

## Asymptotic Analysis

# Topics to be Covered

---

- Time Complexity
- Big Oh Notation
- Solved Problem

# Introduction

---

- ✓ A data structure is the organization of the data in a way so that we can use it efficiently.
- ✓ To use data efficiently we have to store it efficiently.

## Efficiency

Efficiency of a data structure is always measured in terms of **SPACE** and **TIME**

An **ideal** data structure could be the one that takes the **least possible time** for all its operations and consumes the **least memory space**.

Our focus will be on finding the  
**TIME COMPLEXITY**

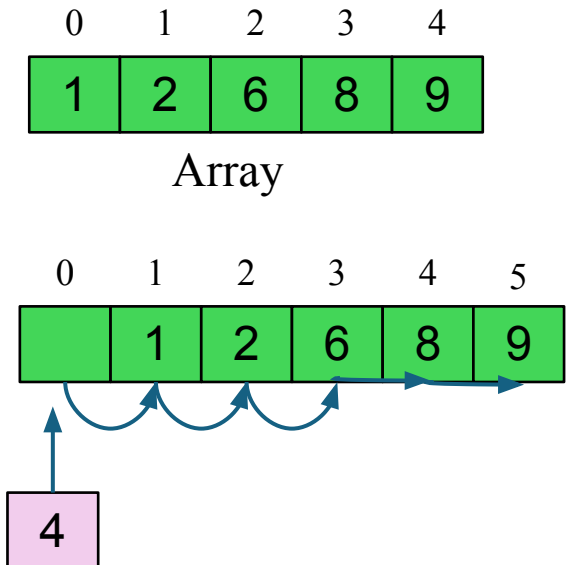
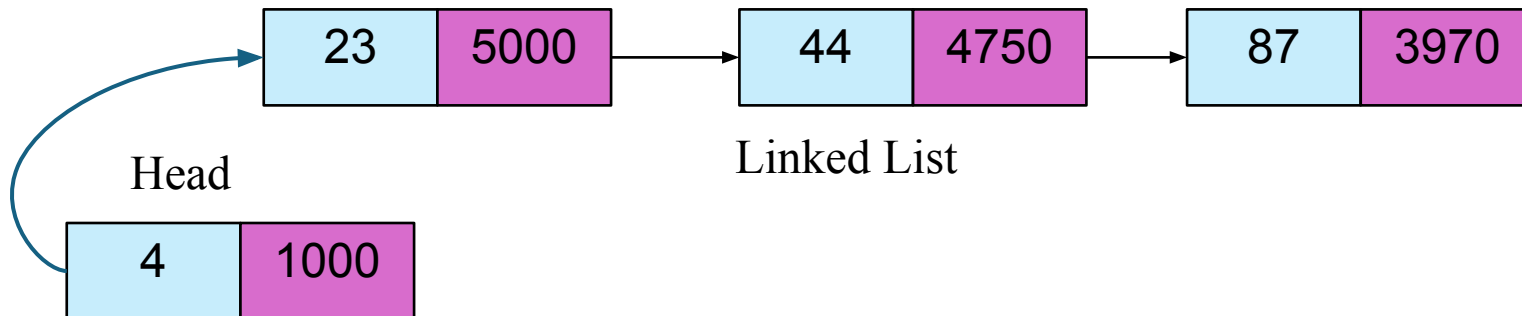
# Cont.

On what basis we could compare the time complexity of the data structure?

On the basis of operations performed on them

Example

Target: Add data at the beginning to the list



# Cont.

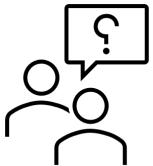
---



## Conclusion

Inserting an element at the beginning of the list is way faster in the linked list than arrays

In this way we can compare data structures, and this will ultimately give the idea to select the best possible data structure for a particular operation



But how to find the **time complexity** or **running time** of an operation performed on data structures?

# How to find time complexity?

---



## Method #1

Examine the exact running time?

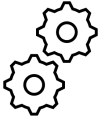
Pick up some machine and turn the timer on. Run the operation for different inputs on the data structures you want to compare one by one and see how much time a particular operation will take on these data structures.

For better analysis, you can compare the data structures on different machines.

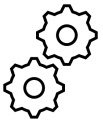
# Cont.



## Problems with this approach



It might be possible that for some input size, **first data structure** gives the best performance and for the other input size, the **second gives the best performance**.



It might also be possible that in **some machine** for some input size of data structure, it is performing well while in the other machine, the second data structure is performing well with a different size.



Therefore, examining the exact running time is not the best solution to calculate the time complexity.

# Cont.

---



So, what's the best solution to this problem?



