To upgrade an unordered list, first we have to search for the element which we want to update with new element

* But here neither curSize ++ nor curSize --,
  because here value is only updating; neither adding nor decreasing.
  So, time complexity of unordered list is also O(n).

6) MERGE:

* We merge/join two lists.
* The resultant will be a combination of the elements of both lists.
* If we want to merge lists

l_1 = n elements
l_2 = m elements
then resultant list will be:

resultant = n+m elements

7) SPLITTING:

* We divide a list into two parts.

8) COMPARISON:

* We compare two lists with each other.

GENERAL OPERATIONS FOR ORDERED LIST:

* Here list is sorted.

(1) INSERTION IN SORTED LIST:

* Here every element couldn't be placed directly at last position, we first have to check where to place the element according to some relationship.
* If we want to insert

element 1 : e1 = 1
element 2 : e2 = 4

* Here each element will not be placed one after another.
First, we have to check

if $e_1 < e_2$? (CHECK)

$e_1 < e_2 < e_3 < e_4 < e_5 \ldots < e_n$

and then insert.

→ So, after checking

1	4		

→ then if we want to again insert

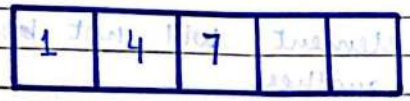
$e_3 = 7$

* Then again, we will compare element. But remember we always compare with last element i.e **cursize.**

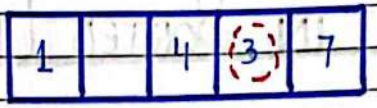
* And if the element (which is being compared) comes according to relationship i.e **"less than cursize"** (< cursize), then we keep comparing backward until we place the element at right position.

* So, here better loop choice is **"while"** because iterations here are not known.

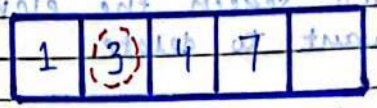
* List became:



→ Then, suppose we want to insert a new element '3' in a sorted list, we have to shift all elements one step forward to place '3' in right place.



(7 < 3)? ✓



* In ordered list (sorted list), we must have to shift elements, to maintain order of list.

* Thus, time complexity of insertion in sorted list is **O(n).**

NOTE:

So, Best way to make a list, first maintain unordered list and then maintain its order by using some sorting algorithm.

(2) SEARCHING IN SORTED LIST:

* In sorted list, we have to traverse each element with one another.

So, its time complexity is also 'n' → **O(n).**

(3) DELETION IN SORTED LIST:

* First, we will search the element which we want to delete.
* Here we cannot place the last element to the deleted position (Like we did in unordered list) because its order will not maintain.
* So, here only way to delete element is by shifting, so that order maintains.

So,
its time complexity will be

$$n + n = 2n$$

Search ← n; → Shift ← n

(4) UPGRADE IN SORTED LIST:

* First, we will have to search the element which we want to upgrade.
* Then, we will check the value with which we want to upgrade the element.
We will check if the element is on its right position or not by comparing / shifting elements
* So, here time complexity will be

$$n + n = 2n$$

Search ← n; → Shift ← n

(5) MERGE:

* To merge two sorted lists, we cannot directly merge them.
* But we have to use some reliable algorithm to perform merge because we have to maintain order of resultant list.

(6) SPLITTING:

* Splitting is easy in a sorted list we will simply divide the list into two parts and each part will be in sorted form.

(7) COMPARISON:

* We compare two sorted lists with each other.

NOTE:

* Generally, it is a good idea to maintain unsorted lists at first instead of maintaining order of elements.
* List can be sorted easily with help of some efficient sorting algorithms later or when needed.
* In DSA, we avoid to use "Continue", "Break" and "GOTO" statements.

Lecture #8

ARRAY:

* "Memory Representation of List is done by Array."
* It is a data structure.
* Array data structure enables us to store a group of data together in one place.
* Array is a finite collection of homogeneous data elements.
* All the elements of array are stored in contigious locations of memory.

ARRAY TERMINOLOGY:

→ **Size:**
The number of elements in an array.

→ **Type:**
The kind of data an array represents e.g. integers, character string, etc.

→ **Base:**
memory address of the first element of array.

→ **Word:**
Denotes the size of an element

→ **Index:**
A subscript integer value used to refer elements of the array.

→ **Range of Indices:**
Indices of the array elements may change from lower bound (LB) to an upper bound (UB), and these bounds are also called the **"Boundaries of an Array".**

ARRAY MEMORY ALLOCATION:

* Array can be declared in one of two ways in C++.

(1) IN STACK:

int ar[50];
→ Declares an array (ar) of 50 integers on Stack memory.
→ Automatically deallocated when goes out of scope.

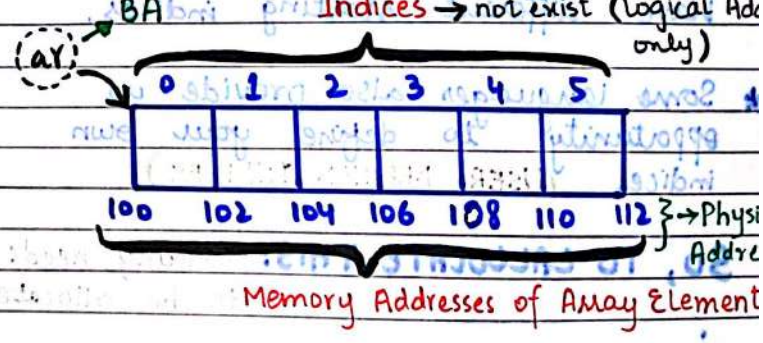
(2) IN HEAP:

int *ar = new int[50];
→ Declares an array (ar) of 50 integers on Heap memory.
→ Needs to be deallocated by programmer with the help of **delete []ar;** statement.

ARRAY DESCRIPTOR TABLE

* Whenever array creates, there is a descriptor table behind it.
* This table is known as: "ARRAY DESCRIPTOR TABLE".
* It holds **Name** of array.
* It holds **Base Address** (Starting Address of array.
* It holds **Word Size.**
* It holds **Lower Bound** (LB) and **Upper Bound** (UB).

int ar[10];



Base Address = 100, Element Size (Word Size) = 2 Bytes, Lower Bound (index) = 0 and, Upper Bound (index) = 5.

Name	Base Address	Word Size	Lower Bound	Upper Bound
ar	100	2 Bytes	0	5

* In most of the arrays, Base Address is written also "Alpha" or "Beta".

ARRAY MEMORY ALLOCATION:

* Most language start with index '0' like C, C++, etc.

* But most of the language also have different starting indices.

* Some languages also provide us opportunity to define your own indices (USER DEFINED INDICES).

SO, TO CALCULATE THIS: (Memory needs to be allocated)

Upper Bound (Index) = 5

Lower Bound (Index) = 0

} These can also be used for Bound checking.

Count = UB − LB + 1

Size (Array) = 5 − 0 + 1

Size (Array) = 6

→ Now, calculating Memory size of Array:

Base Address = 100

Element Size = 2 Bytes (Word size)

Size (Array) = 6

5	4	3	2	1	0

Memory Size (Array) = Size (Array) × Word Size

= 6 × 2

= 12 Bytes