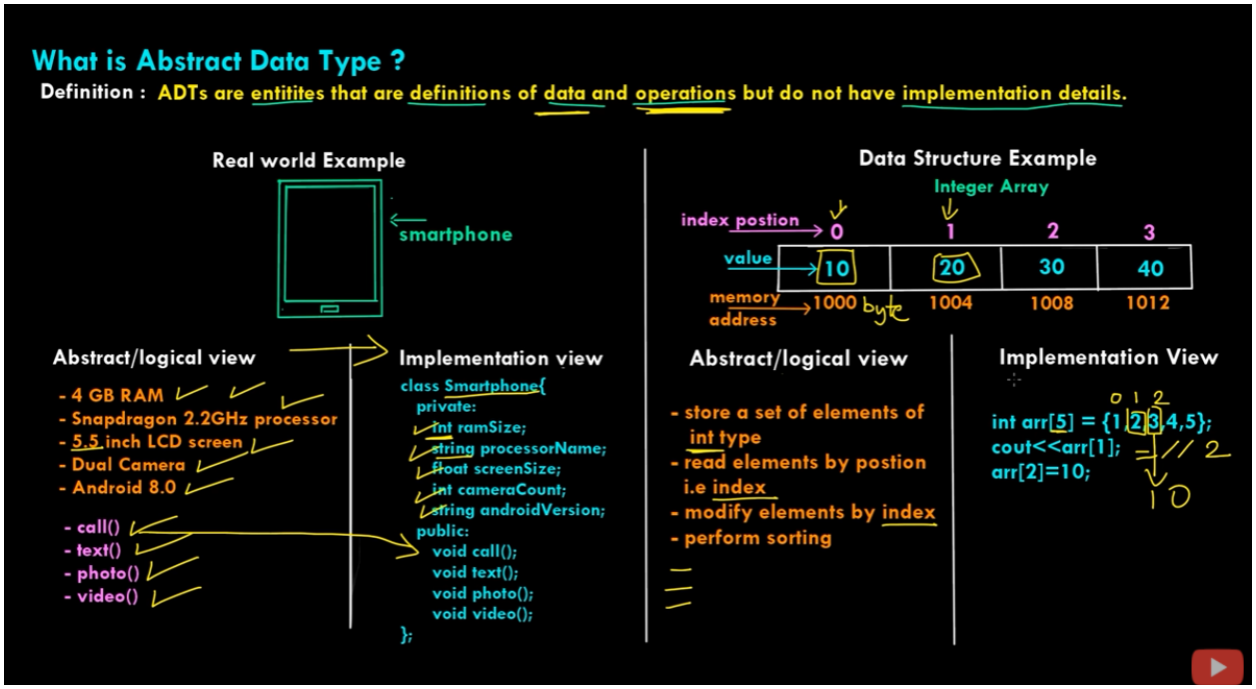
Data Structure

Data Structure is a particular way of storing and organizing the information in a computer so that it can be retrieved and used most productively.

Abstract Data Structures

ADT are entities that have definitions of data and operations but do not have the implementation details.



Algorithm

What is an Algorithm ?

Dictionary Definition : A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

Formal Definition : An algorithm is a finite set of instructions that are carried in a specific order to perform specific task.

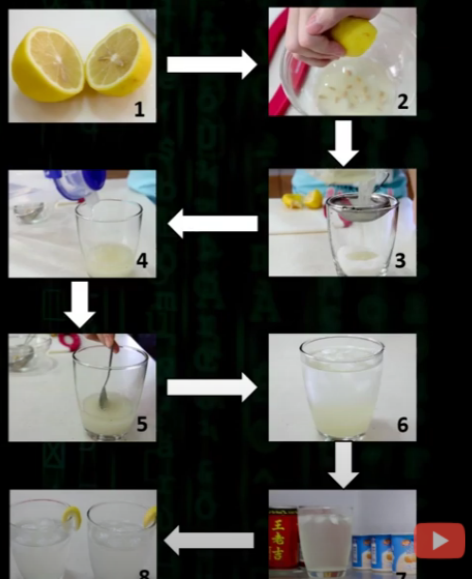
Algorithms typically have the following characteristics –

- Inputs : 0 or more input values.
- Outputs : 1 or more than 1 output.
- Unambiguity : clear and simple instructions.
- Finiteness : Limited number of instructions.
- Effectiveness : Each instruction has an impact on the overall process.

Real World example of an Algorithm –

Algorithm(aka process) to make a lemonade –

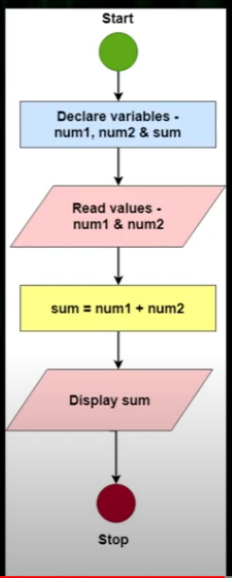
1. Cut your lemon in half.
2. Squeeze all the juice out of it that you can.
3. Pour your juice into a container with 1/4 cup (2 oz) sugar.
4. Add a very small amount of water to your container.
5. Stir your solution until sugar dissolves.
6. Fill up container with water and add ice.
7. Put your lemonade in the fridge for five minutes.
8. Serve and enjoy!



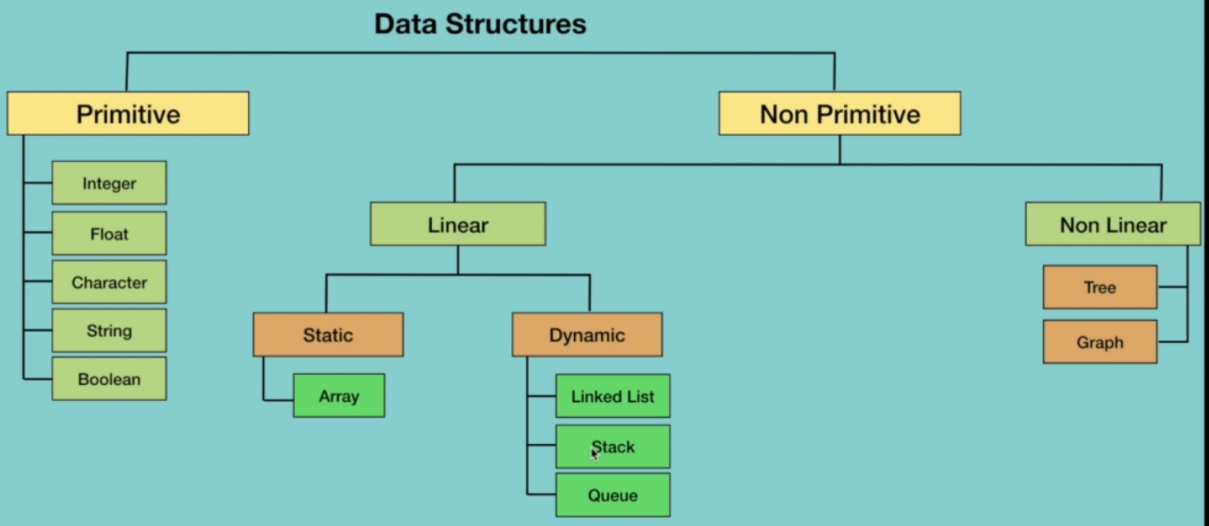
Example of an Algorithm in Programming –