# Cont.



## Points to be noted:

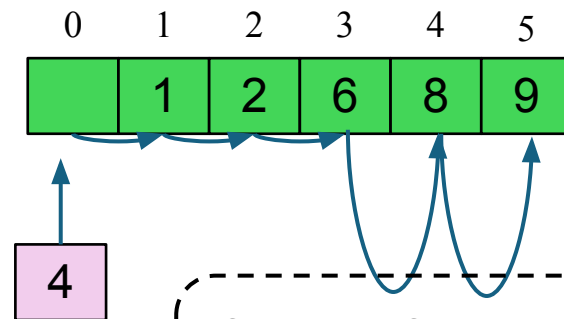
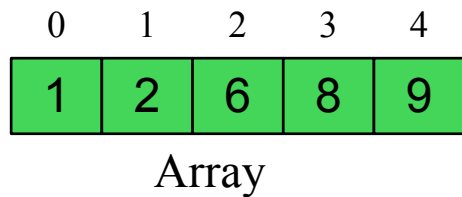


Measuring the actual running time is not practical at all.



The running time generally depends on the **size of the input**

## Example



If one shift takes one unit of time, then 5 shifts will take 5 units of time

But what if we have 10000 elements in the array

In that case, we have to make 10000 shifts which will take 10000 units of time

# Cont.

---



This clearly shows that the running time generally depends on the size of the input

# Definition

- ✓ Therefore, if the size of the input is  $n$ , then  $f(n)$  is a function of  $n$  denotes the time complexity.
- ✓ In other words,  $f(n)$  represents the number of instructions executed for the input value  $n$ .

For example,

```
for ( i =0; i<n; i++){  
    cout<<"Hello World";  
}
```

Assuming, this instruction will take 1 unit of time

`cout` instruction is executed  $n$  number of times.

Therefore,  $f(n) = n$

But how to find  $f(n)$  ?

# Finding $f(n)$

---

- ✓ We can compare two data structures for a particular operation by comparing their  $f(n)$  values
- ✓ We are interested in growth rate of  $f(n)$  with respect to  $n$  because it might be possible that for smaller input size, one data structure may seem better than the other but for larger input size it may not.
- ✓ This concept is applicable in comparing the two algorithms as well.

# Finding $f(n)$

## Example

$f(n) = 5n^2 + 6n + 12$  { Represents number of instructions executed by a program }

For  $n = 1$

$$\% \text{ of running time due to } 5n^2 = \frac{5}{5+6+12} \times 100 = 21.74 \%$$

$$\% \text{ of running time due to } 6n = \frac{6}{5+6+12} \times 100 = 26.09 \%$$

$$\% \text{ of running time due to } 12 = \frac{12}{5+6+12} \times 100 = 52.07 \%$$



