- **Space Complexity**
  - The space complexity of an algorithm is the amount of memory it needs to run to completion
  - Space Complexity = Fixed Part + Variable Part

    $S(P) = c + S_P$, Where P is any algorithm
    - A fixed part:
      - It is independent of the characteristics of the inputs and outputs.
      - Eg:
        - Instruction space(i.e., space for the code)
        - space for simple variables and fixed-size component variables
        - space for constants
    - A variable part:
      - It is dependent on the characteristics of the inputs and outputs.
      - Eg:
        - Space needed by component variables whose size is dependent on the particular problem instance being solved
        - Space needed by referenced variables
        - Recursion stack space.
- **Time Complexity**
  - The time complexity of an algorithm is the amount of computer time it needs to run to completion. Compilation time is excluded.
  - Time Complexity = Frequency Count * Time for Executing one Statement

Space Complexity

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do
                for j=1 to n do
                        c[i,j] := a[i,j] + b[i,j];
}
```

Space Complexity= Space for parameters and Space for local variables

m→1 n→1 a[]→mn b[]→mn c[]→mn i→1 j→1

Space complexity = 3mn + 4

Space Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

Space Complexity
= Space for Stack
= Space for parameters + Space for local variables + Space for return address

For each recursive call the amount of stack required is 3
Space for parameters: a→1 n→1
Space for local variables: No local variables
Space for return address: 1

Total number of recursive call = n+1

Space complexity = 3(n+1)

- Space needed by referenced variables
- Recursion stack space.

Time Complexity

- The time complexity of an algorithm is the amount of computer time it needs to run to completion. Compilation time is excluded.
- Time Complexity = Frequency Count * Time for Executing one Statement
- Frequency Count → Number of times a particular statement will execute

Eg1: Find the time and space complexity of matrix addition algorithm

	Step/Execution	Frequency Count	Total Frequency Count
Algorithm mAdd(m,n,a,b,c)	0	0	0
{	0	0	0
for i=1 to m do	1	m+1	m+1
for j=1 to n do	1	m(n+1)	mn+m
c[i,j] := a[i,j] + b[i,j];	1	mn	mn
}	0	0	0
			2mn + 2m +1

Time Complexity = 2mn + 2m + 1

Space Complexity = Space for parameters and Space for local variables

m→1 n→1 a[]→mn b[]→mn c[]→mn i→1 j→1

Space complexity = 3mn + 4

Eg2: Find the time and space complexity of recursive sum algorithm

Time Complexity

```
Algorithm mAdd(m,n,a,b,c)
{
        for i=1 to m do
                for j=1 to n do
                        c[i,j] := a[i,j] + b[i,j];
}
```

Frequency Count

for i=1 to m do → m+1

for j=1 to n do → m(n+1)

c[i,j] := a[i,j] + b[i,j]; → mn

Time Complexity = 2mn + 2m + 2

Time Complexity

```
Algorithm RSum(a,n)
{
        if(n<=0)
                return 0;
        else
                return a[n] + RSum(a,n-1)
}
```

	Frequency Count n<=0	Frequency Count n>0
if(n<=0)	1	1
return 0;	1	0
else	0	0
return a[n] + RSum(a,n-1)	0	1+T(n-1)
	2	2 + T(n-1)

$$\text{Time Complexity} = T(n) = \begin{cases} 2 & \text{if } n<=0 \\ 2+T(n-1) & \text{Otherwise} \end{cases}$$

```
        a[0] := x;
    }
```

Answer: Here the elementary operation is a[j] :=a[j-1]

The time complexity is depends on the size of array(n)

$T(n) = 1 + 2 + \ldots\ldots\ldots + n-1 = \mathbf{n(n-1)/2}$

Best Case, Worst Case and Average Case Complexity

- In certain case we cannot find the exact value of frequency count. In this case we have 3 types of frequency counts
  - Best Case : It is the minimum number of steps that can be executed for a given parameter
  - Worst Case: It is the maximum number of steps that can be executed for a given parameter
  - Average Case: It is the average number of steps that can be executed for a given parameter
- Eg: Linear Search
  - Best Case: Search data will be in the first location of the array.
  - Worst Case: Search data does not exist in the array
  - Average Case: Search data is in the middle of the array.

Best, Worst and Average Case Complexities