Write an algorithm to add two numbers entered by user. –

1. Step 1: Start
2. Step 2: Declare variables num1, num2 and sum.
3. Step 3: Read values num1 and num2.
4. Step 4: Add num1 and num2 and assign the result to sum.(sum←num1+num2)
5. Step 5: Display sum
6. Step 6: Stop



Types of Data Structures



Types of Algorithms

- Simple recursive algorithms
- Divide and conquer algorithms
- Dynamic programming algorithms
- Greedy algorithms
- Brute force algorithms
- Randomized algorithms

Types of Algorithms

Simple recursive algorithms

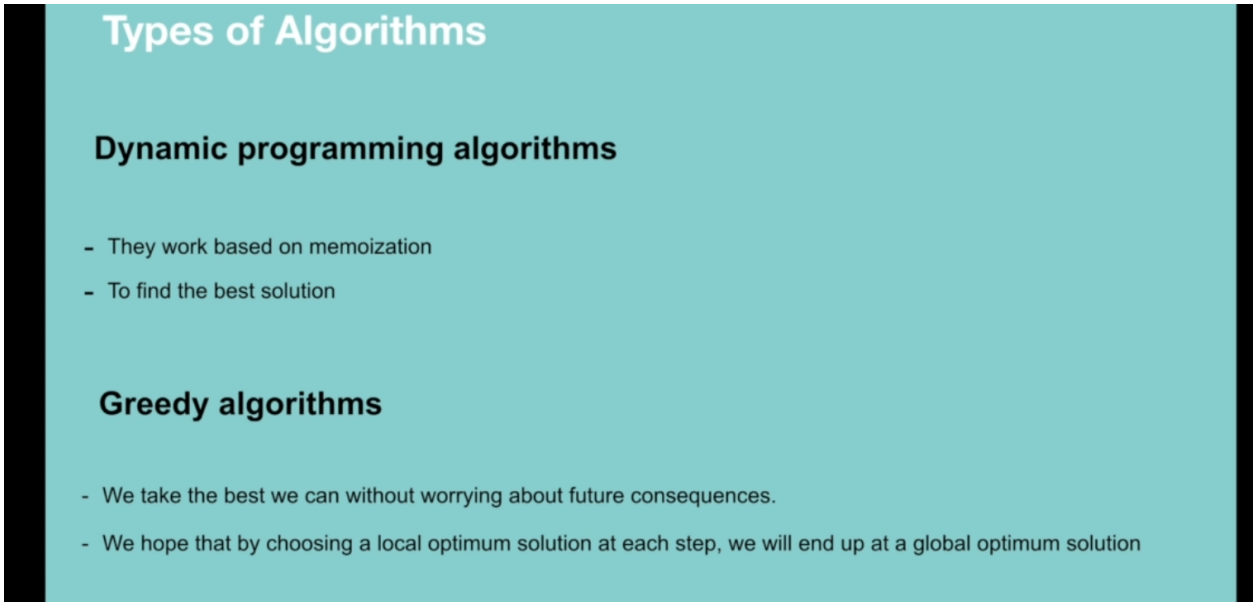
```
Algorithm Sum(A, n)
    if n=1
       return A[0]
    s = Sum(A, n-1)    /* recurse on all but last */
    s = s + A[n-1]     /* add last element */
return s
```

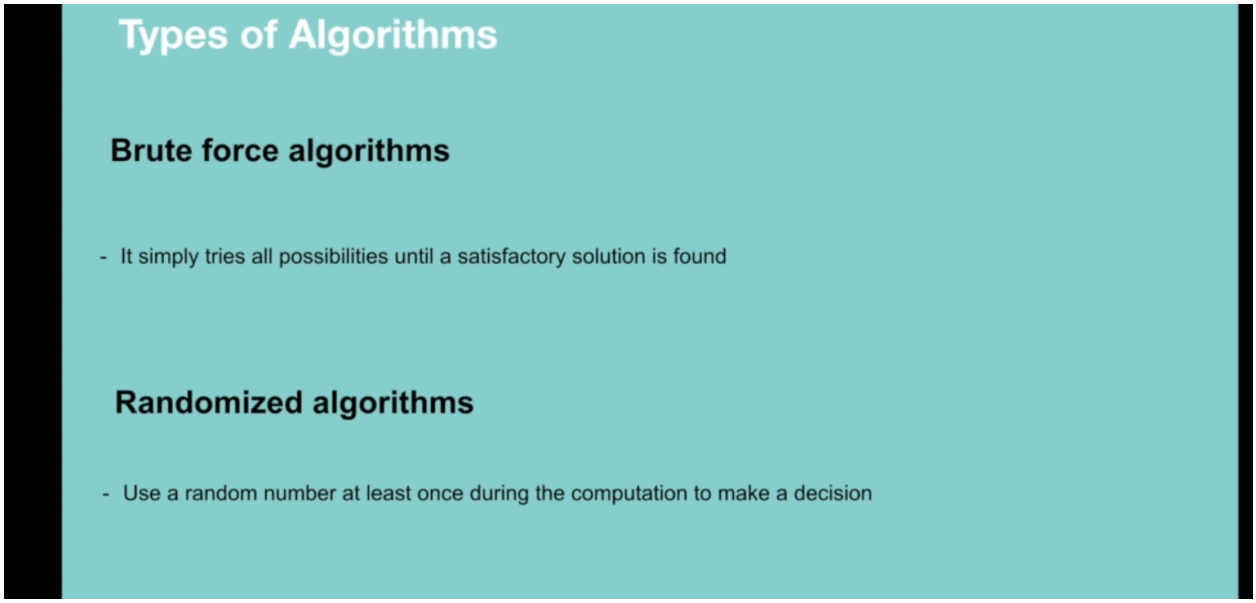
Types of Algorithms

Divide and conquer algorithms

- Divide the problem into smaller subproblems of the same type, and solve these subproblems recursively
- Combine the solutions to the subproblems into a solution to the original problem

Examples: Quick sort and merge sort





Stack Data Structure

What is Stack Data Structure ?

Definition : *Stack is a linear data structure which operates in a LIFO(Last In First Out) or FILO (First In Last Out) pattern.*

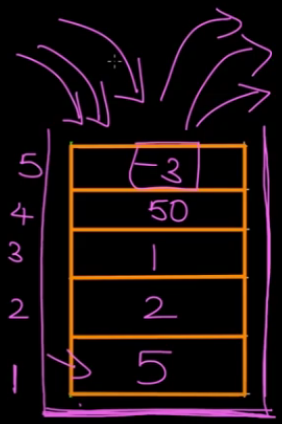
- It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.
- Stack is an abstract data type with a bounded (predefined) capacity.
- It is a simple data structure that allows adding and removing elements in a particular order.
- The order may be LIFO(Last In First Out) or FILO(First In Last Out).