It seems like most of the time consumed here

# Finding $f(n)$

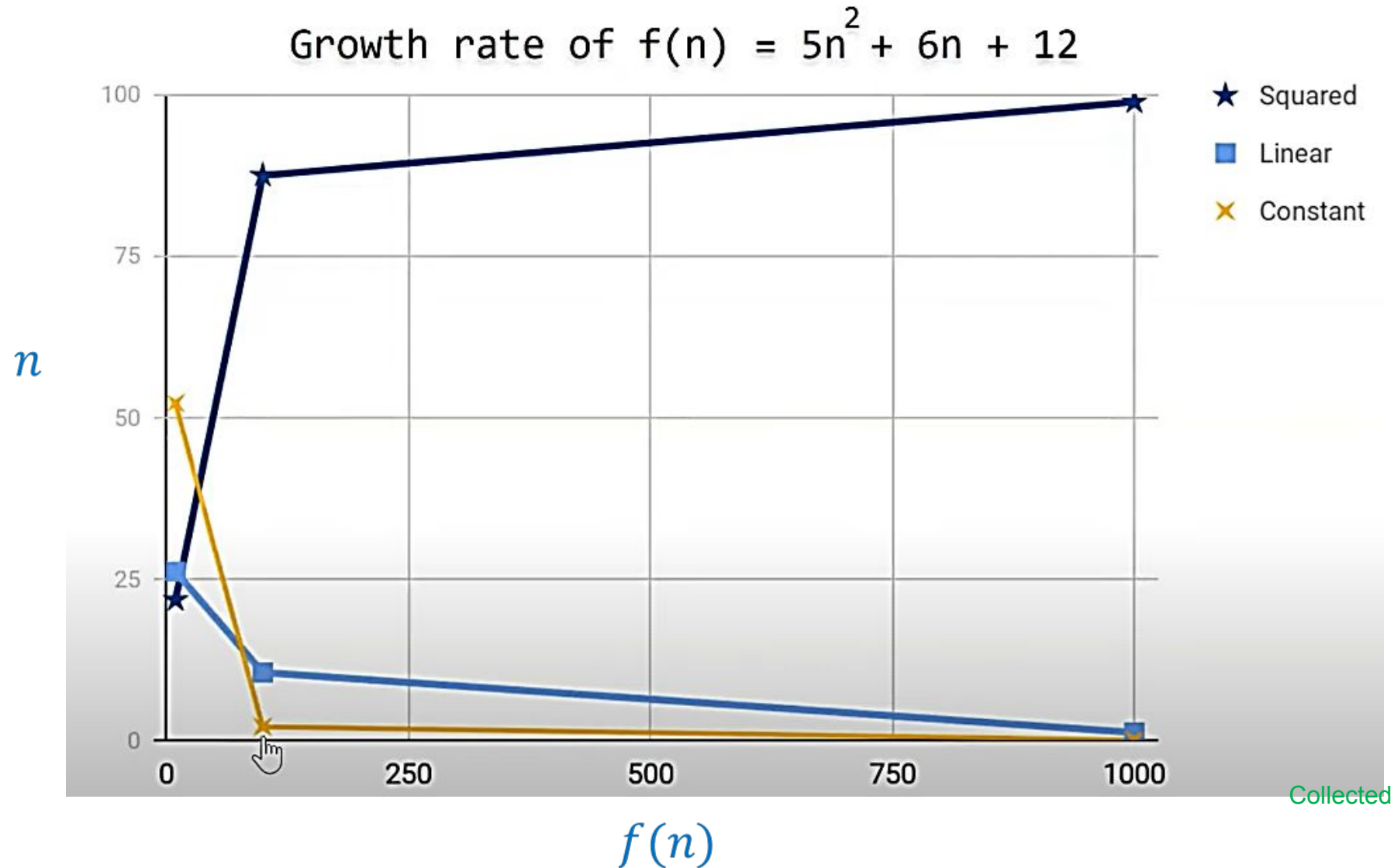
## Example

$$f(n) = 5n^2 + 6n + 12$$

1	21.74 %	26.09 %	52.17 %
10	87.41 %	10.49 %	2.09 %
100	98.79 %	1.19 %	0.02 %
1000	99.88 %	0.12 %	0.0002 %

It is clear observation that for larger values of  $n$ , the squared term consumes the 99% of time

# Finding $f(n)$



# Finding $f(n)$

## Example

$$f(n) = 5n^2 + \boxed{6n + 12} \quad \times$$

1	21.74 %	26.09 %	52.17 %
10	87.41 %	10.49 %	2.09 %
100	98.79 %	1.19 %	0.02 %
1000	99.88 %	0.12 %	0.0002 %

So, there is no harm if we can eliminate the rest of the terms as they are not contributing much to the time



# Finding $f(n)$

---

Therefore,

$$f(n) = 5n^2$$

- ✓ We are getting the approximate time complexity which is very near to the actual result.

This approximate measure of time complexity is called  
**Asymptotic Complexity**

# Big O Notation

---

- ✓ Big O notation is used to measure the performance of any algorithm by providing the order of growth of the function.
- ✓ It gives the least upper bound on a function by which we can make sure that the function will never grow faster than this upper bound.
- ✓ We want the approximate runtime of the operations performed on data structures.
- ✓ We are not interested in the exact runtime.
- ✓ Big O notation will help us to achieve the same

# Big O Notation

---

If  $f(n)$  and  $g(n)$  are the two functions, then

$$f(n) = O(g(n))$$

If there exists constants  $c$  and  $n_0$  such that

$$f(n) \leq c \cdot g(n), \quad \text{for all } n \geq n_0$$

This simply means that  $f(n)$  does not grow faster than  $g(n)$

That means, we are calculating the **worst-case time complexity**.

# Examples

---

$$f(n) = n$$

$$g(n) = 2n$$

Is  $f(n) = O(g(n))$  ?

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

$$n \leq c \cdot 2n \quad \text{For } c = 1 \text{ and } n_0 = 1$$

So,

$$f(n) = O(n)$$

It simply means that  $f(n)$  will grow linearly.

# Examples

---

$$f(n) = 4n + 3$$

$$g(n) = n$$

Is  $f(n) = O(g(n))$  ?

Take  $c = 5$

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

$$4n + 3 \leq c \cdot n$$

$$4n + 3 \leq c \cdot n$$

$$4n + 3 \leq 5n$$

$$3 \leq 5n - 4n$$

$$n \geq 3$$

Therefore

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

*for all  $n \geq 3$*

*where  $c = 5$  and  $n_0 = 3$*

$$f(n) = O(n)$$

It simply means the growth rate of  $f(n)$  is linear. It will not suddenly behave like a quadratic or any other function

# Conclusion

---

- ✓ The idea of Big Oh notation is to give the upper bound on a particular function and eventually it leads to the worst-case time complexity
- ✓ Big Oh notation ensures that the  $f(n)$  will not behave like a quadratic or cubic function suddenly.
- ✓ Big O notation helps us in finding the growth rate of the function without plugging in different values of  $n$

# Conclusion

---

$$f(n) = 5n^2 + 4$$

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

Is  $f(n) = O(g(n))$  ?