```
Algorithm Search(a,n,x)
{
    for i:=1 to n do
      if a[i] ==x then
        return i;
    return -1;
}
```

	Best Case	Worst Case	Average Case
for i:=1 to n do	1	n+1	n/2
if a[i] ==x then	1	n	n/2
return i;	1	0	1
return -1;	0	1	0
Time Complexity =	3	2n + 2	n+1

Asymptotic Notations

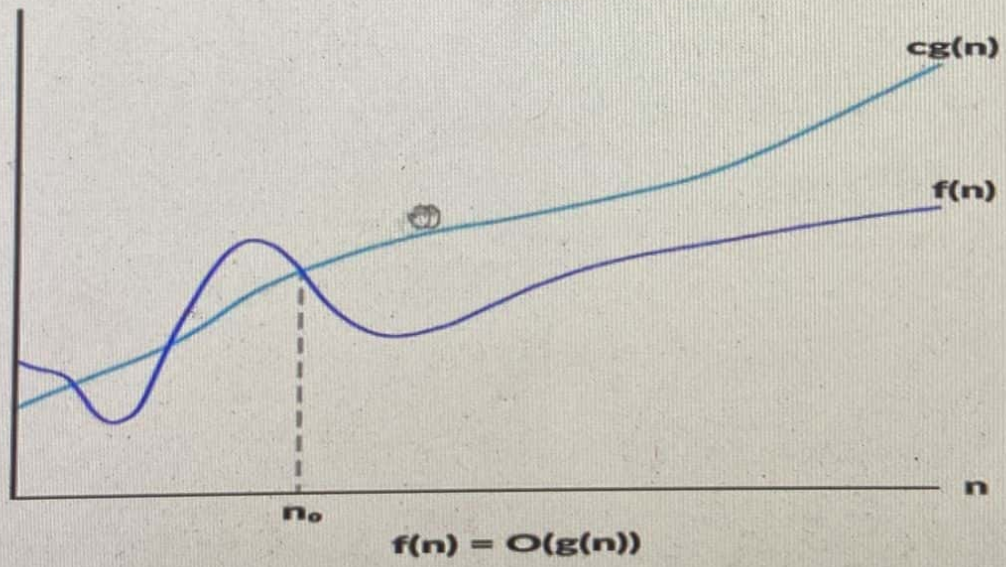
- It is the mathematical notations to represent frequency count.
- 5 types of asymptotic notations
  - **Big Oh (O)**
  - **Omega (Ω)**
  - **Theta (Θ)**
  - **Little Oh (o)**
  - **Little Omega (ω)**

Big Oh (O) Notation

Definition:

The function $f(n) = O(g(n))$ iff there exists 2 positive constants c and n_0 such that

$0 \leq f(n) \leq c\ g(n)$ for all $n \geq n_0$



Big Oh (O) Notation

- It is the measure of longest amount of time taken by an algorithm(Worst case).
- It is asymptotically tight upper bound
- O(1) : Computational time is constant
- O(n) : Computational time is linear
- $O(n^2)$: Computational time is quadratic
- $O(n^3)$: Computational time is cubic
- $O(2^n)$: Computational time is exponential

Big O

Q Find O for following funct. $f(n) = 3n+2$

→ $0 \leq f(n) \leq cg(n) \quad n \geq n_0$

$f(n) = 3n+2$

$g(n) = 3n$ → Highest term selected as $g(n)$ from $f(n)$

Assume (normally as the value with n as coeff +1)

$c = 3+1 = 4$

$0 \leq \underbrace{3n+2}_{LH} \leq \underbrace{4n}_{RH}$

if $n = 1$

$5 \leq 4$ ✗

if $n = 2$

$8 \leq 8$ ✓

$n = 3$

$11 \leq 12$ ✓

$n = 4$

$14 \leq 16$ ✓

$\therefore n_0 = 2$ & the relation holds.

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

$f(n) = O(n)$

Q $f(n) = 4n^3 + 2n + 3$

→ $f(n) = 4n^3 + 2n + 3$

$g(n) = n^3$

Assume $C = 4 + 1 = 5$

$4n^3 + 2n + 3 \leq 5n^3$

for $n = 1$

$9 \leq 5$ ✗

for $n = 2$

$39 \leq 40$ ✓

for $n = 3$

$117 \leq 135$ ✓

3+2
4
3
9

$n_0 = 2$

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

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

Q $f(n) = 2^{n+1}$

→ $f(n) = 2 \times 2^n$

$g(n) = 2^n$

Assume $C = 2+1 = 3$

$2 \times 2^n \leq 3 \times 2^n$

For $n = 1$

