31/05/25

# Session-2

# UNDERSTANDING

## TIME AND SPACE COMPLEXITY

TechnoCrats GSV

# Table Of Content

# Recap of the Previous Session

Arrays, Strings, Recursion

Consider the array given below:

| | |
|---|---|
| name[0] | Abhinav |
| name[1] | Chaman |
| name[2] | Deepak |
| name[3] | Ishan |
| name[4] | Geeta |
| name[5] | Harsh |
| name[6] | Mayank |

**Ques** : How many elements would be moved if the name Anant has to be added in it from the start?

(a) 4
(b) 5
(c) 6
(d) 7

**Ques** : How many elements would be moved if the name Ishan has to be deleted from it?

(a) 3
(b) 4
(c) 5
(d) 6

# Recap of the Previous Session

Arrays, Strings, Recursion

ISRO CSE 2013

**Ques** : Let A(1:8, -5:5, -10:5) be a three dimensional array.
How many elements are there in the array A?

**Ques** : strlen(“Oxford University Press”) is ?

(a) 22
(b) 23
(c) 24
(d) 25

# What is Time and Space Complexity?

Time and Space Complexity are used to measure the efficiency of an algorithm.

- Time Complexity → Measures how much time an algorithm takes as input size grows.
- Space Complexity → Measures how much memory (RAM) it uses.

# Understanding Algorithm Complexity Notations

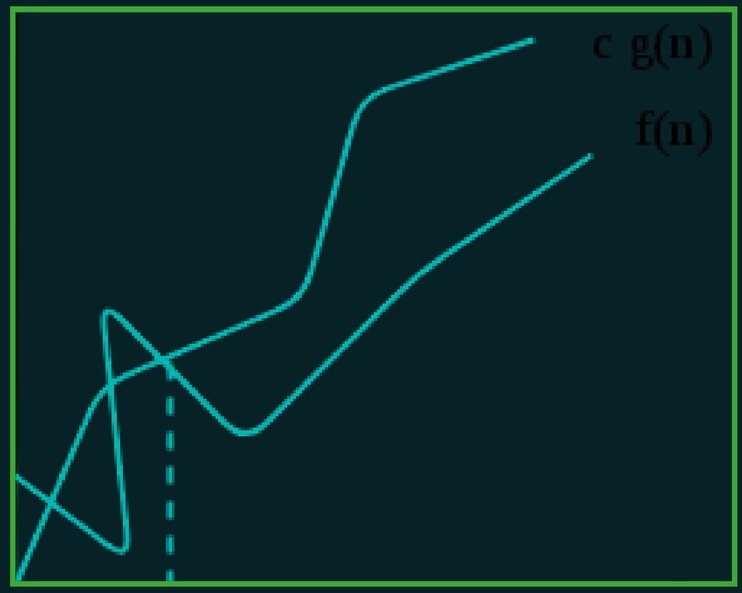
| Notation | Represents | Case |
| --- | --- | --- |
| Big O (O) | Upper Bound | Worst Case |
| Big Omega (Ω) | Lower Bound | Best Case |
| Big Theta (Θ) | Tight Bound | Average |

# BIG O NOTATION

The Big O notation, where O stands for ‘order of’

If f(n) ≤ cg(n), c > 0, ∀ n ≥ n0 , then f(n) = O(g(n)) and g(n) is an asymptotically tight upper bound for f(n)

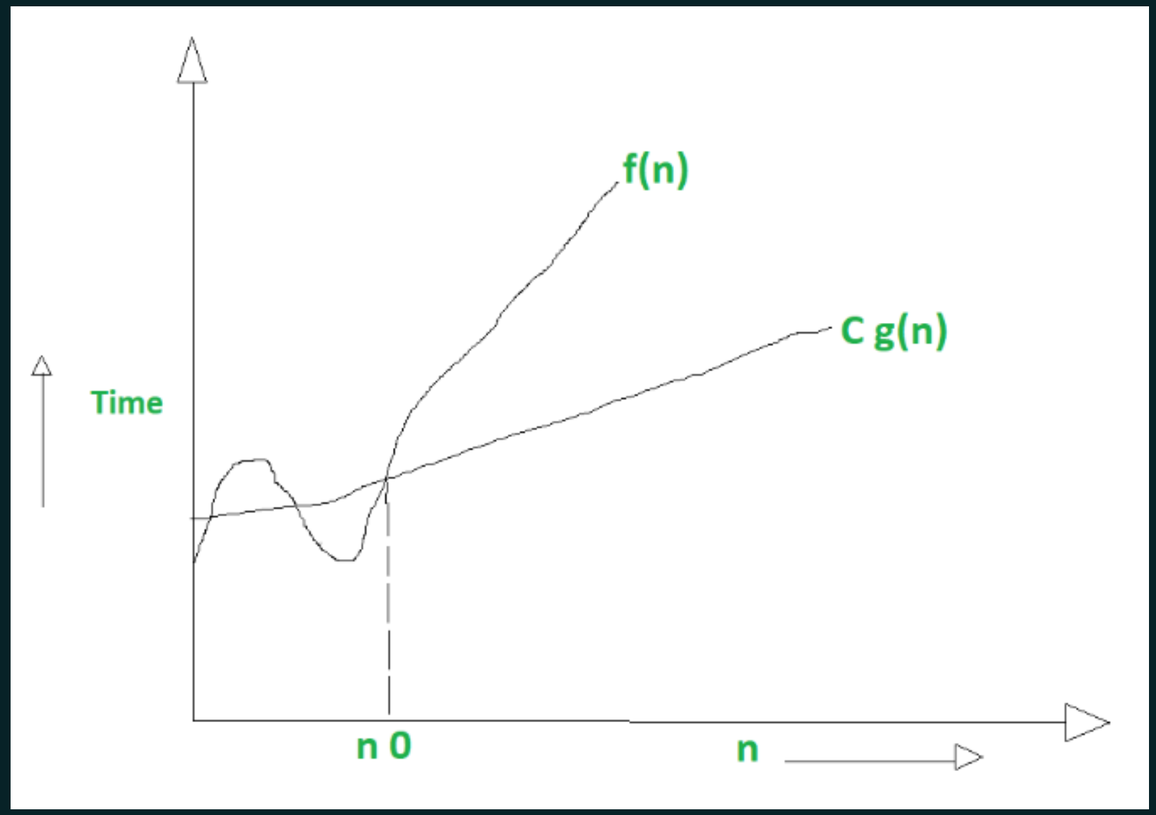
| g(n) | f(n) = O(g(n)) |
|---|---|
| 10 | O(1) |
| 7n2 + logn+5 | O(n^2) |
| 2n3 + 3n2 + 5n – 10 | O(n^3) |