Take  $c = 6$

$$\begin{aligned} f(n) &\leq c \cdot g(n) \\ 5n^2 + 4 &\leq c \cdot n \end{aligned}$$

$$\begin{aligned} 5n^2 + 4 &\leq 6n^2 \\ n^2 &\geq 4 \\ n &\geq 2 \end{aligned}$$

Therefore

$$\begin{aligned} f(n) &\leq c \cdot g(n) \\ \text{for all } n &\geq 2 \\ \text{where } c &= 6 \text{ and } n_0 = 2 \end{aligned}$$

$$f(n) = O(n^2)$$

It simply means the growth rate of  $f(n)$  is quadratic.

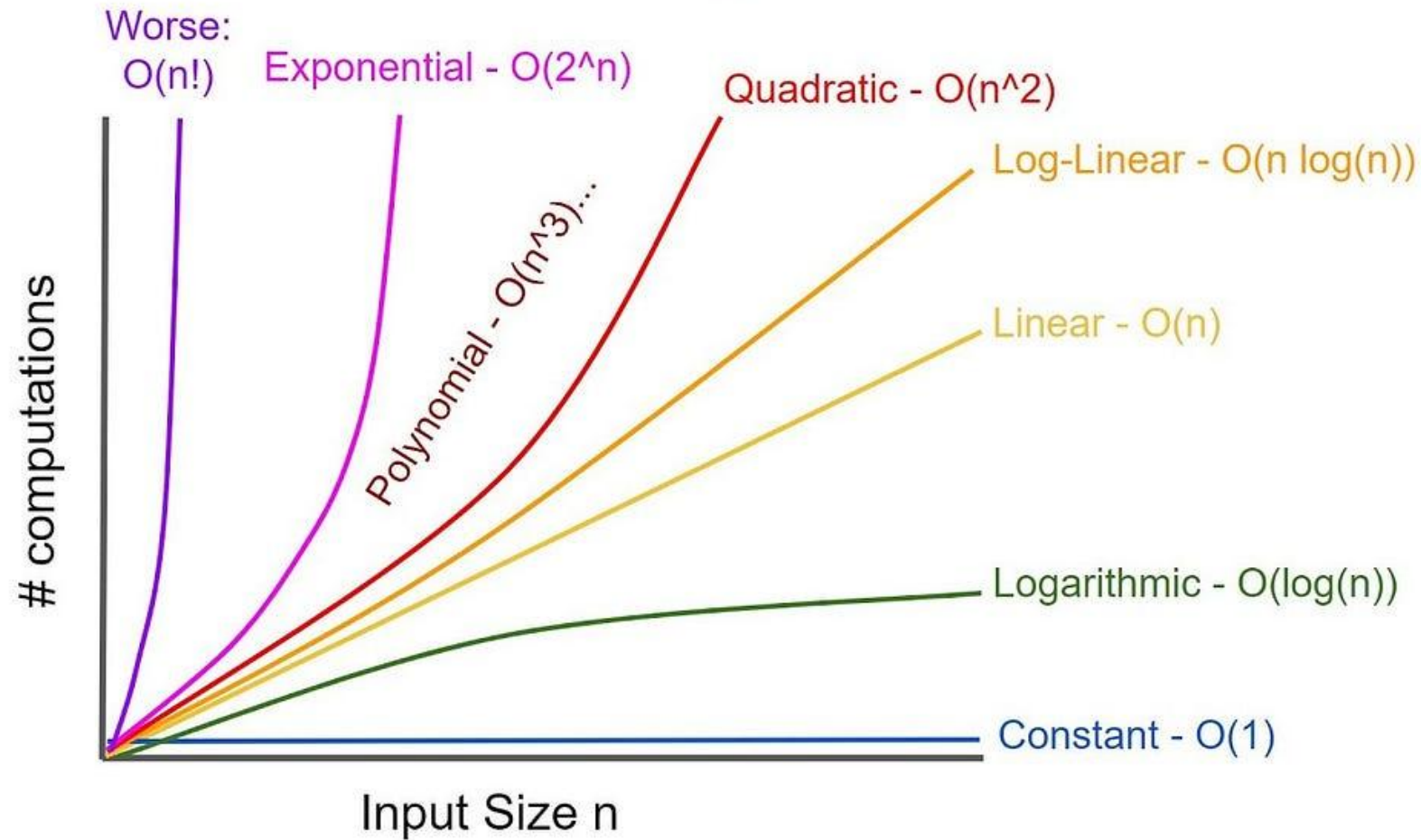
# Growth Rate of Standard Functions

g(n)						
n	$\log_2 n$	n	$n \log_2 n$	$n^2$	$n^3$	$2^n$
1	0	1	0	1	1	2
2	1	2	2	4	8	4
4	2	4	8	16	64	16
8	3	8	24	64	512	256
16	4	16	64	256	4096	65536
32	5	32	160	1024	32768	$429 \times 10^7$