$4 \leq 6$ ✓

For $n = 2$

$8 \leq 12$ ✓

$n = 1$

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

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

Q Is $2^{n+1} = O(2^n)$? Justify.

→ $f(n) \leq c(g(n))$

$2^n \times 2 \leq c\, 2^n$

$2 \leq c$

so $c = 2$

Eqn holds since we could find the value of c

Q Is $2^{2n} = O(2^n)$

→ $f(n) = 2^{2n}$ $g(n) = 2^n$

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

$2^{2n} \leq c\, 2^n$

$2^n \times 2^n \leq c\, 2^n$

Is $2^{2n} = O(2^n)$?

$0 \leq 2^{2n} \leq c\, 2^n$ for $n \geq n_0$

$2^n\, 2^n \leq c\, 2^n$

$2^n \leq c$ for $n \geq n_0$

There is no value for c and n_0 that can make this true.

Therefore 2^{2n} != $\mathbf{O(2^n)}$

Is $2^{n+1} = O(2^n)$?

$0 \leq 2^{n+1} \leq c\, 2^n$ for $n \geq n_0$

$2x2^n \leq c\, 2^n$

$2 \leq c$

$2^{n+1} \leq c\, 2^n$ is True if $c=2$ and $n \geq 1$.

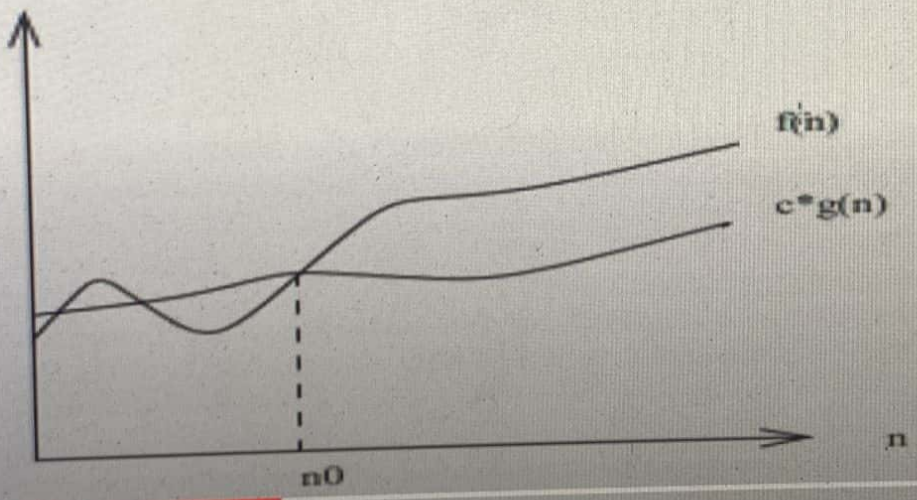
Therefore $2^{n+1} = \mathbf{O(2^n)}$

17:08 / 17:29

Omega (Ω) Notation

Definition:

The function $f(n) = \Omega(g(n))$ iff there exists 2 positive constant c and n_0 such that $f(n) \geq c\ g(n) \geq 0$ for all $n \geq n_0$



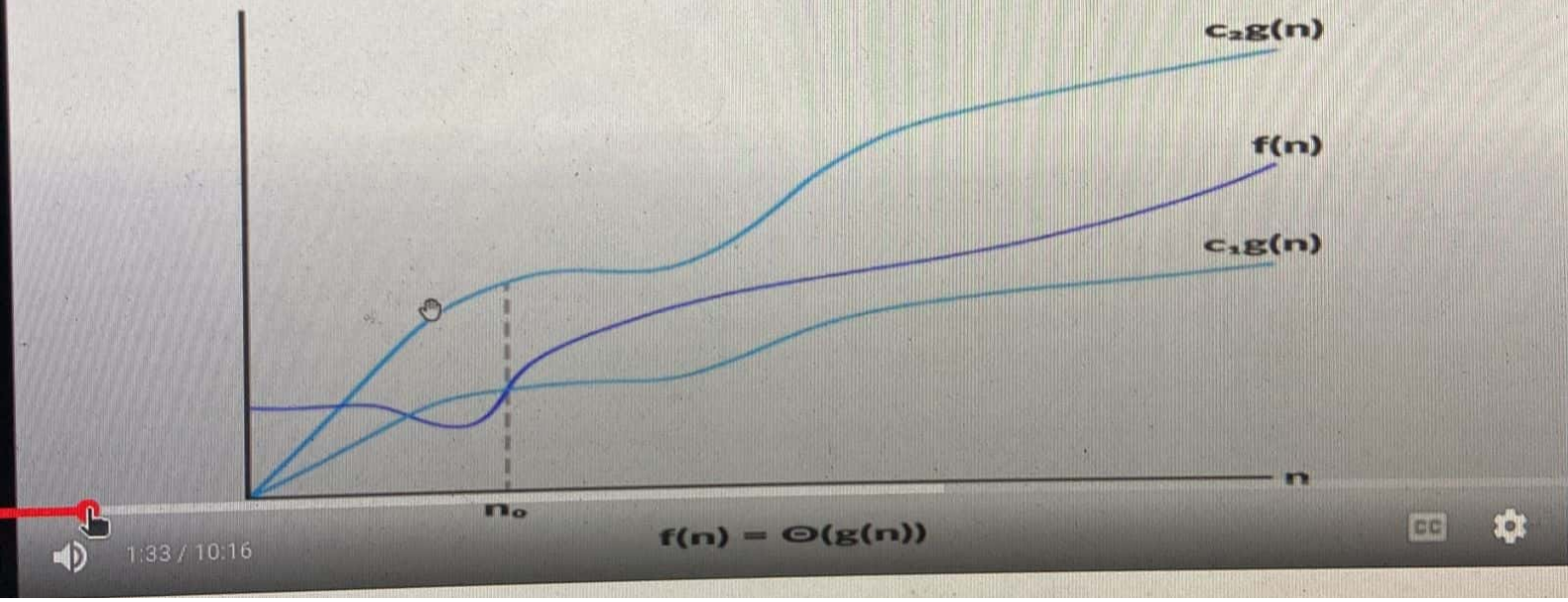
Omega (Ω) Notation

- It is the measure of smallest amount of time taken by an algorithm(Best case)
- It is asymptotically tight lower bound

Theta (Θ) Notation

Definition:

The function $f(n) = \Theta(g(n))$ iff there exists 3 positive constants c_1, c_2 and n_0 such that $0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$



Find the Θ notation of the following function

$f(n) = 3n + 2$

$0 \leq c_1\, g(n) \leq f(n) \leq c_2\, g(n)$ for all $n \geq n_0$

$c_1\, g(n) \leq f(n)$ for all $n \geq n_0$

Here $f(n) = 3n + 2$ $g(n) = n$ $c_1 = 3$ $n_0 = 1$

$3n \leq 3n + 2$ for all $n \geq 1$

$f(n) \leq c_2\, g(n)$ for all $n \geq n_0$

Here $f(n) = 3n + 2$ $g(n) = n$ $c_2 = 4$ $n_0 = 2$

$3n + 2 \leq 4n$ for all $n \geq 2$

$3n \leq 3n + 2 \leq 4n$ for all $n \geq 2$ $c_1 = 3$ $c_2 = 4$ $n_0 = 2$

Little Oh (o)