# OMEGA NOTATION (Ω)

The Omega notation provides a tight lower bound for f(n).

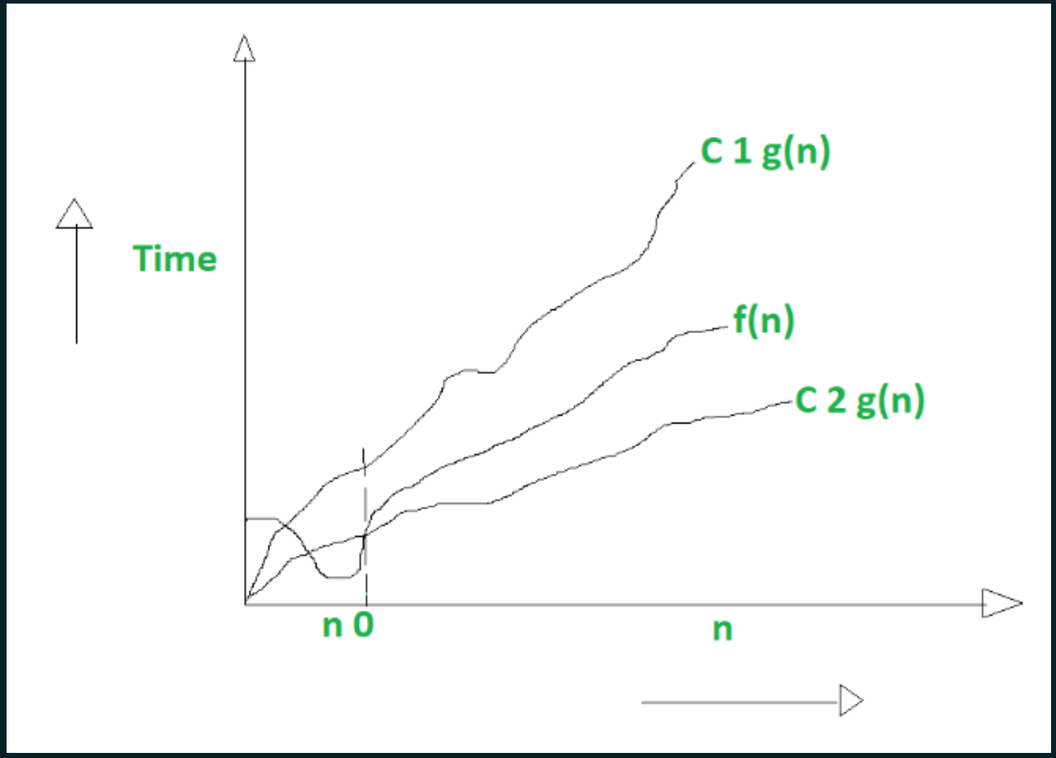
If $cg(n) \leq f(n)$, $c > 0$, $\forall\ n \geq n0$ , then $f(n) \in \Omega(g(n))$ and g(n) is an asymptotically tight lower bound for f(n)



# THETA NOTATION (Q)

Theta notation provides an asymptotically tight bound for f(n).

If f(n) is between c1g(n) and c2g(n), $\forall$ n ≥ n0 , then f(n) $\in$ $\Theta$(g(n)) and g(n) is an asymptotically tight bound for f(n) and f(n) is amongst h(n) in the set.