# Growth Rate of Standard Functions



# Example

Program to calculate the sum of first n natural numbers

```
int main()
{
    int i, sum = 0, n; ← 1 time
    cin >> n; ← 1 time

    for ( i = 1; i <= n; i++ ){
        sum = sum + i; ← n times
    }
    cout << sum << endl; ← 1 time
    return 0; ← 1 time
}
```

$$f(n) = n + 4$$

# Example

$$f(n) = n + 4$$

$$\text{Let } g(n) = n$$

Is  $f(n) = O(g(n))$  ?

$$f(n) \leq c \cdot g(n)$$
$$n + 4 \leq c \cdot n$$

Take  $c = 2$

$$n + 4 \leq 2n$$
$$n \geq 4$$

Therefore

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

*for all  $n \geq 4$*   
*where  $c = 2$  and  $n_0 = 4$*

$$f(n) = O(n)$$

Linear Growth



Can we find a better solution?

# Example

Program to calculate the sum of first n natural numbers

```
int main()  
{  
  int i, sum = 0, n;      ← 1 time  
  cin >> n;               ← 1 time  
  cout << (n*(n+1)/2) << endl; ← 1 time  
  return 0;               ← 1 time  
}
```

$$f(n) = 4$$

# Example

---

$$f(n) = 4$$

$$\text{Let } g(n) = 1$$

Is  $f(n) = O(g(n))$  ?

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

Therefore

$$4 \leq c \cdot 1$$

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

for all  $n \geq 1$

Take  $c = 5$

where  $c = 5$  and  $n_0 = 1$

$$4 \leq 5$$

$$f(n) = O(1)$$

→ Constant Growth

# Guidelines

## Loops

```
for ( i =1; i<=n; i++){  
    //statement  
}
```

Loop executes  $n$  times

Total time =  $O(n)$

## Nested Loops

```
for ( i =1; i<=n; i++){  
    for ( j =1; j<=n; j++){  
        //statement  
    }  
}
```

Outer Loop executes  $n$  times

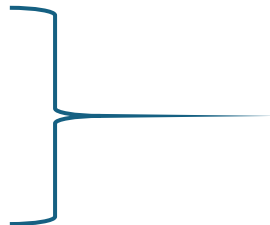
Inner Loop executes  $n$  times

Total time =  $n \times n = O(n^2)$

# Guidelines


## Consecutive Statements

```
int x = 2;  
int i;  
x = x + 1;
```




Total time = 3 units

```
for ( i =1; i<=n; i++){  
    //statement  
}
```



Loop executes  $n$  times

```
for ( i =1; i<=n; i++){  
    for ( j =1; j<=n; j++){  
        //statement  
    }  
}
```



Loop executes  $n^2$  times

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

# Cont.

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

$$\text{Let } g(n) = n^2$$

Is  $f(n) = O(g(n))$  ?

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

Take  $c = 2$

$$n^2 + n + 3 \leq 2n^2$$
$$n^2 - n \geq 3$$
$$n(n - 1) \geq 3$$

$$n \geq 3 \text{ or } n \geq 4$$

Therefore

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

for all  $n \geq 3$   
where  $c = 2$  and  $n_0 = 3$

$$f(n) = O(n^2)$$

→ Quadratic Growth



# Guidelines

## If-Else Statements

```
cin>>n;
```

```
if (n==0){  
    //statement  
}
```

```
else{  
    for ( i =1; i<=n; i++){  
        //statement  
    }  
}
```

Worst case running time = Test + if part or else part (whichever is larger)

**For if part:**

$n == 0$  takes constant time

Statement inside if takes constant time

**Total time = 1 + 1 =  $O(1)$**

# Guidelines

## If-Else Statements

```
cin>>n;
```

```
if (n==0){  
    //statement  
}
```

```
else{  
    for ( i =1; i<=n; i++){  
        //statement  
    }  
}
```

For else part:

$n == 0$  takes constant time

Statement get executed  $n$  times

$$\text{Total time} = 1 + n = O(n)$$

$$\text{Time Complexity} = O(n)$$

# Logarithmic Complexity

---

## What is Logarithm

$$\log_2(8)$$

The above logarithm says “how many times 2 has been multiplied by itself to obtain value 8

Logarithmic time complexity is achieved when the problem size is cut down by a fraction

# Cont.

```
for ( i =1; i<=n; ){  
    //statement  
    i = i*2;  
}
```

Iter 1	Initially	$i = 1$	$2^0$
Iter 2		$i = 2$	$2^1$
Iter 3		$i = 4$	$2^2$
Iter 4		$i = 8$	$2^3$
Iter 5		$i = 16$	$2^4$
•			
•			
•			
Iter k		$i = n$	$= 2^{k-1}$

How many iterations are there or the statements under for loop evaluated?