- The function f(n) = o(g(n)) iff for any positive constant c>0, there exists a constant n_0>0 such that $0 \leq f(n) < c\ g(n)$ for all $n \geq n_0$
- It is asymptotically loose upper bound

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

g(n) becomes arbitrarily large relative to f(n) as n approaches infinity

Little Omege (ω)

- The function f(n) = ω(g(n)) iff for any positive constant c>0, there exists a constant n_0>0 such that $f(n) > c\ g(n) \geq 0$ for all $n \geq n_0$
- It is asymptotically loose lower bound

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

f(n) becomes arbitrarily large relative to g(n) as n approaches infinity

amples:

Find the O notation of the following functions

a) f(n) = 3n + 2

$3n + 2 \leq 4\ n$ for all $n \geq 2$

Here f(n)= 3n + 2 g(n)=n c=4 n_0=2

- **Properties of Asymptotic Notations**
  - Reflexivity
    - $f(n) = \mathbf{O}(f(n))$
    - $f(n) = \mathbf{\Omega}(f(n))$
    - $f(n) = \mathbf{\Theta}(f(n))$
  - Symmetry
    - $f(n) = \mathbf{\Theta}(g(n))$ if and only if $g(n) = \mathbf{\Theta}(f(n)$
  - Transpose Symmetry
    - $f(n) = \mathbf{O}(g(n))$ if and only if $g(n) = \mathbf{\Omega}(f(n))$
    - $f(n) = \mathbf{o}(g(n))$ if and only if $g(n) = \boldsymbol{\omega}(f(n))$
  - Transitivity
    - $f(n) = \mathbf{O}(g(n))$ and $g(n) = \mathbf{O}(h(n))$ imply $f(n) = \mathbf{O}(h(n))$
    - $f(n) = \mathbf{\Omega}(g(n))$ and $g(n) = \mathbf{\Omega}(h(n))$ imply $f(n) = \mathbf{\Omega}(h(n))$
    - $f(n) = \mathbf{\Theta}(g(n))$ and $g(n) = \mathbf{\Theta}(h(n))$ imply $f(n) = \mathbf{\Theta}(h(n))$
    - $f(n) = \mathbf{o}(g(n))$ and $g(n) = \mathbf{o}(h(n))$ imply $f(n) = \mathbf{o}(h(n))$

0:37 / 3:24

Time Complexity

```
int fun1(int n)
{       if(n ≤ 1)
                return n;
        return 2xfun1(n-1);
}
```

	Frequency Count n<=1	Frequency Count n>1
if(n ≤ 1)	1	1
return n;	1	0
return 2xfun1(n-1);	0	1+T(n-1)
	2	2 + T(n-1)

$$\text{Time Complexity} = T(n) = \begin{cases} 2 & \text{if } n<=1 \\ 2+T(n-1) & \text{Otherwise} \end{cases}$$

2:47 / 14:33