# Little o notation

This notation provides a non-asymptotically tight upper bound for f(n)

An imprecise analogy between the asymptotic comparison of functions f(n) and g(n) and the relation between their values can be given as:

$f(n) = O(g(n)) \approx f(n) \leq g(n)$

$f(n) = o(g(n)) \approx f(n) < g(n)$

# Little Omega Notation (ω)

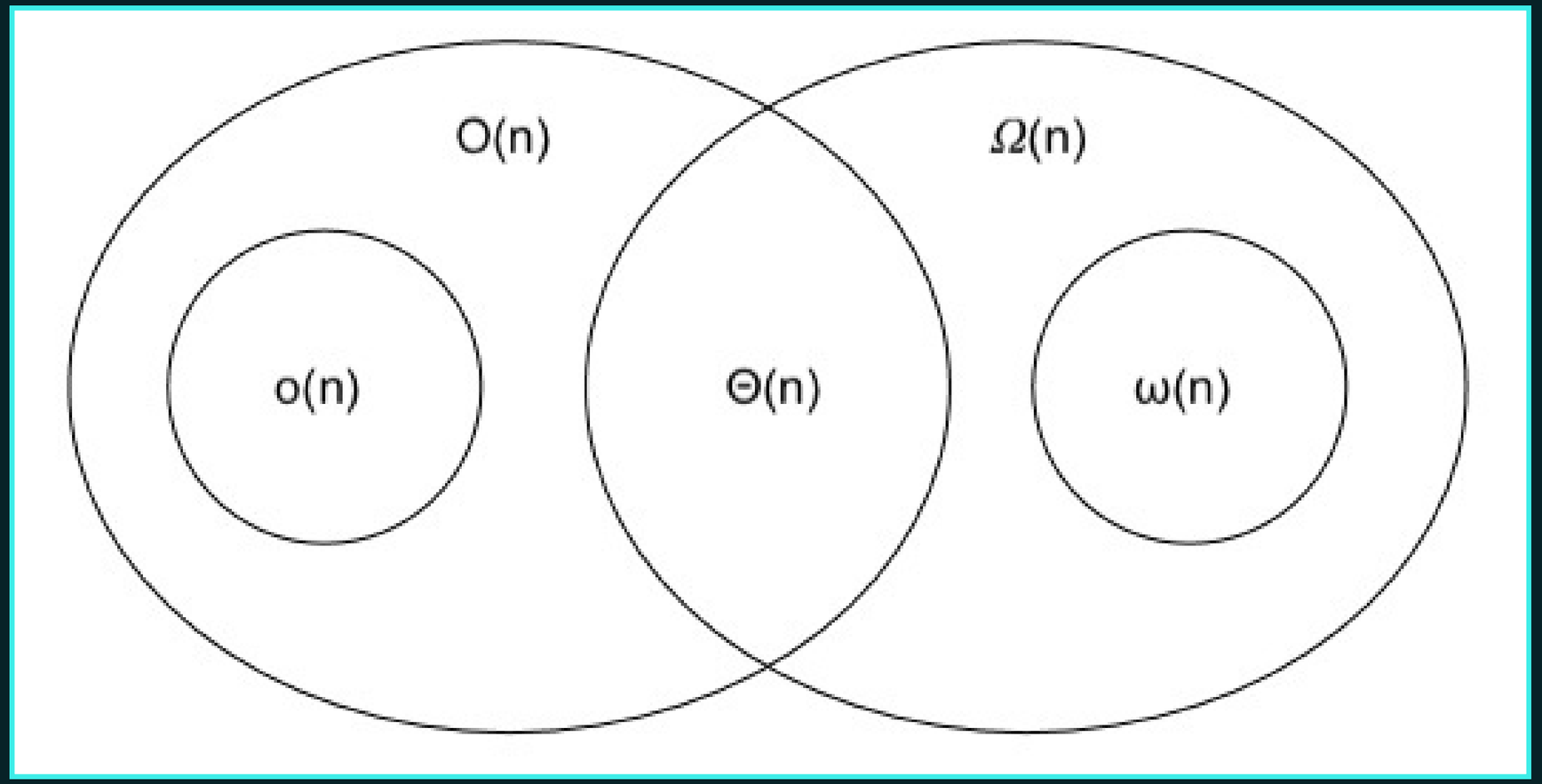
This notation provides a non-asymptotically tight lower bound for f(n)

An imprecise analogy between the asymptotic comparison of functions f(n) and g(n) and the relation between their values can be given as:

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

$$f(n) = \omega(g(n)) \approx f(n) > g(n)$$

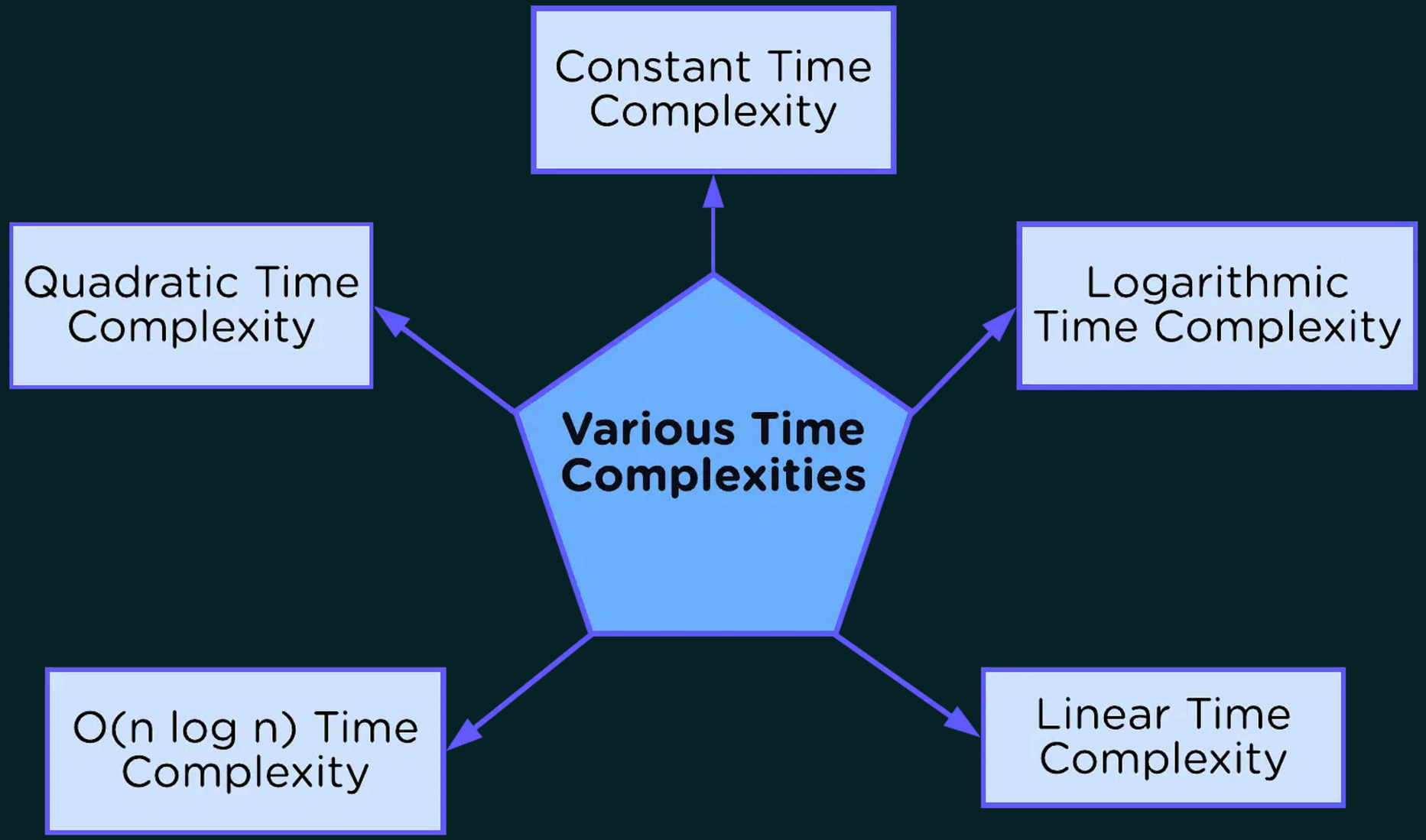
# Asymptotic Notations : Venn Diagram



# Properties of Asymptotic Notations

| | Reflexive | Symmetric | Transitive |
|---|---|---|---|
| **Big O (O)** | Yes | No | Yes |
| **Big Omega (Ω)** | Yes | No | Yes |
| **Big Theta (Θ)** | Yes | Yes | Yes |
| **Little o** | No | No | Yes |
| **Little omega** | No | No | Yes |

# Types of Time Complexities with Examples



# Types of Time Complexities


## Linear loops

```
for(i=0;i<100;i++)
statement block;
```

f(n) = n

```
for(i=0;i<100;i+=2)
    statement
```

f(n) = n/2

# Types of Time Complexities

## Logarithmic Loops

```
for(i=1;i<1000;i*=2)
  statement block;
```

f(n) = log n

```
for(i=1000;i>=1;i/=2)
  statement block;
```

f(n) = log n

# Types of Time Complexities


## Linear logarithmic

```
for(i=0;i<10;i++)
    for(j=1; j<10;j*=2)
     Statement block;
```

f(n) = n log n

# Types of Time Complexities


## Quadratic loop

```
for(i=0;i<10;i++)
    for(j=0; j<10;j++)
        statement block;
```

f(n) = n^2

# Space Complexity

It includes:

1. Input space (memory used to store the input)
2. Auxiliary space (temporary or extra memory used during computation, like variables, recursion stack, arrays, etc.)

## Remember : Space Complexity ≠ Size of Input

It refers to the extra space the algorithm uses to solve the problem.