# Cont.

$$n = 2^{k-1}$$

$$k - 1 = \log_2 n$$

$$k = \log_2 n + 1$$

**Time Complexity** =  $O(\log_2 n)$

## Example

```
for ( i =1; i<=32; ){  
    cout<<"Hello";  
    i = i*2;  
}
```

$$k = \log_2 32 + 1$$

$$k = 6$$

# Cont.

```
for ( i =n; i>=1; ){  
    //statement  
    i = i/2;  
}
```

Iter 1

Initially

$$i = n/2^0$$

Iter 2

$$i = n/2^1$$

Iter 3

$$i = n/2^2$$

Iter 4

$$i = n/2^3$$

•

•

•

Iter k

$$i = n/2^{k-1} = 1$$

How many iterations are there or the statements under for loop evaluated?

# Cont.

---

$$n/2^{k-1} = 1$$

$$n = 2^{k-1}$$

$$k = \log_2 n + 1$$

Time Complexity =  $O(\log_2 n)$

# Exercise 1

```
for ( i =1; i<=n/2; i++){  
    for ( j =1; j<=n; j=j*2){  
        cout<<"Hello"  
    }  
}
```

**Ans:**  $O(n \log_2 n)$

```
for ( i =1; i<=n; i++){  
    for ( j =1; j<=n; j=j*2){  
        for ( k =1; k<=n; k=k*2 )  
  
            cout<<"Hello"  
        }  
    }  
}
```

**Ans:**  $O(n (\log_2 n)^2)$



## Exercise 2

```
for ( i =1; i<=n; i++){  
    for ( j =1;  $j \leq i^2$ ; j++){  
        cout<<"Hello";  
    }  
}
```

Ans:  $O(n^3)$

```
for ( i =1; i<=n; i++){  
    for ( j =1;  $j \leq i^2$ ; j++){  
        for ( k =1;  $k \leq n/2$ ; k++){  
            cout<<"Hello";  
        }  
    }  
}
```

Ans:  $O(n^4)$

# Solved Problem

---

```
void fun(int n) {  
    int i, j, k, count = 0; ←----- O(1)  
    for ( i =n/2; i<=n; i++)  
        for ( j =1; j+n/2<=n; j++)  
            for ( k =1; k<=n; k = k*2)  
                count++;  
}
```

# Solved Problem

```
void fun(int n) {
```

```
    int i, j, k, count = 0; ←----- O(1)
```

```
    for ( i =n/2; i<=n; i++)
```

```
        for ( j =1; j+n/2<=n; j++)
```

```
            for ( k =1; k<=n; k = k*2)
```

```
                count++;
```

```
}
```

Iter 1      $i = \frac{n}{2} + 0$

Iter 2      $i = \frac{n}{2} + 1$

Iter 3      $i = \frac{n}{2} + 2$

Iter 4      $i = \frac{n}{2} + 3$

•

•


•

Iter k      $i = \frac{n}{2} + (k - 1) = n$

$$\Rightarrow k - 1 = n - \frac{n}{2} \quad \Rightarrow k = \frac{n}{2} + 1$$

# Solved Problem

---

```
void fun(int n) {  
    int i, j, k, count = 0;   
    for ( i = n/2; i <= n; i++)  
        for ( j = 1; j + n/2 <= n; j++)  
            for ( k = 1; k <= n; k = k*2)  
                count++;  
}
```

The diagram shows two dashed blue arrows pointing from complexity boxes to the code. The first arrow points from a box labeled  $O(1)$  to the initialization of `count = 0`. The second arrow points from a box labeled  $O(n)$  to the first `for` loop.

# Solved Problem

```
void fun(int n) {
```

```
    int i, j, k, count = 0; ←  $O(1)$ 
```

```
    for ( i = n/2; i <= n; i++) ←  $O(n)$ 
```

```
        for ( j = 1;  $j + n/2 \leq n$ ; j++)
```

```
            for ( k = 1;  $k \leq n$ ; k = k*2)
```

```
                count++;
```

```
}
```

$$\Rightarrow j + \frac{n}{2} \leq n$$

$$\Rightarrow j \leq n - n/2$$

$$\Rightarrow j \leq n/2$$

Iter 1      $j = 1$

Iter 2      $j = 2$

Iter 3      $j = 3$

Iter 4      $j = 4$

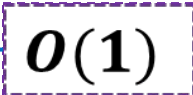
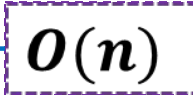
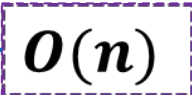
•