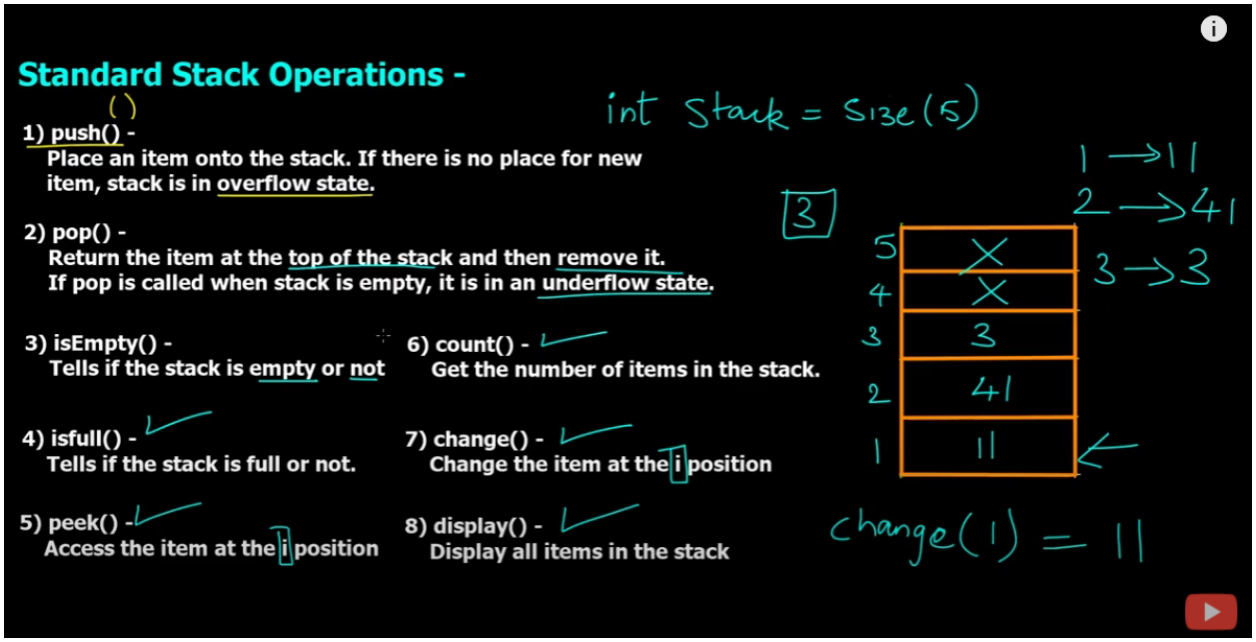
Working of Stack -

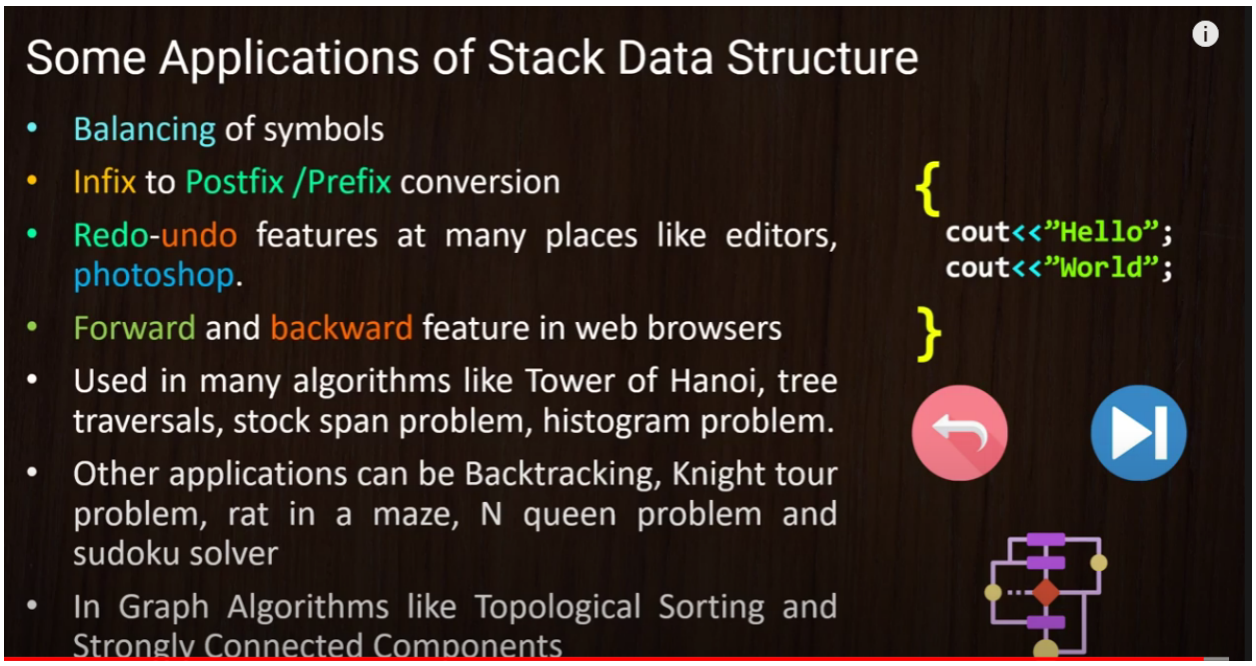
Stack Data Structure operates in a LIFO(Last In First Out) pattern or FILO (First In Last Out) pattern.