# Types of Time Complexities with Examples

## O(1)

```c
#include<stdio.h>
int main(){
    int arr[10]={5,10,15,20,25,30,35,40,45,50};
    //accessing the element at index 5 (6th element)
    printf("The value of index 5 is: %d\n",arr[5]);
    return 0;
}
```

O(1)

O(1)

O(1)

f(n)=O(1)+O(1)+O(1=3*O(1)

# Types of Time Complexities with Examples

## O(logn)

```
int binarySearch(int arr[],int n,int target){
    int st=0;
    int end=n-1;

    while(st<=end){
        int mid=st+(end-st)/2;

        if(arr[mid]==target){
            return mid;                 //Target found at the index mid
        }else if(arr[mid]<target){
            st=mid+1;                   //search in right half
        }else{
            end=mid-1;                  //search in left half
        }
    }
    return -1;                          //Target not found
}
```

O(1)

O(1)

O(logn)

O(1)

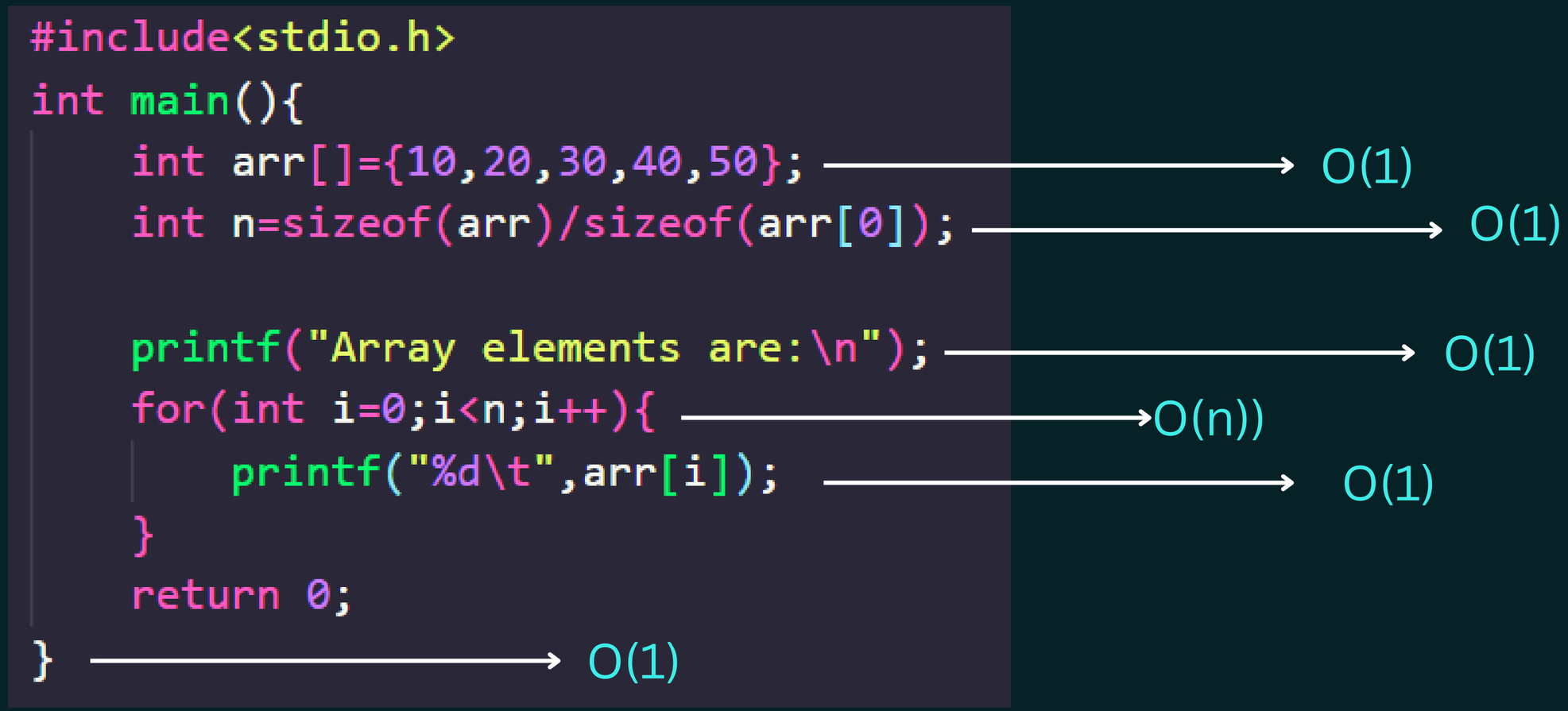
O(1)

**f(n)=O(1)+O(1)+O(logn)+O(1)+O(1)**

**f(n)=4*O(1)+O(logn)**

# Types of Time Complexities with Examples

## O(n)




**f(n)=O(1)+O(1)+O(1)+O(n)+O(1)+O(1)**

**f(n)=5*O(1)+O(n)**

# Types of Time Complexities with Examples

## O(n^2)

```c
#include<stdio.h>
int main(){
    int arr[]={1,2,3,4,5};
    int n=sizeof(arr)/sizeof(arr[0]);

    printf("All pairs in the array:\n");
    for(int i=0;i<n;i++){
        for(int j=0;j<n;j++){
            printf("(%d,%d)\n",arr[i],arr[j]);
        }
    }
    return 0;
}
```

Nested Loop

O(1)

O(1)

O(1)

O(n)

O(n)

O(1)