•

Iter k      $j = k = \frac{n}{2}$

$$\Rightarrow k = \frac{n}{2}$$

# Solved Problem

```
void fun(int n) {  
    int i, j, k, count = 0;   
    for ( i = n/2; i <= n; i++)   
        for ( j = 1; j + n/2 <= n; j++)   
            for ( k = 1; k <= n; k = k*2)  
                count++;  
}
```

# Solved Problem

```
void fun(int n) {
```

```
    int i, j, k, count = 0; ←  $O(1)$ 
```

```
    for ( i = n/2; i <= n; i++) ←  $O(n)$ 
```

```
        for ( j = 1;  $j + n/2 \leq n$ ; j++) ←  $O(n)$ 
```

```
            for ( k = 1;  $k \leq n$ ; k = k*2)
```

```
                count++;
```

```
}
```

$$\Rightarrow n = 2^{p-1}$$

$$\Rightarrow p - 1 = \log_2 n$$

$$\Rightarrow p = \log_2 n + 1$$

Iter 1  $k = 1$   $2^0$   
Iter 2  $k = 2$   $2^1$   
Iter 3  $k = 4$   $2^2$

•

•

•

Iter p  $k = n = 2^{p-1}$

# Solved Problem

```
void fun(int n) {
```

```
    int i, j, k, count = 0; ←  $O(1)$ 
```

```
    for ( i = n/2; i <= n; i++) ←  $O(n)$ 
```

```
        for ( j = 1;  $j + n/2 \leq n$ ; j++) ←  $O(n)$ 
```

```
            for ( k = 1;  $k \leq n$ ; k = k*2) ←  $O(\log_2 n)$ 
```

```
                count++;
```

```
}
```



# Solved Problem

```
void fun(int n) {
```

```
    int i, j, k, count = 0; ←  $O(1)$ 
```

```
    for ( i = n/2; i <= n; i++) ←  $O(n)$ 
```

```
        for ( j = 1;  $j + n/2 \leq n$ ; j++) ←  $O(n)$ 
```

```
            for ( k = 1;  $k \leq n$ ; k = k*2) ←  $O(\log_2 n)$ 
```

```
                count++;
```

```
}
```

**Time Complexity =  $n \times n \times \log_2 n = O(n^2 \log_2 n)$**

# Homework

# Find the time complexity of the following programs

```
void fun (int n){  
    int i = 1;  
    while ( i<=n){  
        int j = n;  
        while ( j>=0){  
            j = j/2;  
        }  
        i = 2*i;  
    }  
}
```

**Ans =  $O((\log_2 n)^2)$**

```
void fun(int n) {  
    int i, j, k, count = 0;  
    for ( i =n/2; i<=n; i++)  
        for ( j =1; j<=n; j =2*j)  
            for ( k =1; k<=n; k = k++)  
                count++;  
}
```

**Ans =  $O(n^2 \log_2 n)$**

# Homework

---

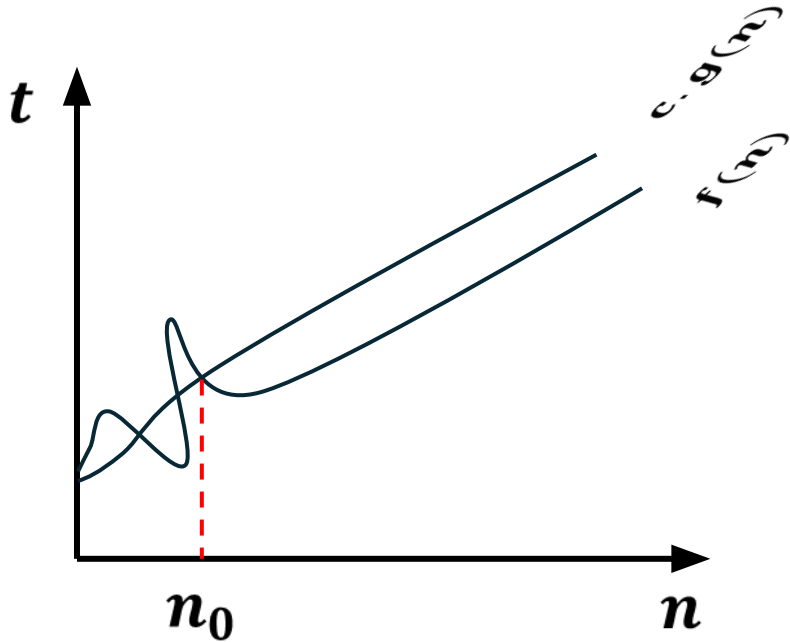
# Find the time complexity of the following program

```
void fun(int n) {  
    int i, j;  
    for ( i =1; i<=n/3; i++)  
        for ( j =1; j<=n; j +=4)  
            Cout<<"Hello";  
}
```

