unacademy

Function and Recursion

Course on Data Structure and Algorithms Using Python

Ankush Saklecha • Lesson 10 • Dec 17, 2023

Polymorphism:

Polymorphism is a concept of object-oriented programming, which means multiple forms or more than one form.
Polymorphism enables using a single interface with input of different datatypes, different class or may be for different number of inputs.

In python as everything is an object hence by default a function can take anything as an argument but the execution of the function might fail as every function has some logic that it follows.

For example,

```
len("hello")          # returns 5 as result

len([1,2,3,4,45,345,23,42])      # returns 8 as resul
```

In this case the function len is polymorphic as it is taking string as input in the first case and is taking list as input in the second case.

In python, polymorphism is a way of making a function accept objects of different classes if they behave similarly.

- **Method overriding : A** child class which is extending the parent class can provide different definition to any function defined in the parent class as per its own requirements.
- **Method Overloading :** Method overriding, or function overloading is a type of polymorphism in which we can define a few methods with the same name but with a different number of parameters as well as parameters can be of different types. These methods can perform a similar or different function.

Note : Python doesn't support method overloading based on different number of parameters in functions.

Defining Polymorphic Classes

```
# without polymorphism
class Square:
    side =5
    def calculate_area_sq(self):
        return self.side * self.side
class Triangle:
    base =5
    height =4
    def calculate_area_tri(self):
        return 0.5* self.base * self.height
sq = Square()
tri = Triangle()
print("Area of square: ", sq.calculate_area_sq())
print("Area of triangle: ", tri.calculate_area_tri())
```

```
# polymorphic classes
class Square:
    side =5
    def calculate_area(self):
        return self.side * self.side
class Triangle:
    base =5
    height =4
    def calculate_area(self):
        return 0.5* self.base * self.height

sq = Square()
tri = Triangle()
print("Area of square: ", sq.calculate_area())
print("Area of triangle: ", tri.calculate_area())
```

Polymorphism with Class Methods

```
class Square:
    side =5
    def calculate_area_sq(self):
        return self.side * self.side
class Triangle:
    base =5
    height =4
    def calculate_area_tri(self):
        return 0.5* self.base * self.height

sq = Square()
tri = Triangle()

for(obj in(sq, tri)):
 obj.calculate_area()
```

```
#Polymorphism with function

 side =5
    def calculate_area_sq(self):
        return self.side * self.side
class Triangle:
    base =5
    height =4
    def calculate_area_tri(self):
        return 0.5* self.base * self.height
find_area_of_shape(obj):
                obj.calculate_area()


sq = Square()
tri = Triangle()

# calling the method with different objects

find_area_of_shape(sq)
find_area_of_shape(tri)
```

Polymorphism with Inheritance

```
class Bird:
      def intro(self):
        print("There are different types of birds")

      def flight(self):
        print("Most of the birds can fly but some cannot")

class parrot(Bird):
      def flight(self):
        print("Parrots can fly")

class penguin(Bird):
      def flight(self):
        print("Penguins do not fly")

obj_bird = Bird()
obj_parr = parrot()
obj_peng = penguin()

obj_bird.intro()
obj_bird.flight()

obj_parr.intro()
obj_parr.flight()

obj_peng.intro()
obj_peng.flight()
```

Abstraction

Abstraction is one of the most important features of object-oriented programming. It is used to hide the background details or any unnecessary implementation.

Pre-defined functions are similar to data abstraction

```
from abc import ABC
class type_shape(ABC):
    def area(self):
 #abstract method
        pass
class Rectangle(type_shape):
 length = 6
 breadth = 4
 def area(self):
    return self.length * self.breadth
class Circle(type_shape):
 radius = 7
 def area(self):
    return 3.14 * self.radius * self.radius
class Square(type_shape):
 length = 4
 def area(self):
    return self.length*self.length
class triangle:
 length = 5
 width = 4
 def area(self):
    return 0.5 * self.length * self.width
r = Rectangle() # object created for the class 'Rectangle'
c = Circle() # object created for the class 'Circle'
s = Square() # object created for the class 'Square'
t = triangle() # object created for the class 'triangle'
print("Area of a rectangle:", r.area()) # call to 'area' method
defined inside the class.
print("Area of a circle:", c.area()) # call to 'area' method
defined inside the class.
print("Area of a square:", s.area()) # call to 'area' method
defined inside the class.
print("Area of a triangle:", t.area()) # call to 'area' method
defined inside the class.
```

Area of a rectangle: 24
Area of a circle: 153.86
Area of a square: 16
Area of a triangle: 10.0

Python File Open

File handling is an important part of any web application.

Python has several functions for creating, reading, updating, and deleting files.

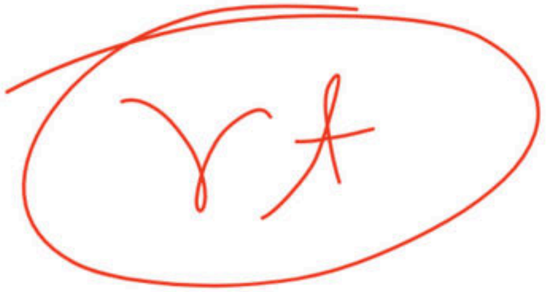
File Handling

The key function for working with files in Python is the open() function.

The open() function takes two parameters; filename, and mode.

There are four different methods (modes) for opening a file:

"r" - Read - Default value. Opens a file for reading, error if the file does not exist
"a" - Append - Opens a file for appending, creates the file if it does not exist
"w" - Write - Opens a file for writing, creates the file if it does not exist
"x" - Create - Creates the specified file, returns an error if the file exists
"t" - Text - Default value. Text mode
"b" - Binary - Binary mode (e.g. images)



Syntax

To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

Or

```
f = open("demofile.txt", "rt")
```

Open a File on the Server

Assume we have the following file, located in the same folder as Python:

newfile.txt

```
Hello! Welcome to newfile.txt
This file is for testing purposes.
Good Luck!
```

To open the file, use the built-in `open()` function.