>> Items are added on top of the stack. This is know as PUSH operation

>> Items are removed from top of the stack. This is know as POP operation



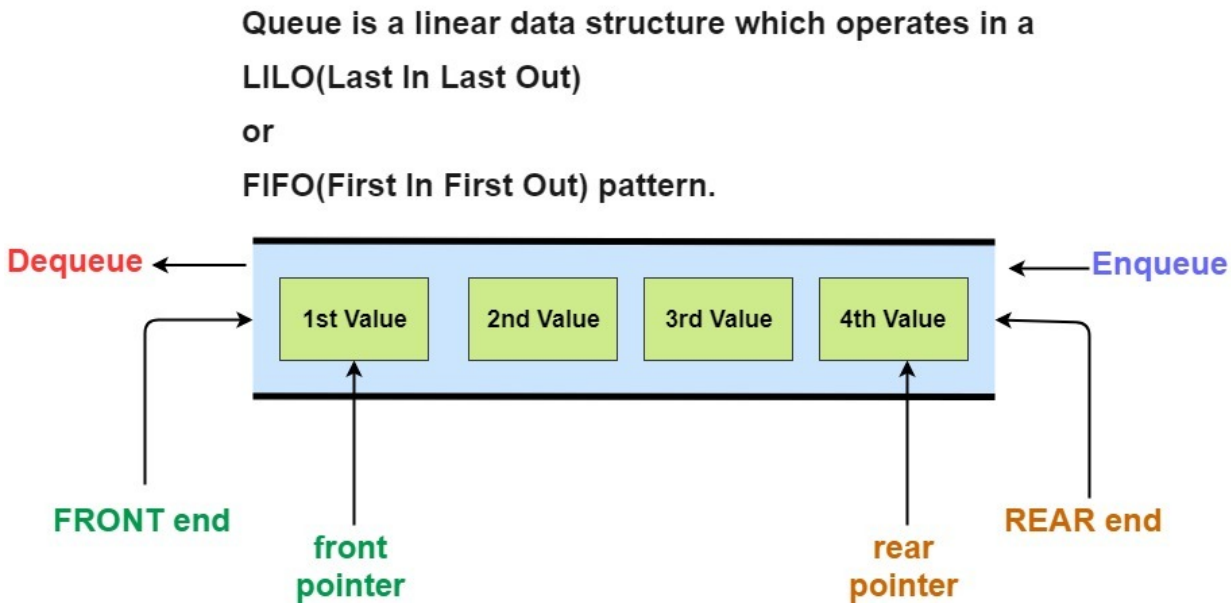




Queue is a linear data structure which operates in a First IN First OUT or Last IN Last OUT.

- *It is named queue as it behaves like a real-world queue, for example – queue(line) of cars in a single lane, queue of people waiting at food counter etc.*
- *Queue is an abstract data type with a bounded (predefined) capacity.*

- *It is a simple data structure that allows adding and removing elements in a particular order.*
- *The order is FIFO(First IN First OUT) or LILO(Last In Last Out).*



Standard Queue Operations –

- **Enqueue() – Add item to the queue from the REAR.**
- **Dequeue() – Remove an item from the queue from the FRONT.**
- **isFull() – Check if the queue is full or not.**
- **isEmpty() – Check if the queue is empty or not.**
- **count() – Get the number of items in the queue.**

Some types of Queue (We will discuss them in detail in other articles)-

- **Simple Queue**
- **Circular Queue**
- **Priority Queue**

Some Applications of Queue Data Structure –

Queue is used when things have to be processed in First In First Out order. Like –

- **CPU scheduling, Disk Scheduling.**
- **Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.**
- **In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.**
- **When data is transferred asynchronously between two processes. Queue is used for synchronization.**

BIG O

Big O notation is used in Computer Science to describe the performance or complexity of an algorithm. Big O is used to describe the execution time required or the space used (e.g. in memory or on disk) by an algorithm.

To measure the efficiency of an algorithm, we need to consider two things:
Time Complexity: How much time does it take to run completely?
Space Complexity: How much extra space does it require in the process?

How to define good code?

Readable

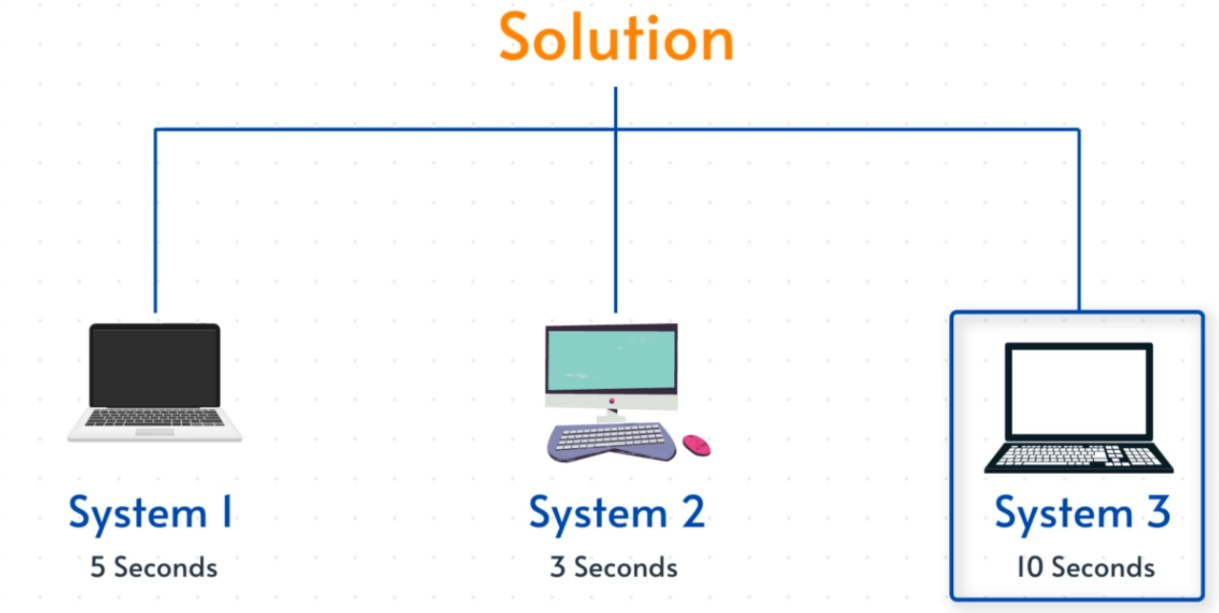
Time

Space

Scalable

Write a program to calculate the first 100 natural numbers

This program can take different amounts of time in different systems

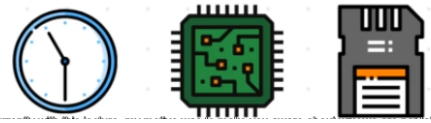


So time difference between starting of execution and programing ending is not a good solution because it is hardware dependent and hence that's why use Big o as the parameter to judge the algorithm

Why we need Big O?

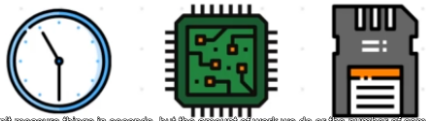
We require BIG O to give a performance rating to our program for time and space.