**Ans =  $O(N^2)$**

# Asymptotic Notation

## Big – Oh ( $O$ )



Worst Case

Least upper bound

If  $f(n)$  and  $g(n)$  are the two functions, then

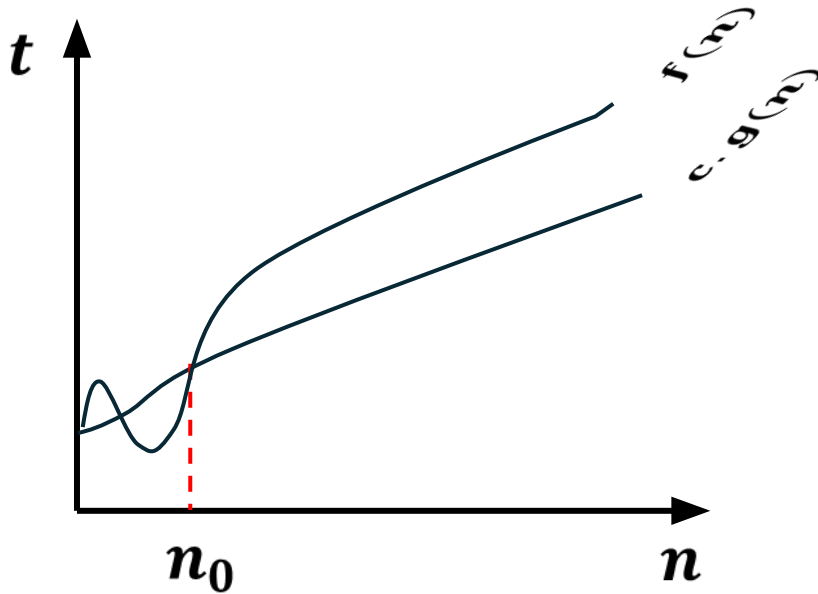
$$f(n) = O(g(n))$$

If there exists constants  $c$  and  $n_0$  such that

$$f(n) \leq c \cdot g(n), \quad \text{for all } n \geq n_0$$

# Cont.

## Big – Omega ( $\Omega$ )



Best Case

Greatest lower bound

If  $f(n)$  and  $g(n)$  are the two functions, then

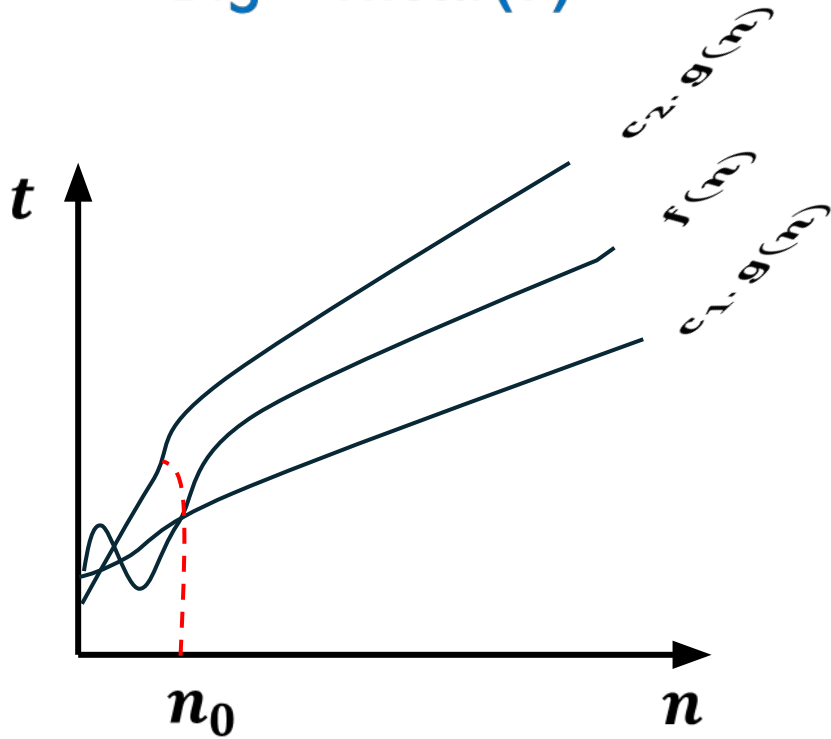
$$f(n) = \Omega(g(n))$$

If there exists constants  $c$  and  $n_0$  such that

$$f(n) \geq c \cdot g(n), \quad \text{for all } n \geq n_0$$

# Cont.

## Big – Theta ( $\theta$ )



If  $f(n)$  and  $g(n)$  are the two functions, then

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

If there exists constants  $c_1, c_2$  and  $n_0$  such that

$$c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n), \quad \text{for all } n \geq n_0$$

Average Case

**Thank you**