The `open()` function returns a file object, which has a `read()` method for reading the content of the file:

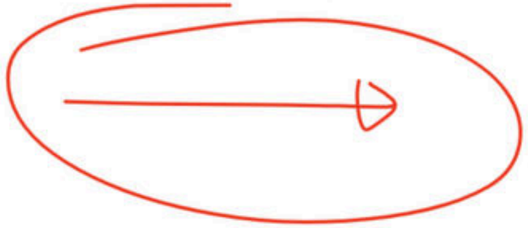
```
f = open("newfile.txt", "r")
print(f.read())
```

If the file is located in a different location

```
f = open("D:\\myfiles\welcome.txt", "r")
print(f.read())
```

By default the `read()` method returns the whole text, but you can also specify how many characters you want to return:

```
f = open("demofile.txt", "r")
print(f.read(5))
```



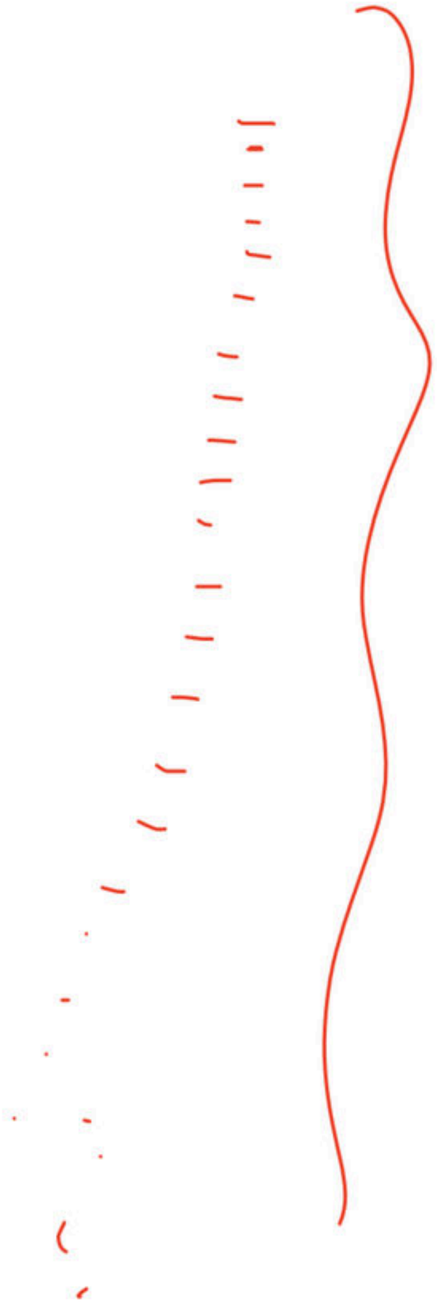
Read Lines

You can return one line by using the `readline()` method:

```
f = open("demofile.txt", "r")
print(f.readline())
```

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```



Close Files

It is a good practice to always close the file when you are done with it.

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

↳ open
↳ read.
↳ close

Write to an Existing File

To write to an existing file, you must add a parameter to the `open()` function:
`"a"` - Append - will append to the end of the file
`"w"` - Write - will overwrite any existing content

```
f = open("newfile2.txt", "a")
f.write("Extra content added to  the file!")
f.close()

#open and read the file after the appending:
f = open("newfile2.txt", "r")
print(f.read())
```

Python Delete File

Delete a File

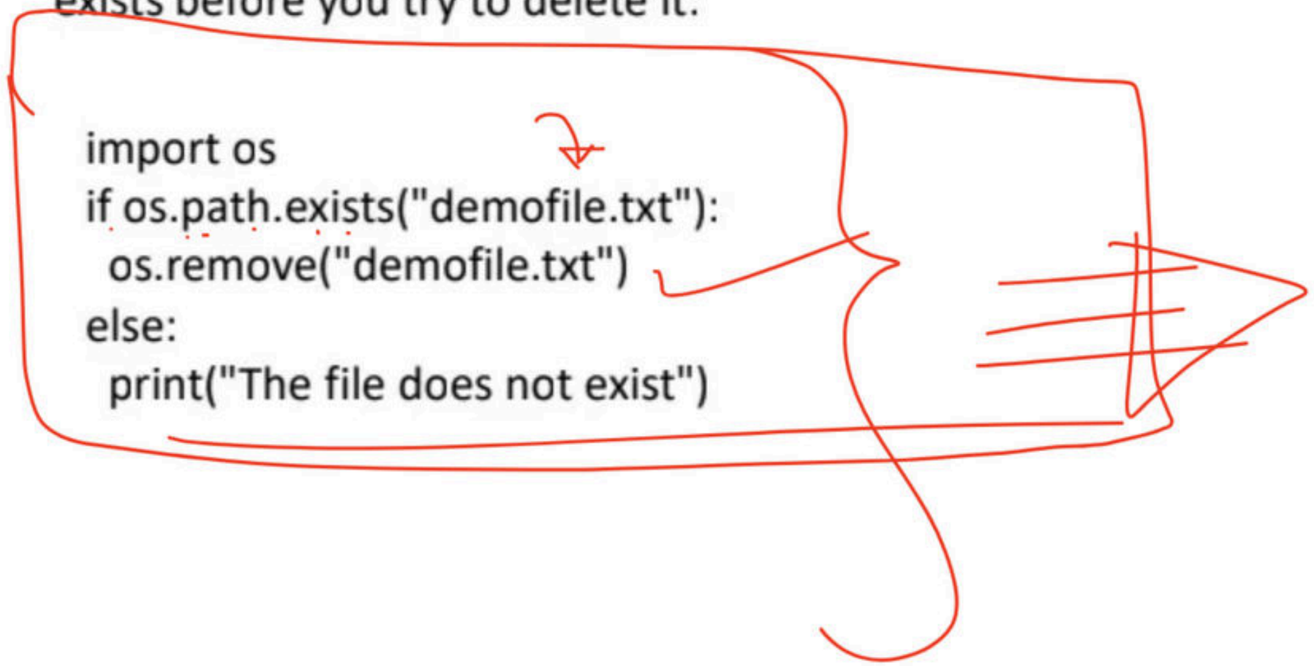
To delete a file, you must import the OS module, and run its `os.remove()` function:

```
import os
os.remove("newfile.txt")
```

Check if File exist:

To avoid getting an error, you might want to check if the file exists before you try to delete it:

```
import os
if os.path.exists("demofile.txt"):
  os.remove("demofile.txt")
else:
  print("The file does not exist")
```



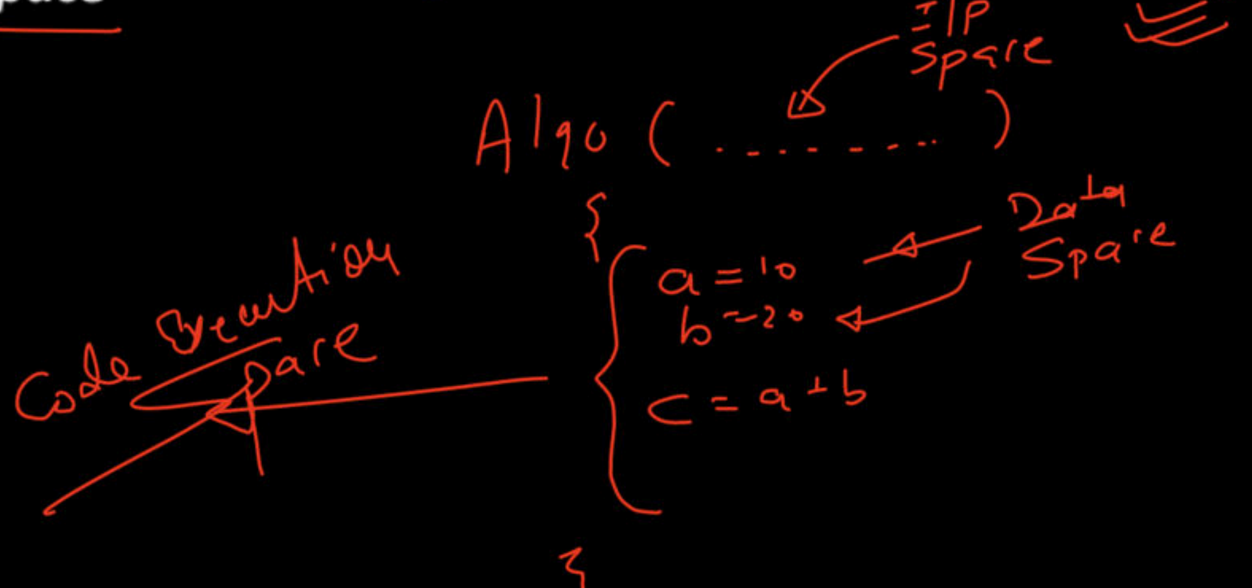
Delete Folder

To delete an entire folder, use the `os.rmdir()` method:

```
import os
os.rmdir("myfolder")
```

Time and Space Complexity of an Algorithm

- **Time Complexity:** The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input.
- The **space complexity** of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input.
  - ***Auxiliary Space*** [Data Space] + [Code Execution Space]
  - ***Input Space***

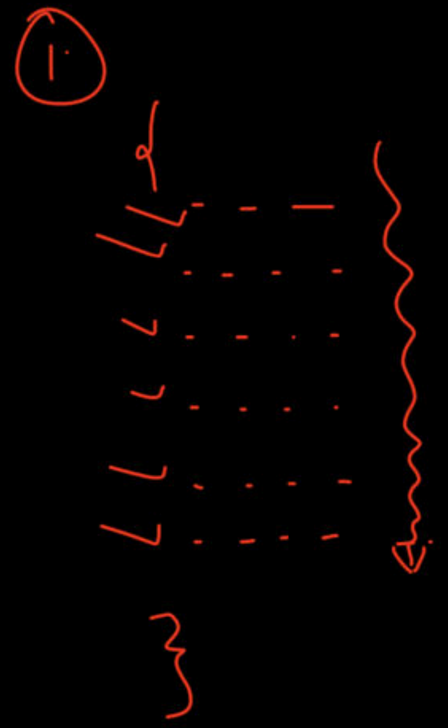


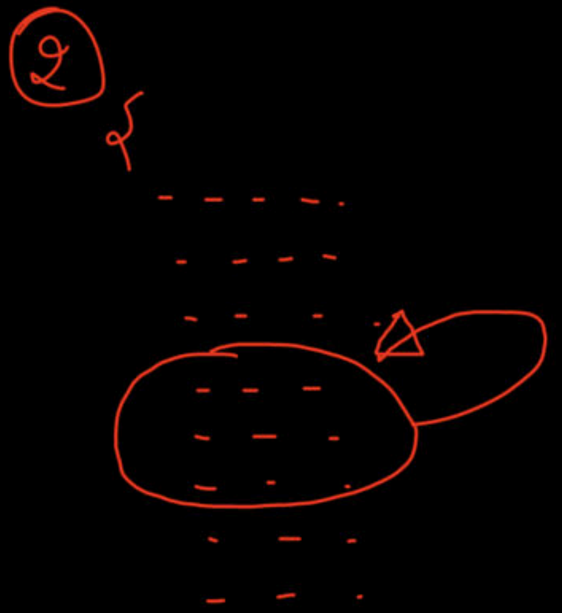
Space = Auxiliary Space
= Code Exe. Space + Data Space