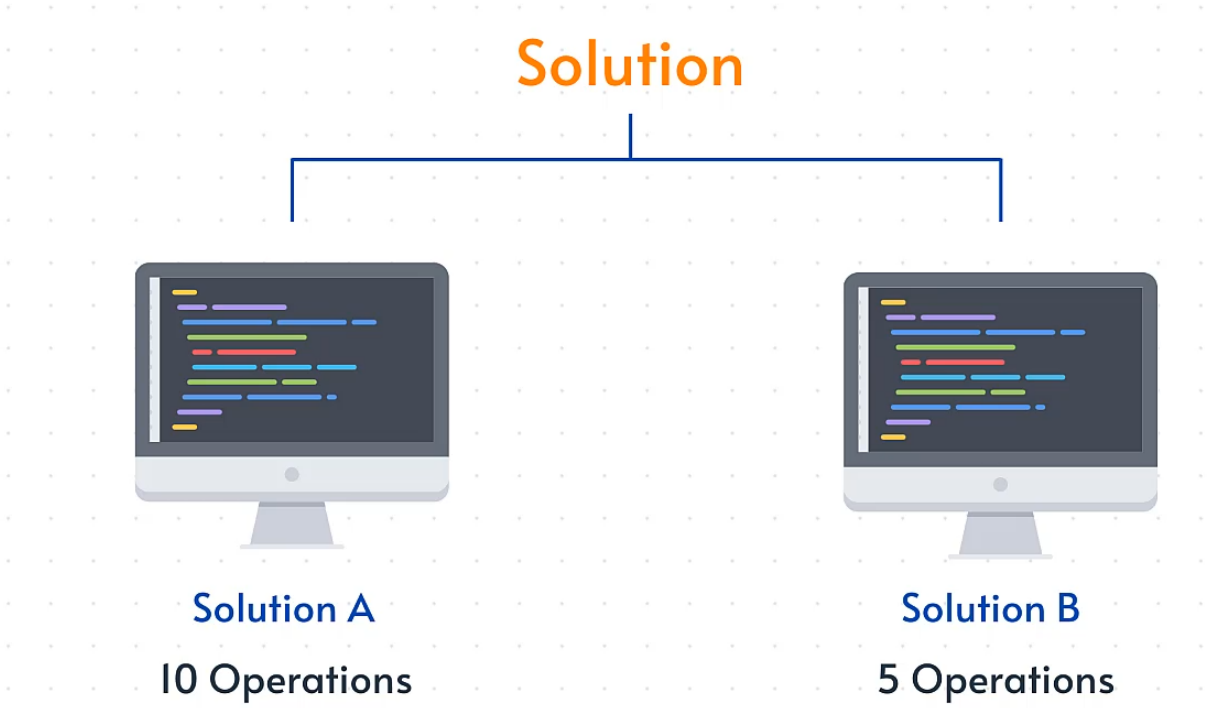
We cannot depend on our system clock time as well as hardware for each program.



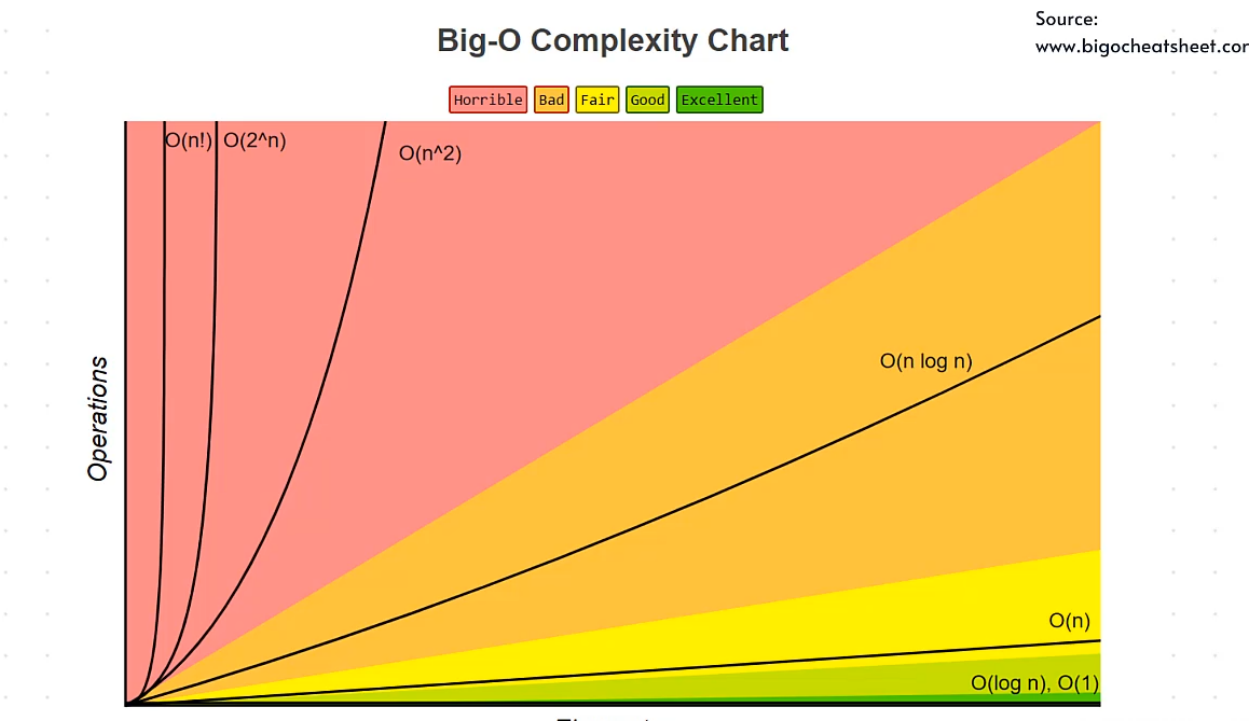
BIG O

- It doesn't depends on time, it depends on how many steps we are performing.
- The main focus is to calculate the amount of work we do or the number of comparisons we perform.



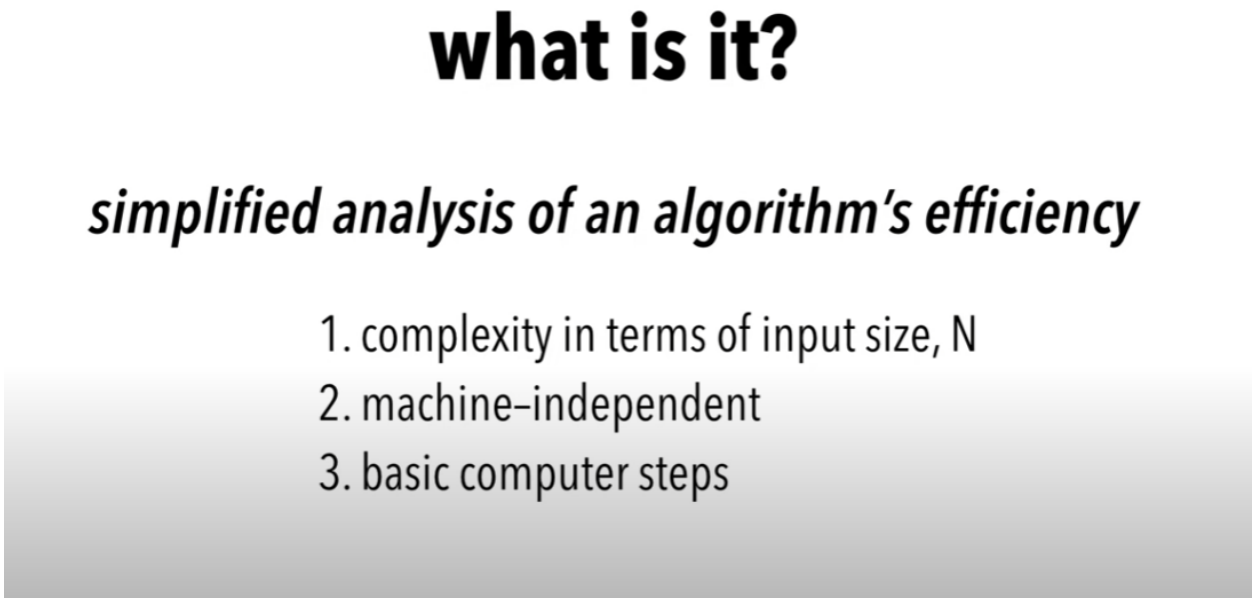


Whichever system we run solution b is much more efficient than the solution a



In short, Big-O is a parameter to tell the time and space complexity of the algorithm.