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

$$f(n)=4*O(1)+O(n^2)$$

# Types of Time Complexities with Examples

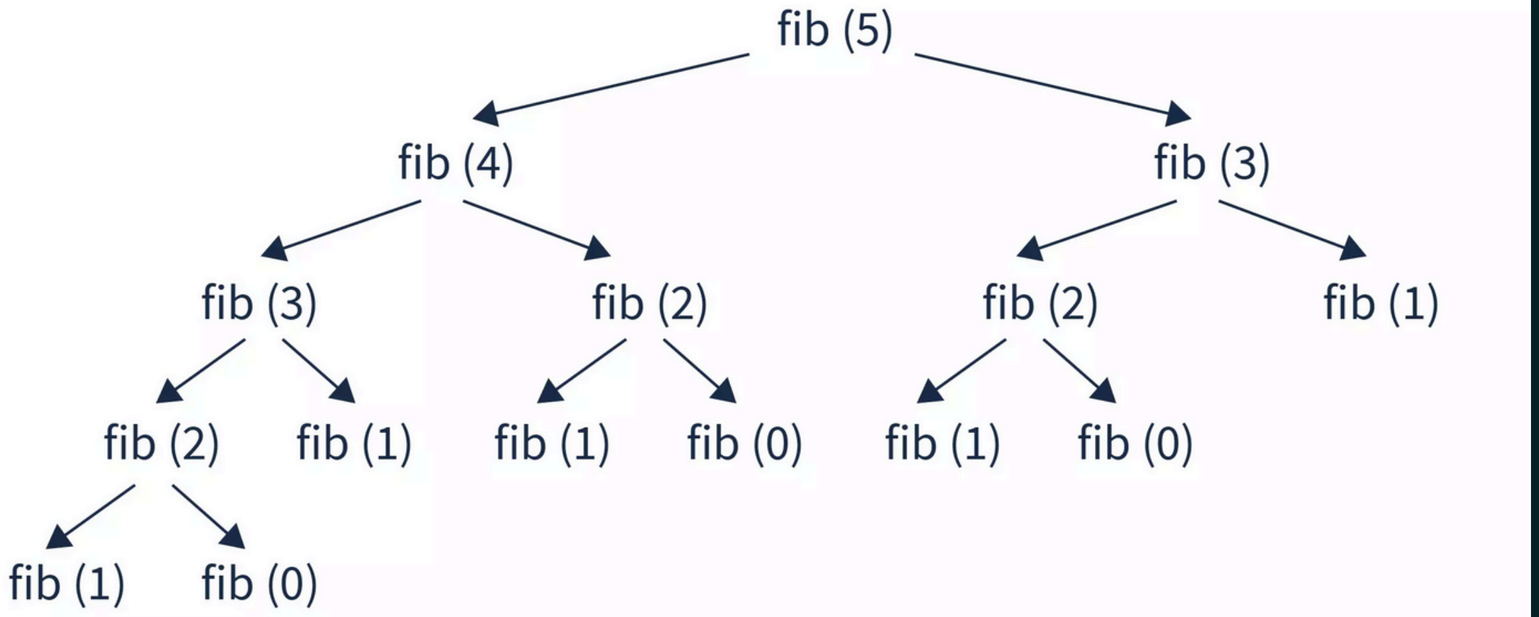
## O(n^3)

```c
#include <stdio.h>      // O(1) - Compile-time inclusion

int main() {
    int arr[] = {1, 2, 3, 4, 5};          // O(1) time, O(n) space (array of size 5)
    int n = sizeof(arr)/sizeof(arr[0]); // O(1) time, O(1) space

    printf("All triplets (i,j,k) are:\n"); // O(1) time

    // Triple nested loop
    for (int i = 0; i < n; i++) {           // O(n)
        for (int j = 0; j < n; j++) {       // O(n)
            for (int k = 0; k < n; k++) {   // O(n)
                printf("(%d,%d,%d)\n", arr[i], arr[j], arr[k]); // O(1)
            }
        }
    }
    return 0;
}
```

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

# Types of Time Complexities with Examples

## O(2^n)




```c
int fibonacci(int n){
    if(n==0){
        return 0;
    }
    else if(n==1){
        return 1;
    }else{
        return fibonacci(n-1)+fibonacci(n-2);
    }
}
```