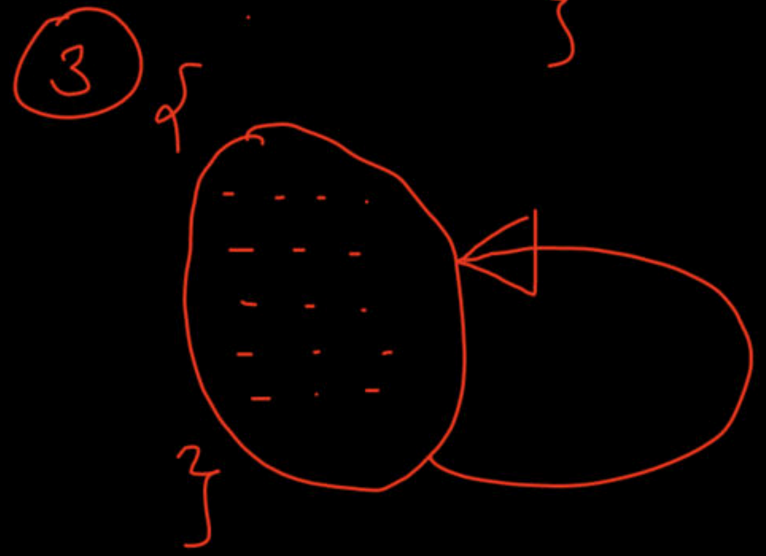
unacademy

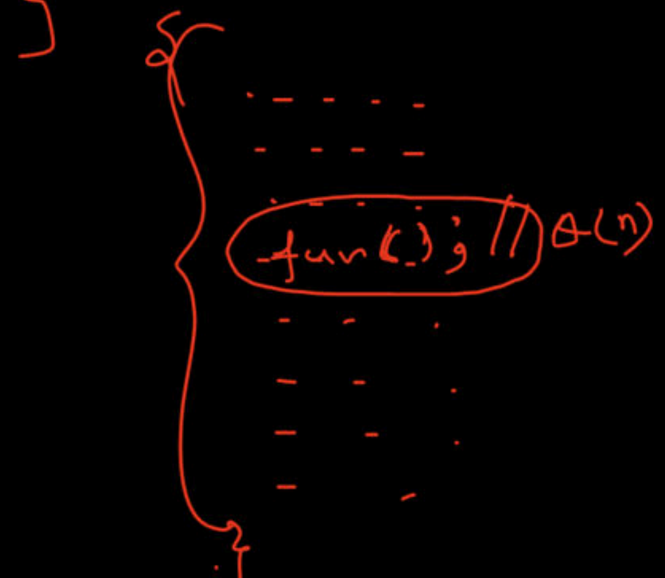
Types of Algorithm

1. Sequential Algorithm
2. Iterative Algorithm
3. Recursive Algorithm



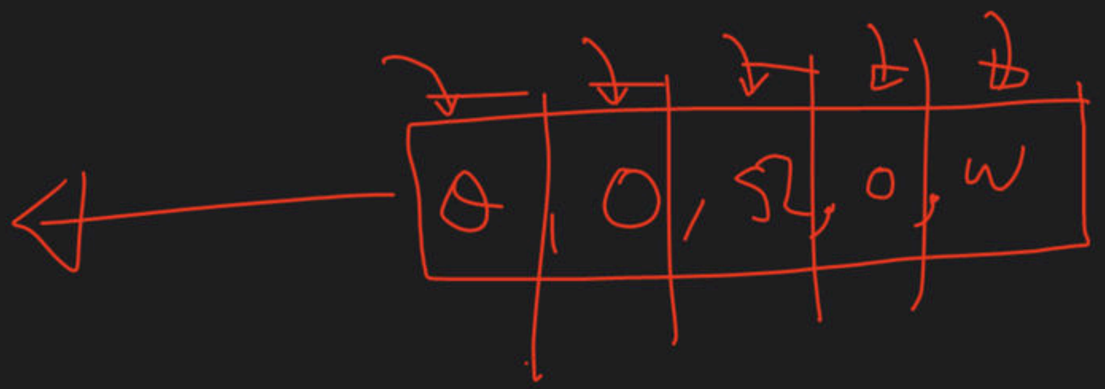






```
a = 10
b = 20
c = a + b
Print(c)
```

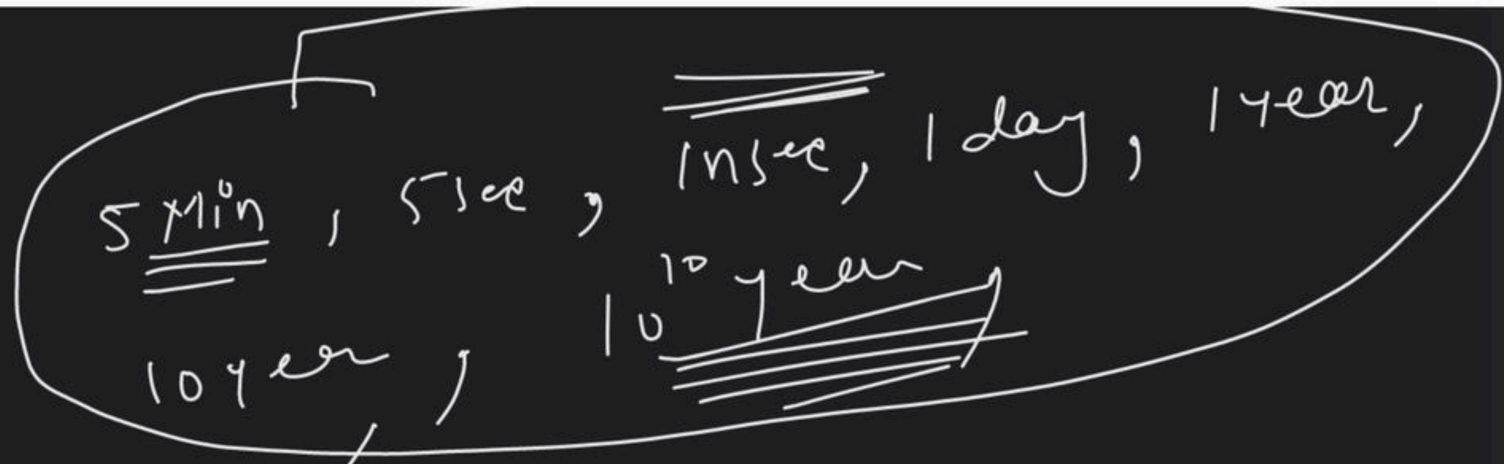
→ Sequential code ⇒ $\Theta(1)$



Asymptotic Notation (Mathematical Notation)

↳ use to represent Complexity of Algorithm

- (1) Big oh ("O")
- (2) omega Notation ("Ω")
- (3) Theta Notation ("θ")
- (4) Little oh ("o")
- (5) Little omega ("ω")

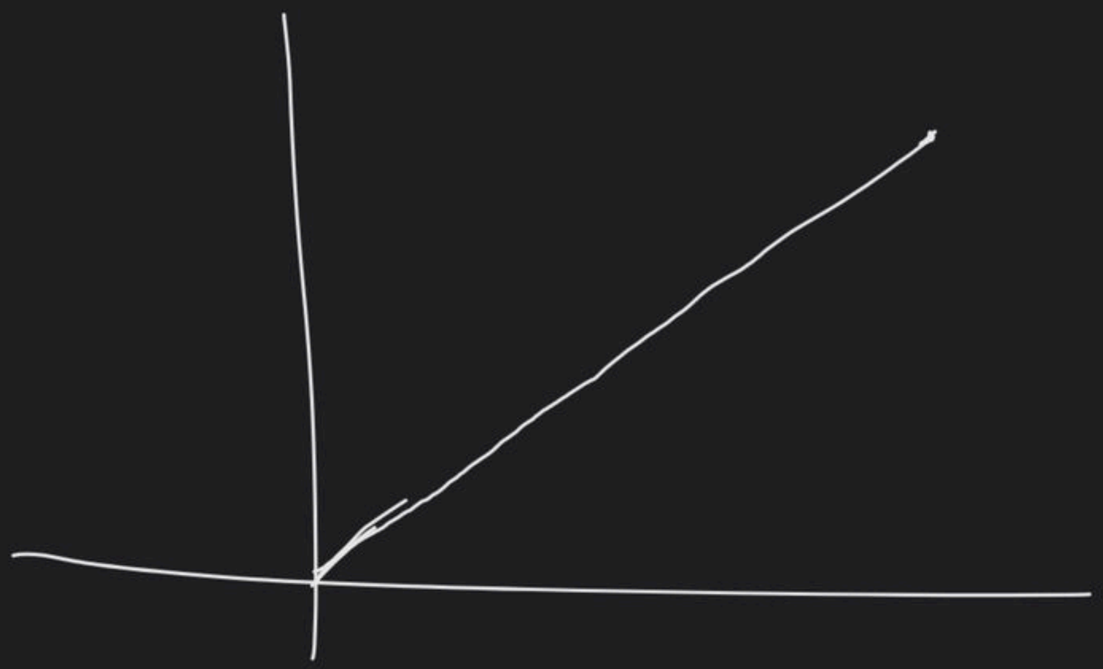


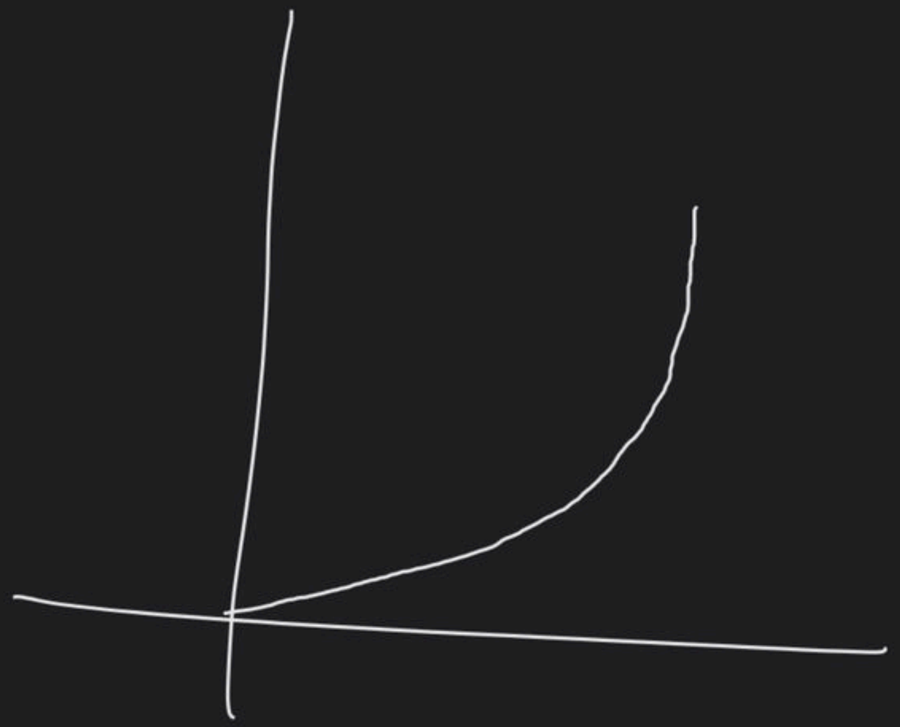
Constant Complexity

C or K or 1

unacademy

n^2





$$\{1, n, n^2, n^3, \ldots\ldots e^n, 2^n, 3^n, n^n \ldots\ldots n!\}$$

$$\{\log n, \log^2 n \ldots\ldots\ldots\ldots\}$$

(1)

(2)

1. Big-oh Notation :→ (upper bound)

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

Algorithm $= O(g(n))$

→ when growth rate of $g(n)$ is greater than or Equal to $f(n)$.

$f(n) = n$

$g(n) = n^2$

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

n	n^2
1	1
2	4
3	9

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

unacademy

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

$$f(n) \leq c * g(n), \forall n \geq n_0$$

+ve Const^n

$f(n) = n+4$		$g(n) = n^2 * c=1$
$n=0$	$\Rightarrow 4$	0
$n=1$	$\Rightarrow 5$	1
$n=2$	$\Rightarrow 6$	4
$n=3$	$\Rightarrow 7$	9

$n_0 =$

$f(n) < g(n)$

$f(n) = n + 4$

$g(n) = n^2 * \frac{1}{100} * 100$

$n = 3 \Rightarrow 7$ $\quad > \quad \frac{9}{100}$

$n = 100 \Rightarrow 104$ $\quad \Rightarrow \quad \frac{100 \times 100}{100}$

$n = 1000$

$\Rightarrow 1004$ $\quad < \quad \frac{1000 \times 1000}{100} \Rightarrow 10000$

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

$g(n) = 2^n$

$h(n) \Rightarrow n!$

Arrange function in their growth rate in ascending order.

$n \Rightarrow 10$

$f(n) = 10 \times 10 \Rightarrow 100$

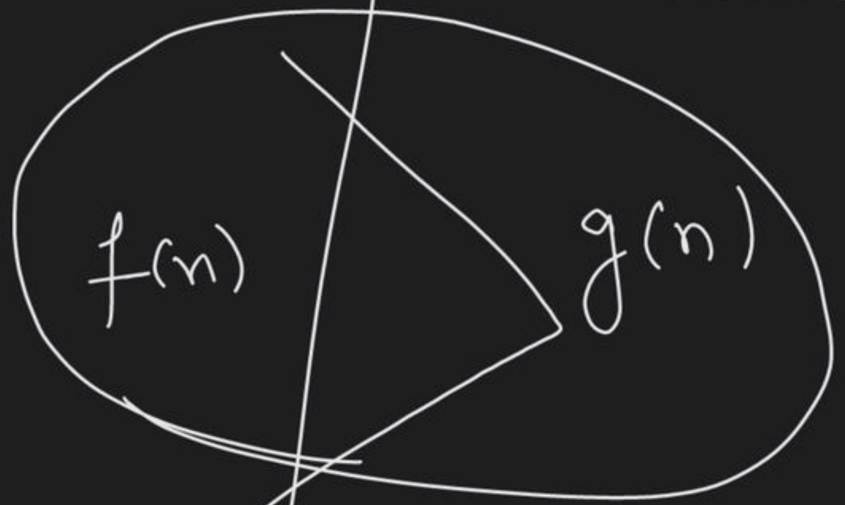
$g(n) = 2 \times 2 \times 2 \cdots \cdots 2 \Rightarrow 1024$

$h(n) = 10 \times 9 \times 8 \times \cdots 1 \Rightarrow$ Large

$$f(n) < g(n) < h(n)$$

(1) $f(n) = n + 4$ $\quad c = 1$ $\quad g(n) = n + 2$

$f(n) = n+4$	$g(n) = (n+2) * 1$	$g(n) = (n+2) * \frac{1}{2}$	$g(n)$ $(n+4)$ $*2$



$n = 10^{10}$ $\quad f(n) = 10^{10} + 4$

$g(n) = \frac{10^{10} + 2}{2}$

$f(n) > g(n)$

$f(n) < g(n)$

if $c = 1$. $f(n) > g(n)$

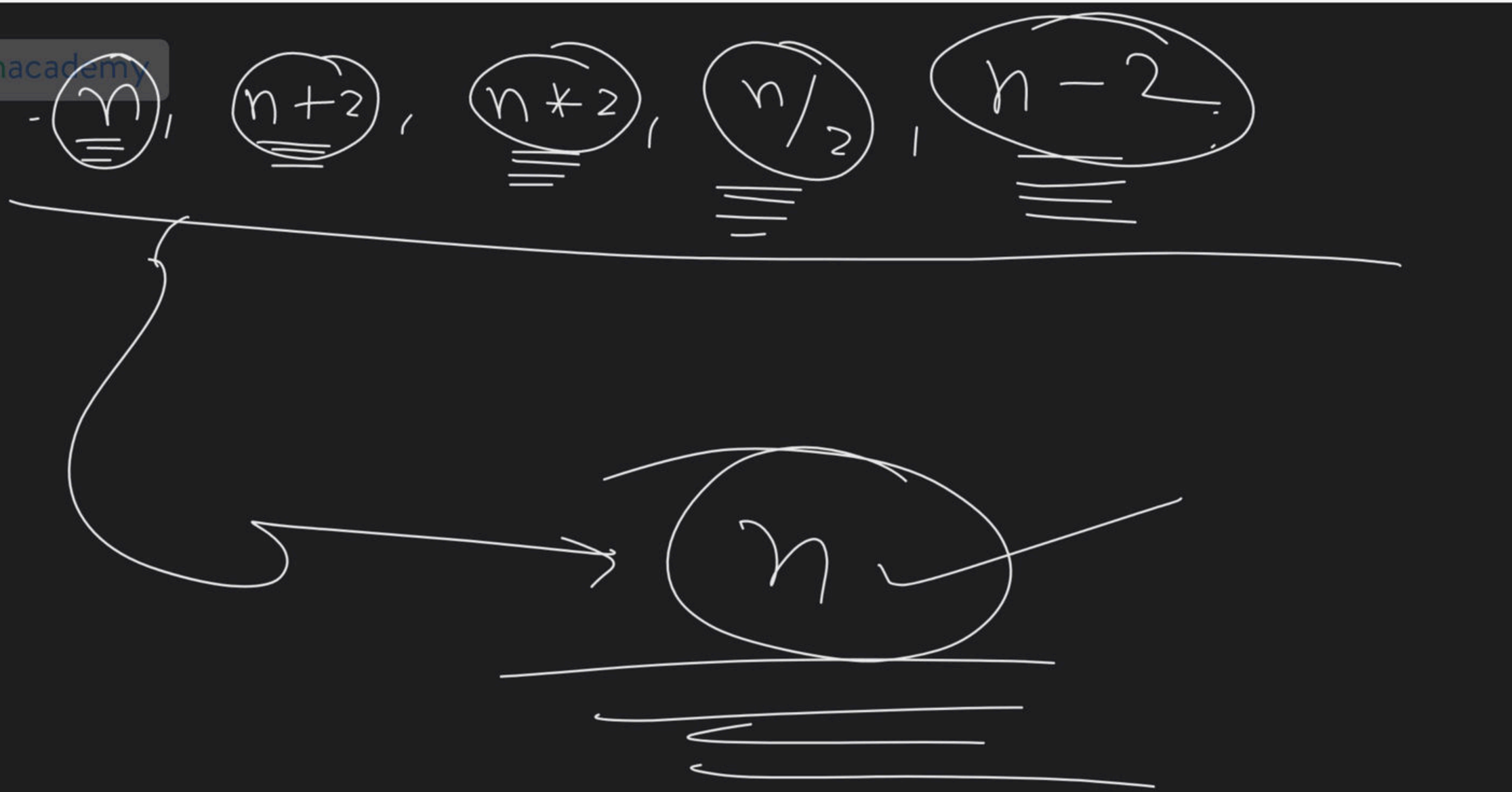
$c = 2$ $f(n) < g(n)$

$n+2$

$n+4$

n

$f(n) \equiv g(n)$



① $f(n) = 2^{n+1} \Rightarrow 2 \cdot 2^n \Rightarrow 2^n$

② $g(n) = 2^{2n} \Rightarrow (4)^n \Rightarrow 4^n$

③ $h(n) = 2^{n-1} \Rightarrow \frac{2^n}{2} \Rightarrow 2^n$

growth rate

$$f = h < g$$

(1)

$$\begin{cases} f(n) = n + 4 \\ g(n) = n + 2 \end{cases}$$

Same

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

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

(c) ~~Both~~

(d) None

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

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

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

(h) $g(n) = \Theta(f(n))$

Omega Notation (Ω) ⇒ Lower bound of an Algorithm.

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

↳ ⟶ growth rate of $g(n)$ is less than or equal to $f(n)$.

$f(n) \geq c * g(n) \quad \forall n \geq n_0$

① $f(n) = n + 4$ ∴
$g(n) = n^2$ ∴

$f(n) < g(n)$

(a) $g(n) = \Omega(f(n))$ → growth less

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

(c) Both

(d) None

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

III. Theta Notation :- (Equal bound)

$f(n) = \Theta(g(n))$ → when growth of $g(n)$ is Equal to the $f(n)$.

$$c_2 * g(n) \leq f(n) \leq c_1 * g(n)$$ → $\forall n \geq n_0$

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

(a) $f(n) = \Theta(g(n))$
(b) $g(n) = \Theta(f(n))$
(c) Both
(d) None

$f(n) = n + 4$

$g(n) = n + 2$

(a) $f(n)$

little oh Notation ("o") :- Strict upper bound

$f(n) = o(g(n))$ ⟶ growth rate of $g(n)$ is strictly greater than $f(n)$.

$$f(n) < c * g(n), \forall n \geq n_0$$

$f(n) = n + 4$ } → Same
$g(n) = n + 2$ }

(a) $f(n) = o(g(n))$

(b) $g(n) = o(f(n))$

(c) Both

(d) None

Little omega Notation ("ω") :- Strict lower bound.

$f(n) = \omega(g(n))$ → growth rate of $f(n)$ is strictly less than $f(n)$.

$f(n) > c * g(n), \forall n \geq n_0$

$f(n) = n^2 \log n$

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

$h(n) = n \log n$

$n^2 \log n$ $\quad$ $2n \log n$ $\quad$ $n \log n$

$n * n \log n$ $\quad$ Same

$f > g = h$

~~[1]~~ $f(n) = O(g(n))$

~~(2)~~ $f(n) = O(h(n))$

~~(3)~~ $g(n) = O(f(n))$

~~(4)~~ $g(n) = O(h(n))$

~~(5)~~ $g(n) = \Omega(h(n))$

~~6~~ $g(n) = \Theta(f(n))$

~~7~~ $h(n) = \Theta(f(n))$

~~8~~ $f(n) = \Omega(h(n))$

1. $\log(m \times n) = \log m + \log n$
2. $\log m^n = n \log m$
3. $\log_a b = \frac{1}{\log_b a}$
4. $a^{\log_a b} = b$
5. $\log\left(\frac{m}{n}\right) = \log m - \log n$

$f(n) = 3n^{\sqrt{n}}$ $\longrightarrow$ $3n^{\sqrt{n}}$

$\sqrt{n}\log_2 n$ $\qquad$ $\log_2 n^{\sqrt{n}}$

$g(n) = 2 \Rightarrow 2$ $\Rightarrow$ $\sqrt{n}$ $\Rightarrow$ n

Same

$h(n) = n^n$

$$f = g < h$$

$g_1(n) = n^{3/4} \rightarrow n^{0.75}$

7, 1, 8 = 9 = 11, 10, 4, 3, 5, 6, 2

lowest ⟹

$g_2(n) = 2^{n^2}$

$g_3(n) = 2^n$

$g_4(n) = 4^{\sqrt{n}} \rightarrow 2^{2^{\sqrt{n}}}$

$g_5(n) = n!$

$g_6(n) = n^n$

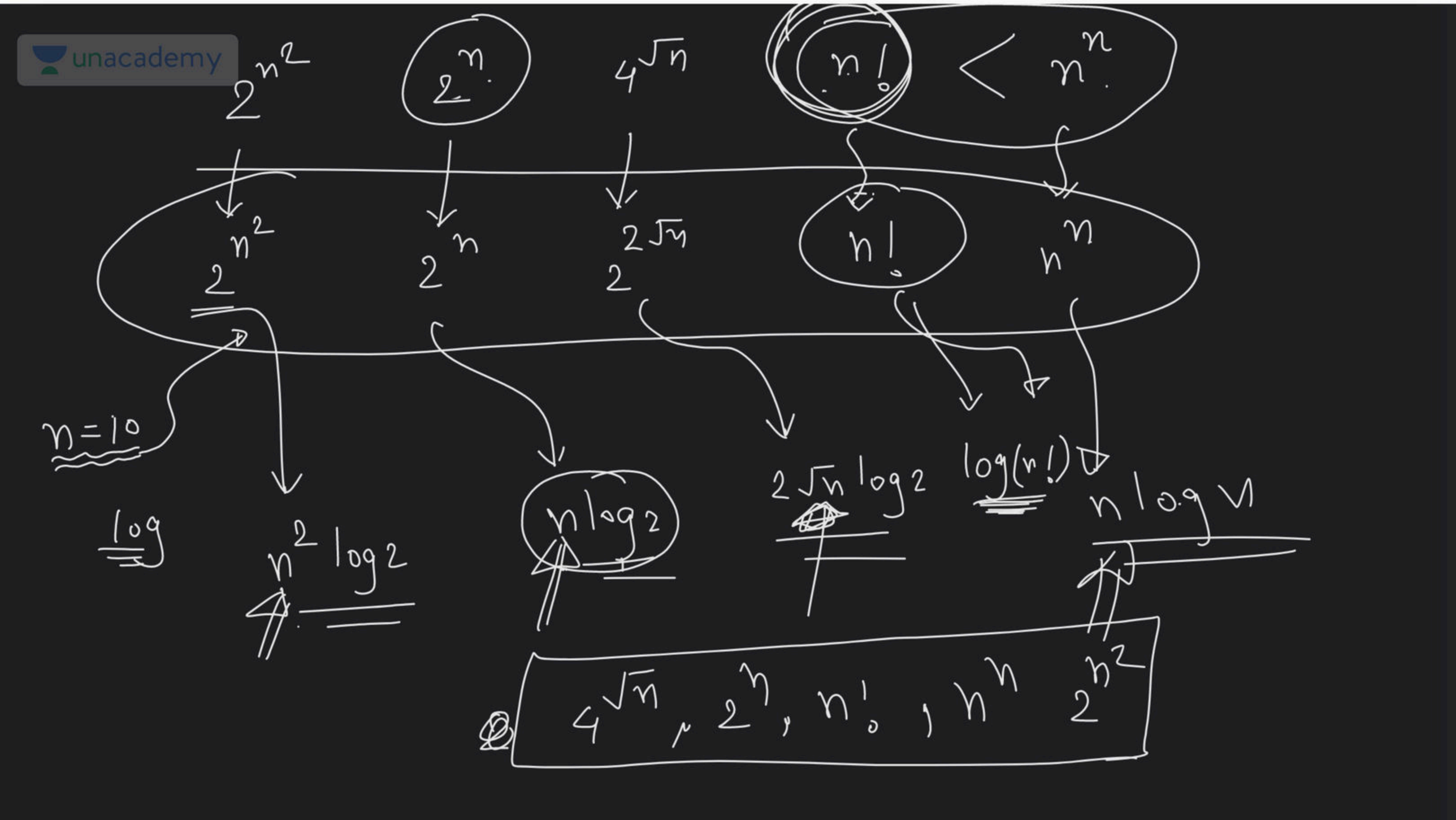
$g_7(n) = \sqrt{n} \rightarrow n^{0.5}$

$g_8(n) = n \log \sqrt{n} \rightarrow n \log n^{1/2} \Rightarrow \frac{1}{2} n \log n \Rightarrow n \log n$

$g_9(n) = n \log n \rightarrow n \log n$

$g_{10}(n) = n^2 \log n^2$

$g_{11}(n) = n! \log n^2 \rightarrow n^{\log n}$


unacademy
2^{n^2}
2^n
$4^{\sqrt{n}}$
$n!$
$< n^n$
n^2
n
$2\sqrt{n}$
$n!$
n^n
$n=10$
log
$n^2 \log 2$
$n \log 2$
$2\sqrt{n} \log 2$
$\log(n!)$
$n \log n$
$4^{\sqrt{n}}$, 2^n, $n!$, n^n 2^{n^2}

unacademy

Master Theorem For Subtract Recurrences

- Let T(n) be a function defined on positive n as shown below:

$$T(n) \leq \begin{cases} c, & \text{if } n \leq 1, \\ aT(n-b) + f(n), & n > 1, \end{cases},$$

for some constants c, a>0, b>0, k>=0 and function f(n). If f(n) is $O(n^k)$, then

1. If a<1 then $T(n) = O(n^k)$
2. If a=1 then $T(n) = O(n^{k+1})$
3. if a>1 then $T(n) = O(n^k a^{n/b})$

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

(2) $T(n) = 0.5T(n-1) + n$

(3) $T(n) = 2T(n-2) + 1$

(4) $T(n) = 2T(n-1) + n$

(5) $T(n) = 2T(n-2) + n$

(6) $T(n) = 3T(n-1) + n^2$

(7) $T(n) = 4T(n-2) + n^2$

unacademy

Master’s Method for Divide and Conquer

unacademy

(1) $T(n) = T(n/2) + 1$

(2) $T(n) = 2T(n/2) + 1$

(3) $T(n) = 4T(n/2) + n$

(4) $T(n) = T(n/2) + logn$

(5) $T(n) = 2T(n/2) + nlogn$

unacademy

(1) $T(n) = 2T(n/4) + \sqrt{n}logn$

(2) $T(n) = 8T(n/2) + n^2$

(3) $T(n) = 2T(n/8) + n$

(4) $T(n) = T(n/2) + n^2$

(5) $T(n) = 4T(n/2) + n^2$

Extended Master's Method 1

- $T(n) = aT(n/b) + f(n)\,(logn)^k$
- $Here\ f(n)\ is\ polynomial\ function$
- $a, b, k\ are\ positive\ constant\ a > 0, b > 1, k = -1$

- Calculate $n^{\log_b a}$

Case 1: If $f(n) < n^{\log_b a}$ then T(n) = $\boldsymbol{\theta}(n^{\log_b a})$

Case 2: If $f(n) > n^{\log_b a}$ then T(n) = $\boldsymbol{\theta}(f(n))$

Case 2: If $f(n) = n^{\log_b a}$ then T(n) = $\boldsymbol{\theta}(f(n)loglogn)$

Extended Master's Method 2

- $T(n) = aT(n/b) + f(n)\,(logn)^k$
- $Here\ f(n)\ is\ polynomial\ function$
- $a, b, k\ are\ positive\ constant\ a > 0, b > 1, k < -1$

- Calculate $n^{\log_b a}$

Case 1: If $f(n) < n^{\log_b a}$ then T(n) = $\boldsymbol{\theta}(n^{\log_b a})$

Case 2: If $f(n) > n^{\log_b a}$ then T(n) = $\boldsymbol{\theta}(f(n))$

Case 2: If $f(n) = n^{\log_b a}$ then T(n) = $\boldsymbol{\theta}(n^{\log_b a})$

(1) $T(n) = T(n/2) + (\log n)^{-1}$

(2) $T(n) = 2T(n/2) + (\log n)^{-2}$

(3) $T(n) = 4T(n/2) + \frac{n}{\log n}$

(4) $T(n) = T(n/2) + \frac{1}{(\log n)^2}$

(5) $T(n) = 2T(n/2) + \frac{n}{\log n}$

(6) $T(n) = 2T(n/4) + \frac{\sqrt{n}}{(\log n)^3}$

(7) $T(n) = 8T(n/2) + \frac{n^2}{\log n}$

- Recursive Relation
  - Substitution Method
  - Trees Method

How to Calculate Space Complexity of an Algorithm

unacademy

Space complexity *is the amount of memory used by the algorithm to execute and produce the result.*

- ***Auxiliary Space***
- ***Input Space***

unacademy

Space Complexity

- How to calculate Space complexity of
  - Sequential Algorithm
  - Iterative Algorithm
  - Recursive Algorithm

Algorithms

- Sequential Algorithm

Algorithms

- Iterative Algorithm

```
def initialize_array(A, n):
    for i in range(n):
        A[i] = 0

# Example usage:
n = 10  # Replace with the desired array size
A = [0] * n  # Initialize an array of size n with zeros
initialize_array(A, n)
print(A)  # You can print the array to verify the initialization
```

```python
# Example array A
A = [1, 2, 3, 4, 5]  # Replace with your array

n = len(A)
B = [0] * n  # Initialize array B with zeros
C = [0] * n  # Initialize array C with zeros

for i in range(n):
    B[i] = A[i]
    C[i] = A[i]

# Printing the results
print("Array B:", B)
print("Array C:", C)
```

```python
# Example array A
A = [1, 2, 3, 4, 5]  # Replace with your array
n = len(A)

# Create an empty 2D array C with dimensions n x n
C = [[0 for _ in range(n)] for _ in range(n)]

# Copy the elements from A to C
for i in range(n):
    for j in range(n):
        C[i][j] = A[i]

# Printing the resulting 2D array C
for row in C:
    print(row)
```

Algorithms

- Recursive Algorithm

unacademy

```
def f(n):
    if n / 2 <= 1:
        return n
    f(n / 2)
    f(n / 2)

# Example usage
result = f(8)  # Replace 8 with the desired value
print(result)
```

Algo 1:

```python
def fun(n):
    if n <= 1:
        return n
    else:
        return fun(n - 1) + fun(n - 1)

# Example usage:
result = fun(5)  # Replace 5 with the desired value
print(result)
```

Algo 2:

```python
def fun(n):
    if n <= 1:
        return n
    else:
        return 2 * fun(n - 1)

# Example usage:
result = fun(5)  # Replace 5 with the desired value
print(result)
```

unacademy

```
int fun( int n)
{
if(n<=1)
return n;
else
{
fun(n-1);
for(i=1;i<=n; i=i+10);
return i;
}
}
```

```python
def fun(n):
    if n <= 1:
        return n
    else:
        fun(n - 1)
        i = 1
        while i <= n:
            i += 10
        return i

# Example usage:
result = fun(5)  # Replace 5 with the desired value
print(result)
```

unacademy

```
int fun( int n)
{
if(n<=1)
return n;
else
{
fun(n/2);
for(i=1;i<=n; i=i+10);
return i;
}
}
```

```python
def fun(n):
    if n <= 1:
        return n
    else:
        fun(n // 2)  # Note: Use integer division (//) to match C++ behavior
        i = 1
        while i <= n:
            i += 10
        return i

# Example usage:
result = fun(20)  # Replace 20 with the desired value
print(result)
```

What is Space Complexity?

(a) $\theta(n)$

(b) $\theta(\log n)$

(c) $\theta(n \log n)$

(d) None