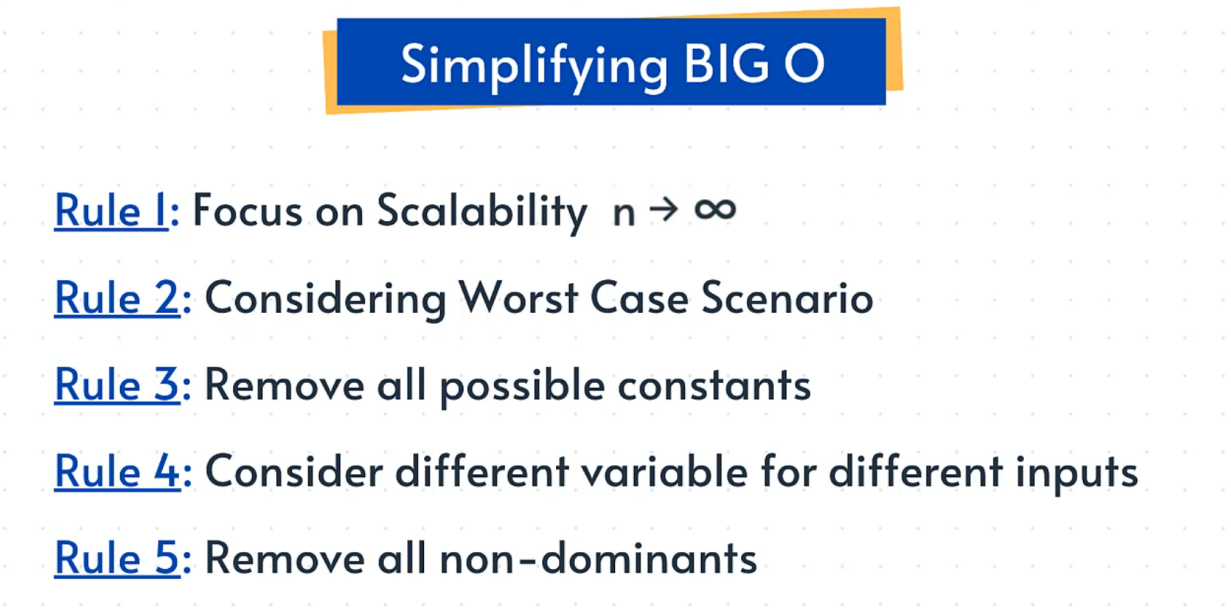
More About Big O:



Big-O notation in 5 minutes — The basics

5TH VIDEO COMPLETED

Steps for simplifying the big-o notation



Our n(number of inputs can be very large we have to keep that in mind)
We always have to consider the worst case scenario while calculating the big-o
We have to remove any constants after adding all the individual time or space complexity
If we have more than one input consider them different while calculating time or space complexities like n,m..etc
Remove all the non dominant i.e while n^2 n is non dominant and similarly we have to remove then non-dominant

The worst complexity is O(n!) which is rear

For space complexity we have to calculate the amount of extra space required to run the algorithm

Logarithm

log(1) = 0

log(2) = 1

log(4) = 2

log(8) = 3

log(16) = 4

log(32) = 5

log(64) = 6

When you double the value of n, the value of log only increases by 1.

Where we can expect log?

We see usage of log during searching and sorting algorithms.

Recrussion

What is Recursion?

Recursion = a way of solving a problem by having a function calling itself

- Performing the same operation multiple times with different inputs
- In every step we try smaller inputs to make the problem smaller.
- Base condition is needed to stop the recursion, otherwise infinite loop will occur.

When to use the Recursion

Why Recursion?

1. Recursive thinking is really important in programming and it helps you break down big problems into smaller ones and easier to use
   - when to choose recursion?
     - If you can divine the problem into similar sub problems
     - Design an algorithm to compute nth...
     - Write code to list the n...
     - Implement a method to compute all.
     - Practice
2. The prominent usage of recursion in data structures like trees and graphs.
3. Interviews
4. It is used in many algorithms (divide and conquer, greedy and dynamic programming)

Recursive vs Iterative Solutions

```python
def powerOfTwo(n):
    if n == 0:
        return 1
    else:
        power = powerOfTwo(n-1)
        return power * 2
```

```python
def powerOfTwoIt(n):
    i = 0
    power = 1
    while i < n:
        power = power * 2
        i = i + 1
    return power
```

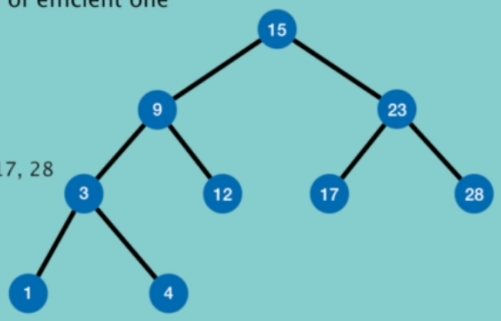
Points	Recursion	Iteration	
Space efficient?	No	Yes	No stack memory require in case of iteration
Time efficient?	No	Yes	In case of recursion system needs more time for pop and push elements to stack memory which makes recursion less time efficient
Easy to code?	Yes	No	We use recursion especially in the cases we know that a problem can be divided into similar sub problems.

When to Use/Avoid Recursion?

When to use it?

- When we can easily breakdown a problem into similar subproblem
- When we are fine with extra overhead (both time and space) that comes with it
- When we need a quick working solution instead of efficient one
- When traverse a tree
- When we use memoization in recursion

  - preorder tree traversal : 15, 9, 3, 1, 4, 12, 23, 17, 28



When avoid it?

- If time and space complexity matters for us.
- Recursion uses more memory. If we use embedded memory. For example an application that takes more memory in the phone is not efficient
- Recursion can be slow

How to write recursion in 3 steps?