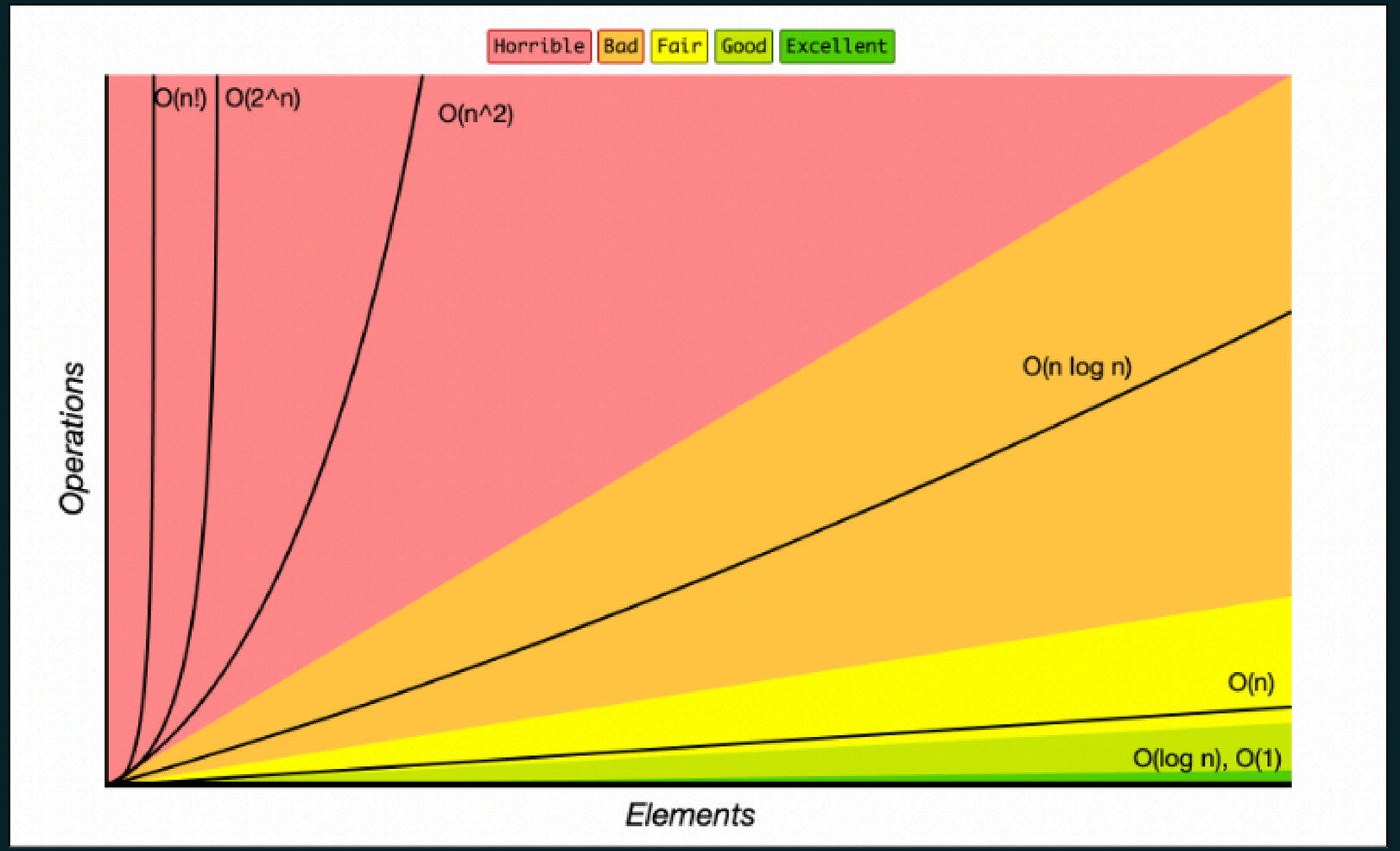
# Types of Time Complexities with Examples


## Your Task is : Read these Problems

**Merge Sort**
https://www.geeksforgeeks.org/merge-sort/

**Solving Traveling Salesman Problem**
https://www.geeksforgeeks.org/traveling-salesman-problem-using-branch-and-bound-2/

# Categories of Algorithms



# Categories of Algorithms

- **Constant time algorithm** : running time complexity given as **O(1)**
- **Linear time algorithms** have running time complexity given as **O(n)**
- **Logarithmic time algorithms** have running time complexity given as **O(log n)**
- **Polynomial time algorithms** have running time complexity given as **O(n^k)**
- **Exponential time algorithms** have running time complexity given as **O(2^n)**

**Table 2.2** Number of operations for different functions of n

| n | O(1) | O(log n) | O(n) | O(n log n) | $O(n^2)$ | $O(n^3)$ |
|---|---|---|---|---|---|---|
| 1 | 1 | 1 | 1 | 1 | 1 | 1 |
| 2 | 1 | 1 | 2 | 2 | 4 | 8 |
| 4 | 1 | 2 | 4 | 8 | 16 | 64 |
| 8 | 1 | 3 | 8 | 24 | 64 | 512 |
| 16 | 1 | 4 | 16 | 64 | 256 | 4096 |

# Recurrence Relation

## Methods to solve Recurrence

- Substitution
- Recurrence Tree
- Master Method

# Substitution Method

## Substitution Method – Example 1

- We make a guess for the solution and then we use mathematical induction to prove the guess is correct or incorrect.

**Example 1:**

Time to solve the instance of size $n - 1$

Time to solve the instance of size $n$

$$T(n) = T(n-1) + n \quad \text{-------} (1)$$

- Replacing $n$ by $n - 1$ and $n - 2$, we can write following equations.