Step 1 : Recursive case – the flow

$n! = n * (n-1) * (n-2) * \dots * 2 * 1$ ⟶ **$n! = n * (n-1)!$**

$(n-1)!$

$(n-1)! = (n-1) * (n-1-1) * (n-1-2) * \dots * 2 * 1 = (n-1) * (n-2) * (n-3) * \dots * 2 * 1$

Step 2 : Base case – the stopping criterion

- $0! = 1$
- $1! = 1$

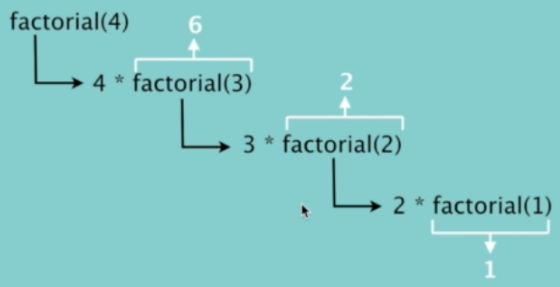
Step 3 : Unintentional case – the constraint

- factorial(-1) ??
- factorial(1.5) ??

How to write recursion in 3 steps?

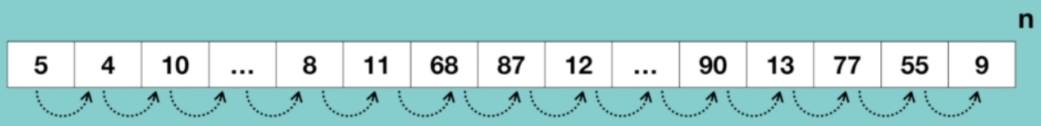
```
def factorial(n):
    assert n >= 0 and int(n) == n, 'The number must be positive integer only!'
    if n in [0,1]:
        return 1
    else:
        return n * factorial(n-1)
```

factorial(4) = 24



Big O, Big Theta and Big Omega

- **Big O** : It is a complexity that is going to be less or equal to the worst case.
- **Big - Ω (Big-Omega)** : It is a complexity that is going to be at least more than the best case.
- **Big Theta (Big - Θ)** : It is a complexity that is within bounds of the worst and the best cases.



Algorithm run time complexities

Complexity	Name	Sample
O(1)	Constant	Accessing a specific element in array
O(N)	Linear	Loop through array elements
O(LogN)	Logarithmic	Find an element in sorted array
$O(N^2)$	Quadratic	Looking ar a every index in the array twice
$O(2^N)$	Exponential	Double recursion in Fibonacci

Why do we drop constants and non dominant terms?

- It is very possible that O(N) code is faster than O(1) code for specific inputs
- Different computers with different architectures have different constant factors.

Fast computer
Fast memory access
Lower constant

Slow computer
Slow memory access
Higher constant

- Different algorithms with the same basic idea and computational complexity might have slightly different constants

  Example: a*(b-c) **vs** a*b - a*c

- As $n \to \infty$, constant factors are not really a big deal

Add vs Multiply

```
for a in arrayA:
    print(a)

for b in arrayB:
    print(b)
```

Add the Runtimes: *O(A + B)*

```
for a in arrayA:
    for b in arrayB:
        print(a, b)
```

Multiply the Runtimes: *O(A * B)*

- If your algorithm is in the form "do this, then when you are all done, do that" then you add the runtimes.
- If your algorithm is in the form "do this for each time you do that" then you multiply the runtimes.

How to measure the codes using Big O?

No	Description	Complexity
Rule 1	Any assignment statements and if statements that are executed once regardless of the size of the problem	O(1)
Rule 2	A simple "for" loop from 0 to n (with no internal loops)	O(n)
Rule 3	A nested loop of the same type takes quadratic time complexity	$O(n^2)$
Rule 4	A loop, in which the controlling parameter is divided by two at each step	O(log n)
Rule 5	When dealing with multiple statements, just add them up	

sampleArray | 5 | 4 | 10 | ... | 8 | 11 | 68 | 87 | ...

```
def findBiggestNumber(sampleArray):
    biggestNumber = sampleArray[0] ----------> O(1)
    for index in range(1,len(sampleArray)):----> O(n)
        if sampleArray[index] > biggestNumber: ---> O(1)
            biggestNumber = sampleArray[index] ---> O(1)
    print(biggestNumber) ----------> O(1)
```

O(n) + O(1) + O(1) = O(n)

Time complexity : O(1) + O(n) + O(1) = O(n)

For recursive algorithms calculation of time complexity is different

How to measure Recursive Algorithm?

sampleArray | 5 | 4 | 10 | ... | 8 | 11 | 68 | 87 | 10

```
def findMaxNumRec(sampleArray, n):----------> M(n)
    if n == 1: ----------> O(1)
        return sampleArray[0] ----------> O(1)
    return max(sampleArray[n-1],findMaxNumRec(sampleArray, n-1))--------> M(n-1)
```

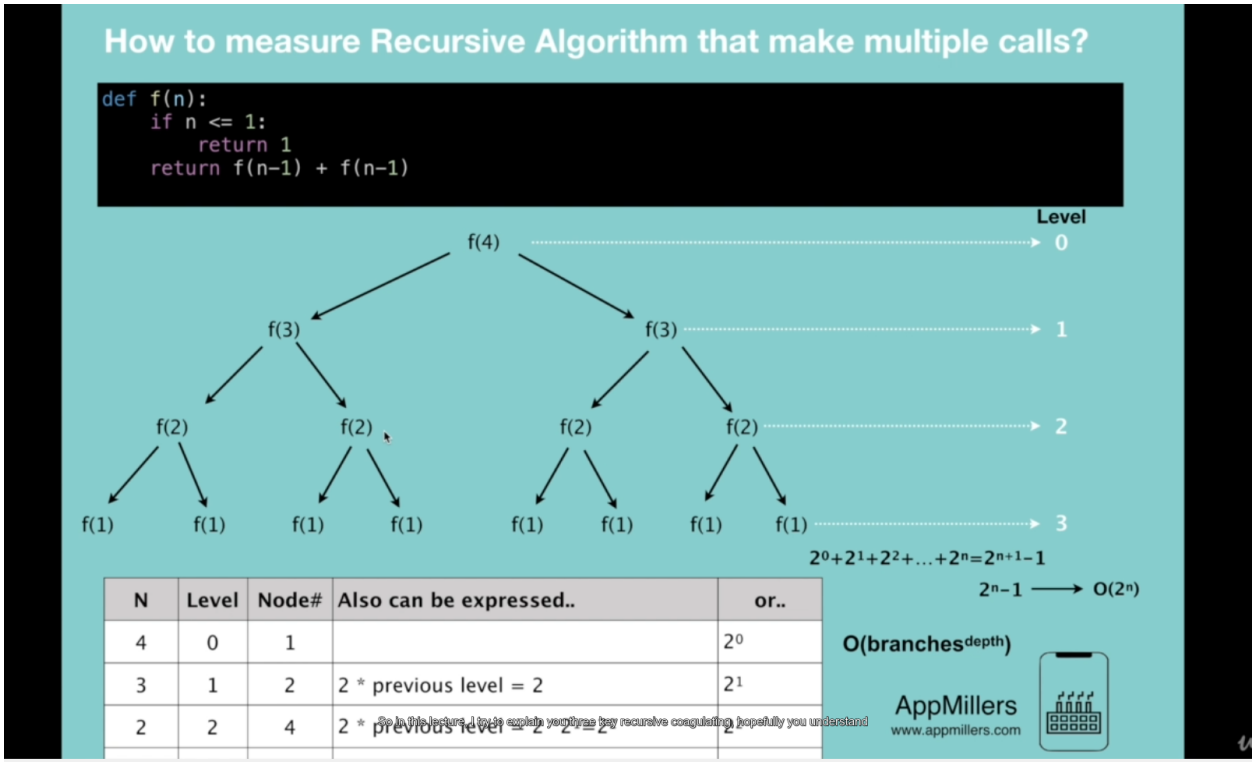
$M(n)=O(1)+M(n-1)$

$M(1)=O(1)$

$M(n-1)=O(1)+M((n-1)-1)$

$M(n-2)=O(1)+M((n-2)-1)$

$M(n)=1+M(n-1)$
$=1+(1+M((n-1)-1))$
$=2+M(n-2)$
$=2+1+M((n-2)-1)$
$=3+M(n-3)$
$\vdots$
$=a+M(n-a)$
$=n-1+M(n-(n-1))$
$=n-1+1$
$=n$