$$T(n-1) = T(n-2) + n - 1 \quad \text{-------} (2)$$

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

- Substituting equation 3 in 2 and equation 2 in 1 we have now,

$$T(n) = T(n-3) + n - 2 + n - 1 + n \quad \text{----} (4)$$

# Substitution Method

## Substitution Method – Example 1

$$T(n) = T(n-3) + n - 2 + n - 1 + n \quad \text{----} (4)$$

- From above, we can write the general form as,

$$T(n) = T(n-k) + (n-k+1) + (n-k+2) + \ \ldots + n$$

- Suppose, if we take $k = n$ then,

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

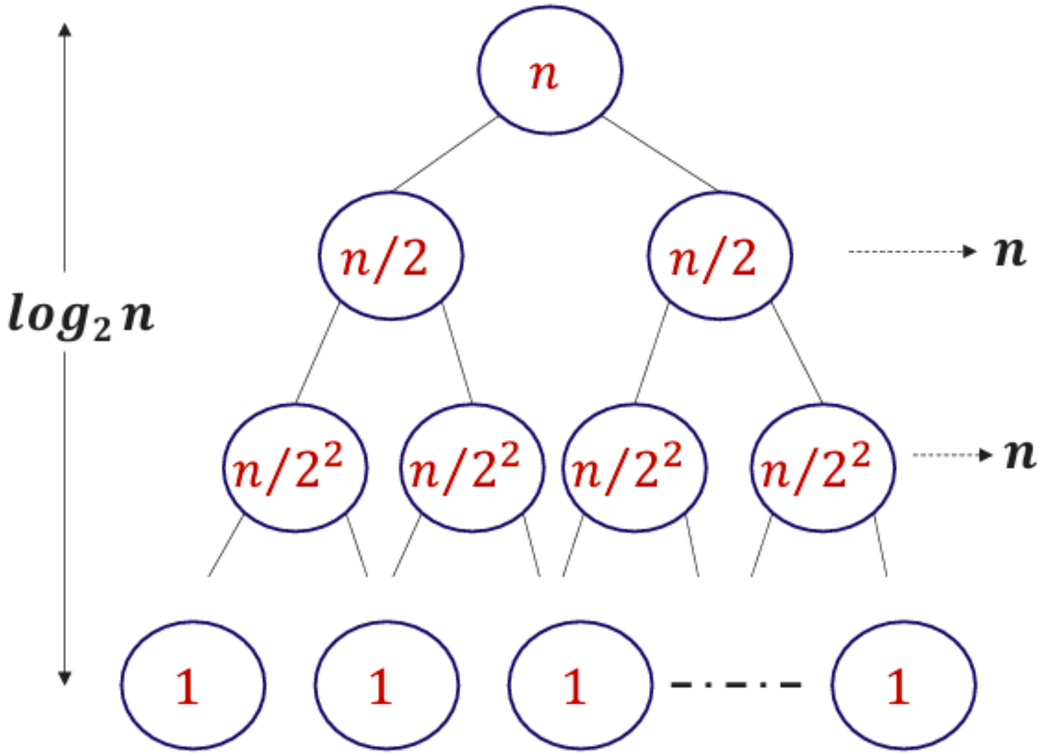
$$T(n) = 0 + 1 + 2 + \ \ldots + n$$

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

# Recurrence Tree Method

## Recurrence Tree Method

The recursion tree for this recurrence is




**Example 1:** $T(n) = 2T(n/2) + n$

- When we add the values across the levels of the recursion tree, we get a value of $n$ for every level.
- The bottom level has $2^{\log n}$ nodes, each contributing the cost $T(1)$.
- We have $n + n + n + \ldots\ldots$ $\log n$ $times$

$$T(n) = \sum_{i=0}^{log_2 n-1} n + 2^{log\, n} T(1)$$

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

$$\boxed{T(n) = O(n \log n)}$$

# Master Method

## Your task is: Read Master Method & Solve Some Examples

Used to solve divide-and-conquer recurrence relations of the form:

$$T(n) = aT(n/b) + f(n)$$

- `a ≥ 1` is the number of subproblems
- `b > 1` is the input size shrink factor
- `f(n)` is the cost of the work done outside recursive calls

**Link:** **Master Theorm : GFG**

# Methods to Find Time Complexity

## Summary

| Method | Best For | Use Case Example |
|---|---|---|
| **Substitution** | Simple recursions | $T(n) = T(n-1) + 1$ |
| **Master Theorem** | Divide-and-conquer | $T(n) = 2T(n/2) + n$ |
| **Recursion Tree** | Visual learners | $T(n) = 2T(n/2) + n$ |

# BASIC ARRAY OPERATION

| Operation | Shifting Required? | Time Complexity |
|---|---|---|
| **Insert at start** | Yes (right shift) | O(n) |
| **Insert in middle** | Yes (right shift) | O(n) |
| **Insert at End** | No | O(1) |
| **Delete at Start** | Yes (left Shift) | O(n) |
| **Delete in middle** | Yes (left Shift) | O(n) |
| **Delete at End** | No | O(1) |

# Practice Time

**Ques** : Which of the following time complexities grows the fastest?

(a) O(n log n)

(b) O($n^2$)

(c) O(2^n)

(d) O(n)

**Ques** : Algorithm A and B have a worst-case running time of O(n) and O(logn), respectively. Therefore, algorithm B always runs faster than algorithm A.

(a) True

(b )False

# Practice Time

```
    int a = 0, b = 0;
for (int i = 0; i < N; i++) {
    a = a + rand();
}
for (int j = 0; j < M; j++) {
    b = b + rand();
}
```

# Practice Time

```
int a = 0;
for (i = 0; i < N; i++) {
    for (j = N; j > i; j--) {
        a = a + i + j;
    }
}
```

# Practice Time

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

# Q&A

# THANK YOU