N	Level	Node#	Also can be expressed..	or..
4	0	1		2^0
3	1	2	2 * previous level = 2	2^1
2	2	4	2 * previous level = $2^1 * 2 = 2^2$	

Interview Questions

Answer:

Interview Questions - 1

What is the runtime of the below code?

```
def foo(array):
    sum = 0 ----------> O(1)
    product = 1 ----------> O(1)

    for i in array: ----------> O(n)
        sum += i ----------> O(1)

    for i in array: ----------> O(n)
        product *= i ----------> O(1)
    print("Sum = "+str(sum)+", Product = "+str(product)) ----------> O(1)
```

Time Complexity : O(N)

Interview Questions - 2

What is the runtime of the below code?

```
def printPairs(array):
    for i in array:
        for j in array:
            print(str(i)+","+str(j))
```

Answer:

Interview Questions - 2

What is the runtime of the below code?

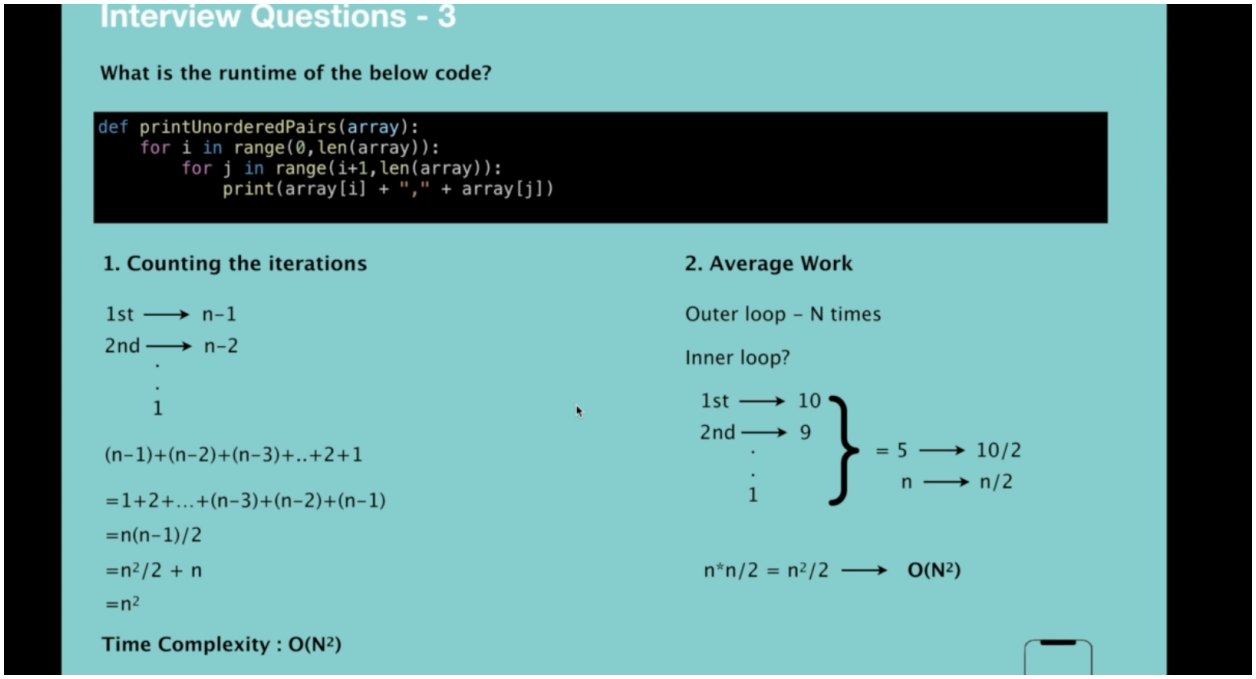
```
def printPairs(array):
    for i in array: ----------> O(n^2)} ------> O(n^2)
        for j in array: ----------> O(n)}
            print(str(i)+","+str(j)) ----------> O(1)
```

Interview Questions - 3

What is the runtime of the below code?

```
def printUnorderedPairs(array):
    for i in range(0,len(array)):
        for j in range(i+1,len(array)):
            print(array[i] + "," + array[j])
```

Answer:



Interview Question - 4

What is the runtime of the below code?

```
def printUnorderedPairs(arrayA, arrayB):
    for i in range(len(arrayA)):
        for j in range(len(arrayB)):
            if arrayA[i] < arrayB[j]:
                print(str(arrayA[i]) + "," + str(arrayB[j]))
```

Answer:

Interview Question - 5

What is the runtime of the below code?

```
def printUnorderedPairs(arrayA, arrayB):
    for i in range(len(arrayA)):
        for j in range(len(arrayB)):
            for k in range(0,100000):
                print(str(arrayA[i]) + "," + str(arrayB[j]))
```

Answer

Interview Question - 5

What is the runtime of the below code?

```
def printUnorderedPairs(arrayA, arrayB):
    for i in range(len(arrayA)):          }
        for j in range(len(arrayB)): }  ----------> O(ab)
            for k in range(0,100000):   ----------> O(1)
                print(str(arrayA[i]) + "," + str(arrayB[j])) ----------> O(1)
```

a = len(arrayA)

b = len(arrayB)

100,000 units of work is still constant

Time Complexity : O(ab)

Interview Question - 6

What is the runtime of the below code?

```
def reverse(array):
    for i in range(0, int(len(array)/2)):
        other = len(array)-i-1
        temp = array[i]
        array[i] = array[other]
        array[other] = temp
    print(array)
```

Answer:

Interview Question - 6

What is the runtime of the below code?

```
def reverse(array):
    for i in range(0, int(len(array)/2)): ----------> O(N/2) ---> O(N)
        other = len(array)-i-1 ----------> O(1)
        temp = array[i]  ----------> O(1)
        array[i] = array[other]----------> O(1)
        array[other] = temp  ----------> O(1)
    print(array) ----------> O(1)
```

Time Complexity : O(N)

Question and Answer: