**Data Structures** are different ways of organizing data on your computer, that can be used effectively.

**Algorithms –** set of steps to accomplish a task. In CS, it’s a set of rules for a computer program to accomplish a task.

What makes a good algorithm?

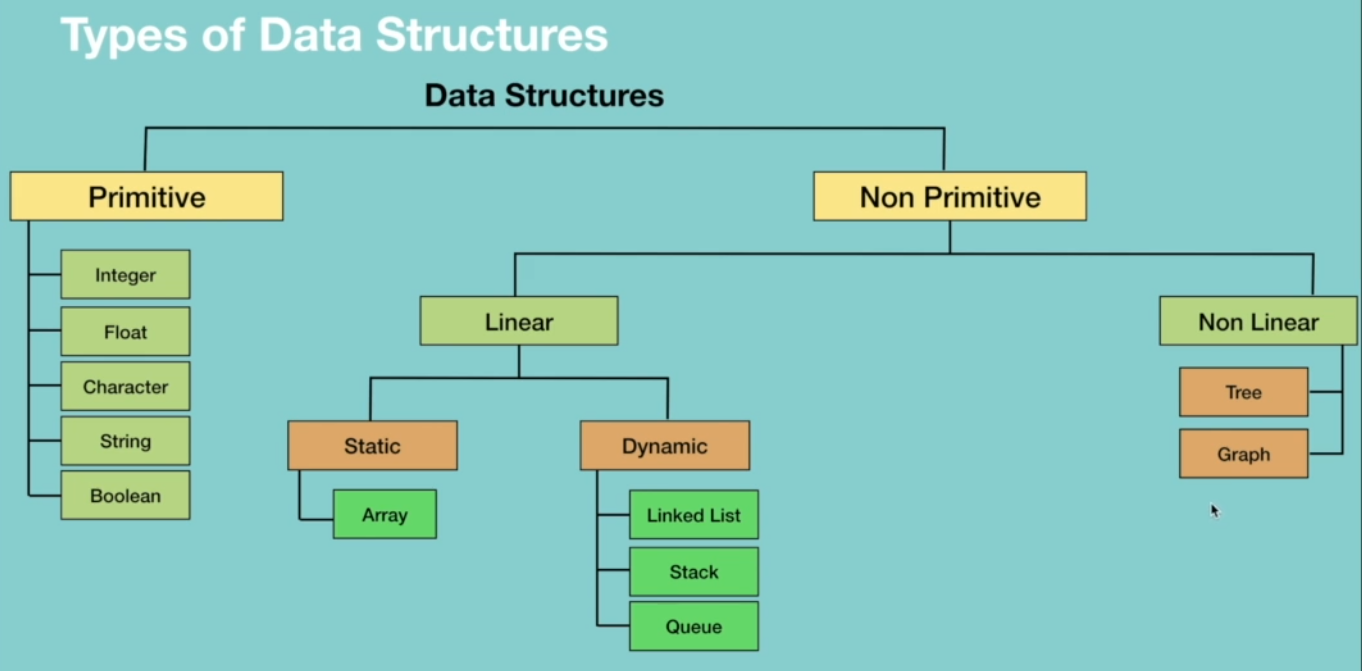
* Correctness
* Efficiency

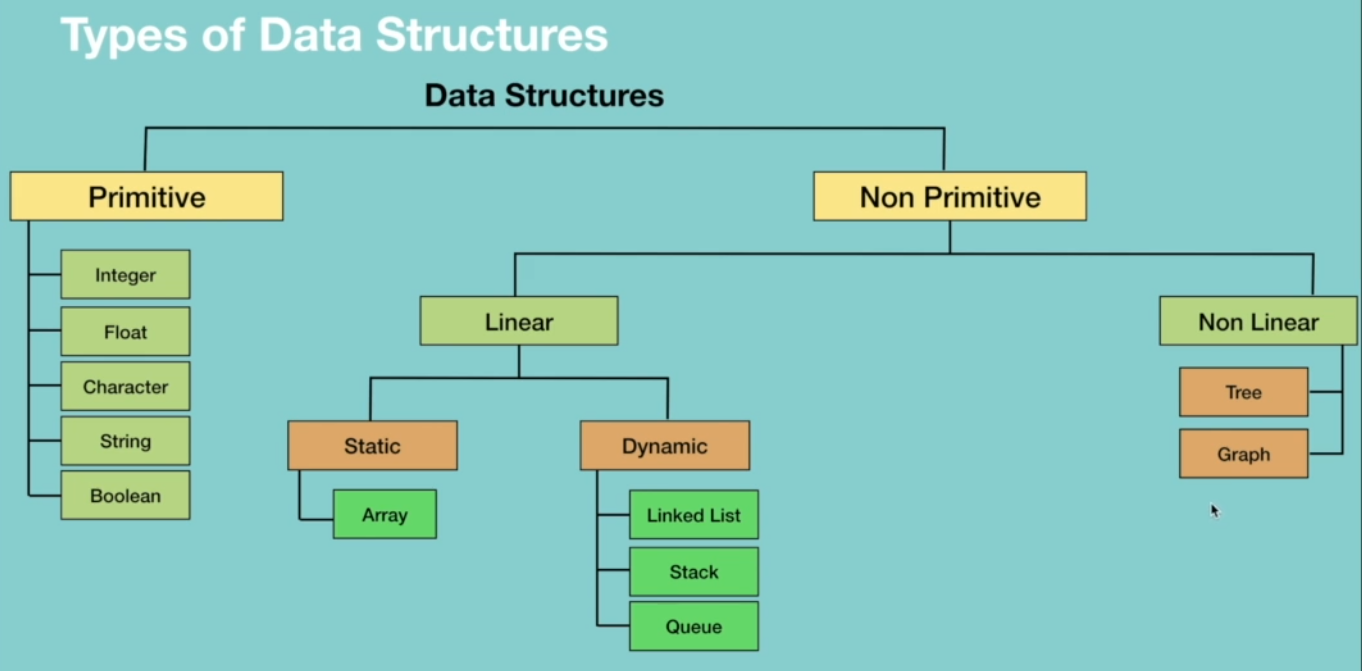
**Types of Data Structures**

1. Primitive data structures – built in the programming language itself and are available to the programmer e.g. integer, float, character, string, and Boolean
2. Non-primitive data structures – user defined data structures, combine two or more data structures divided into:

Linear data structure – data items are arranged in memory in a linear sequential manner and can be static or dynamic. Static linear DS – structured associated memory locations are fixed in the runtime but in dynamic linear DS memory locations change.

Non-linear DS – data item is connected to several other data items.



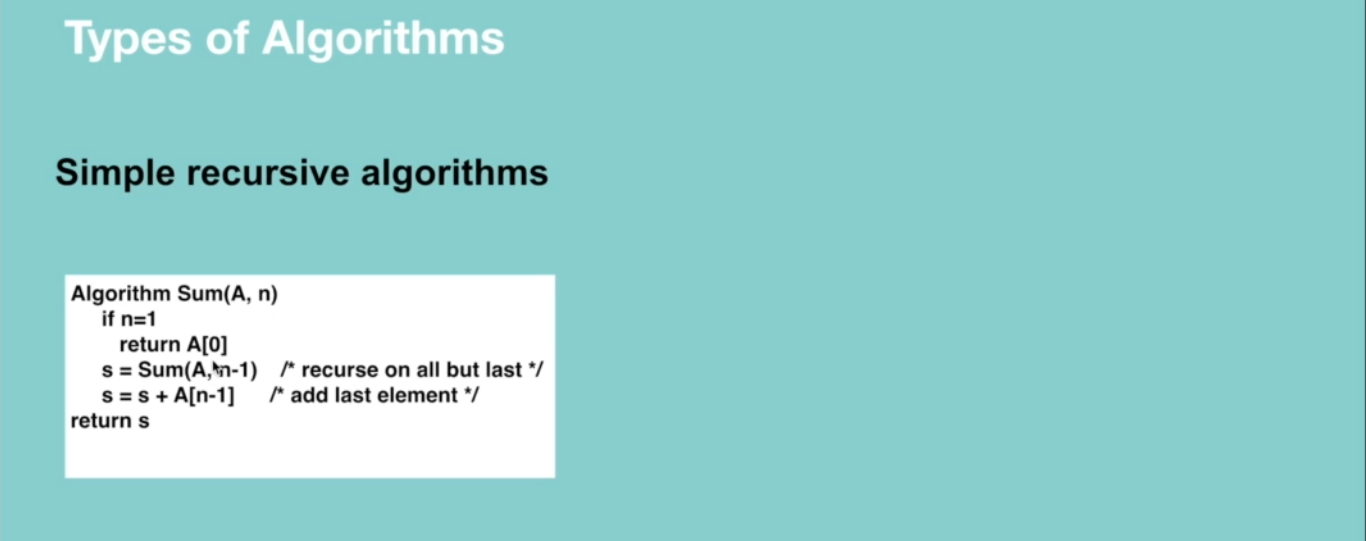


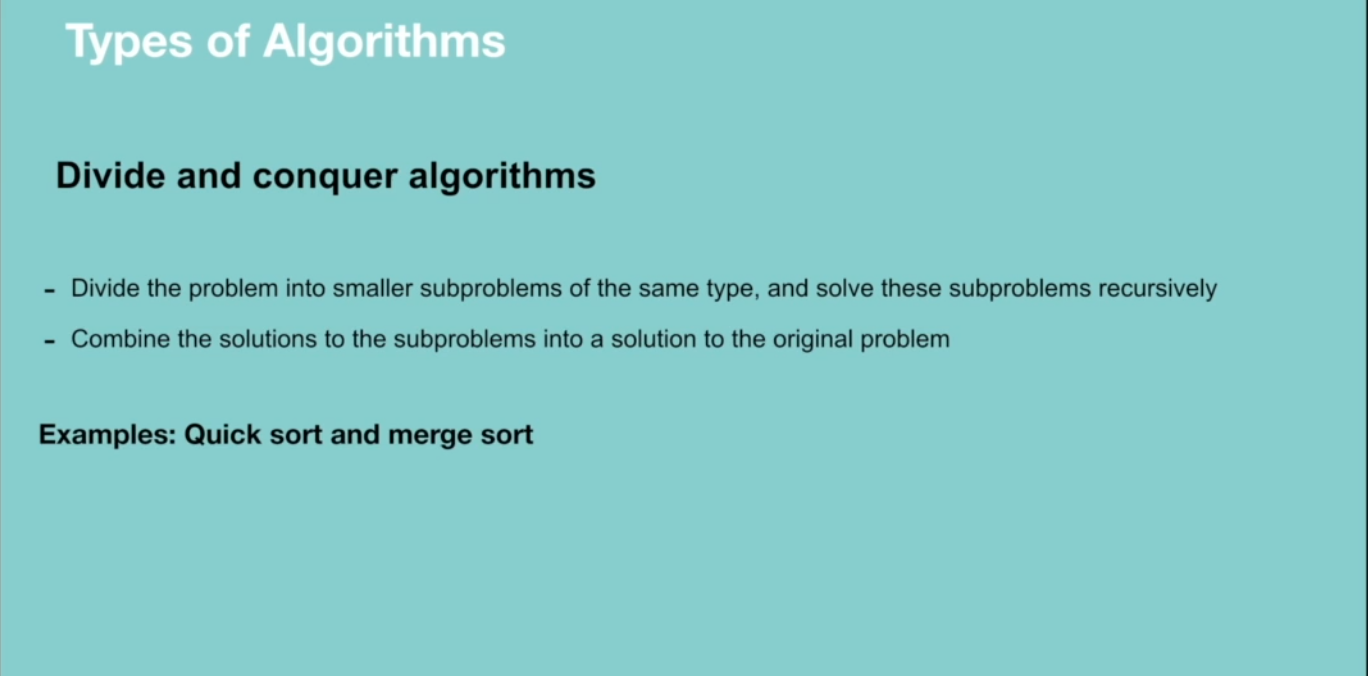
**Types of Algorithms**

They are classified either based on the type of problem they are trying to solve or based on the problem-solving technique.

They include:

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

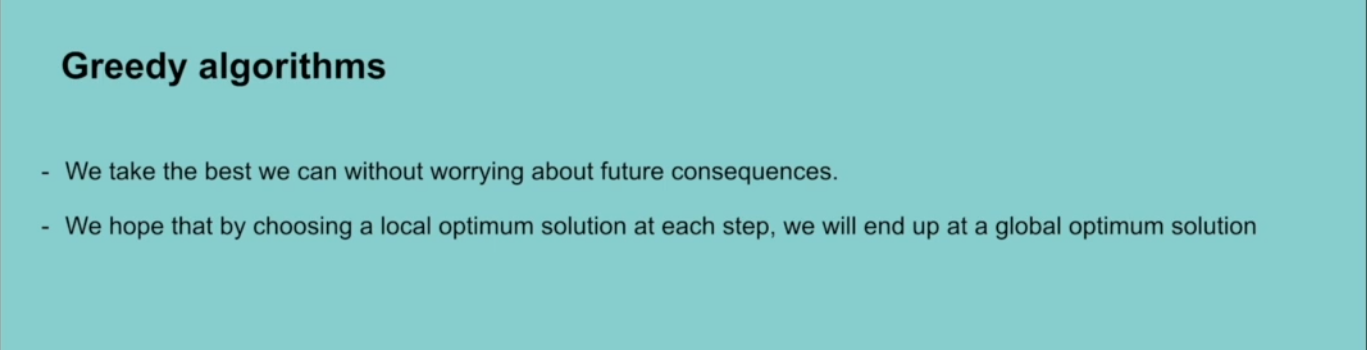


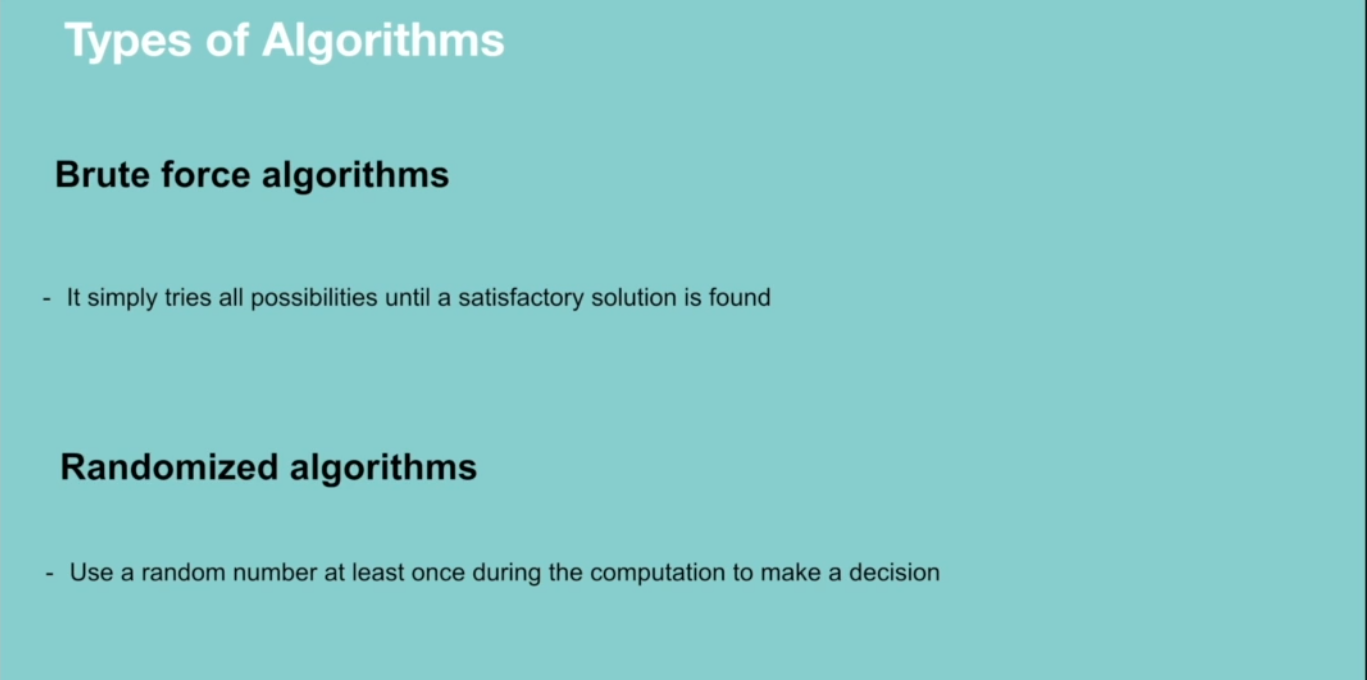


Dynamic programming works on memoization which means the algorithm remembers the past results and use them to find new results. It’s generally used for optimization problems i.e. to find the best solution among multiple solutions.



Greedy algorithm works in phases. It’s also used for finding the best solution i.e. optimization problems.





**RECURSION**

A way of solving a problem by having a function calling itself.

**Properties of recursion**

* Performing the same operation multiple times with different inputs.
* In every step we try smaller inputs to make the problem smaller.
* A base condition is needed to stop the recursion.

**Why Recursion?**

* Recursive thinking is really important in programming and it helps you break down big problems into smaller ones and easier to use. Note: There are situations when iterations perform better than recursion.

When to choose recursion – when you can divide the problem into similar subproblems. The subproblem must be similar otherwise recursion is not a choice.

**How do you know that the subproblem is similar in nature?**

* Design an algorithm to compute nth…
* Write code to list the n…
* Implement a method to compute all.

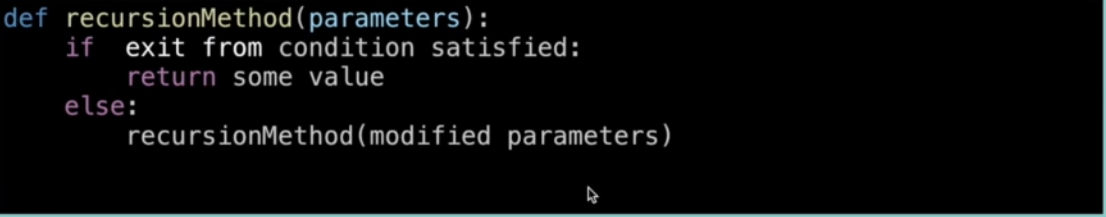
**Why Recursion**

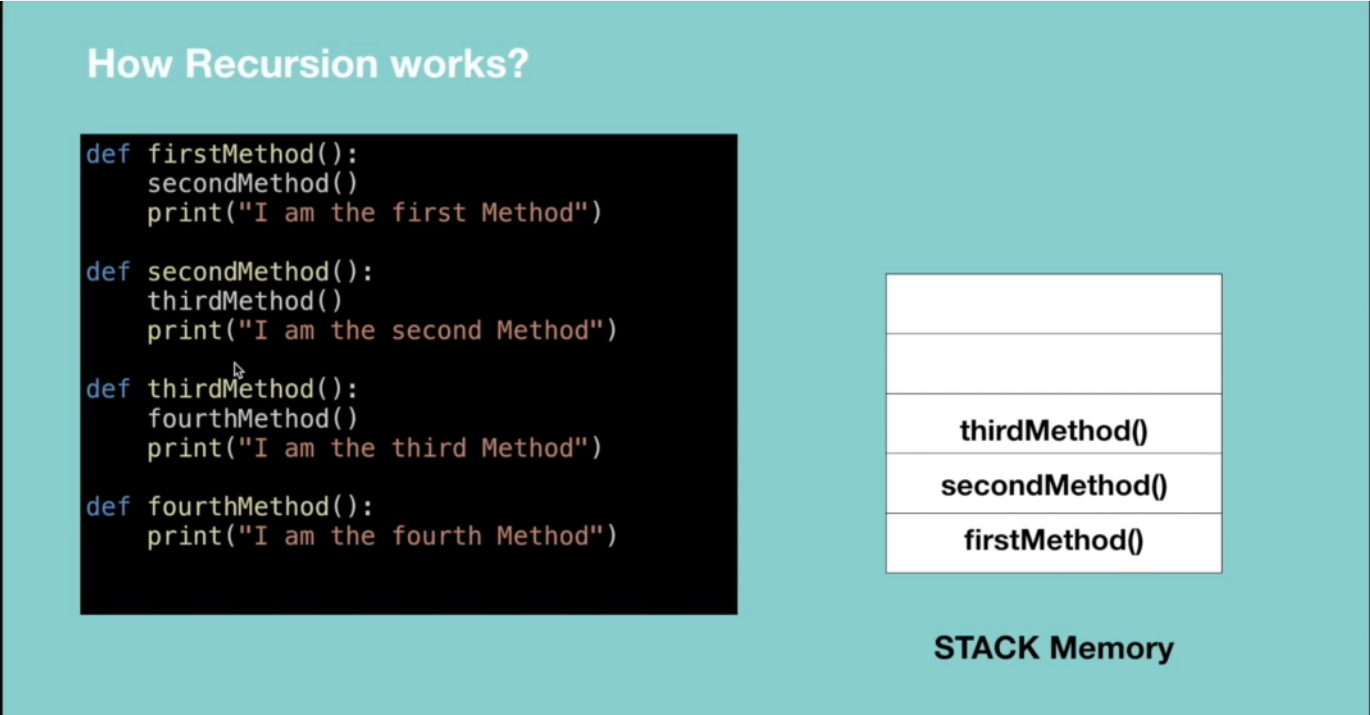
* The prominent usage of recursion in data structures like trees and graphs.
* Many big companies ask questions in relation to recursion in the interviews.
* It is used in many algorithms (divide and conquer, greedy and dynamic programming)

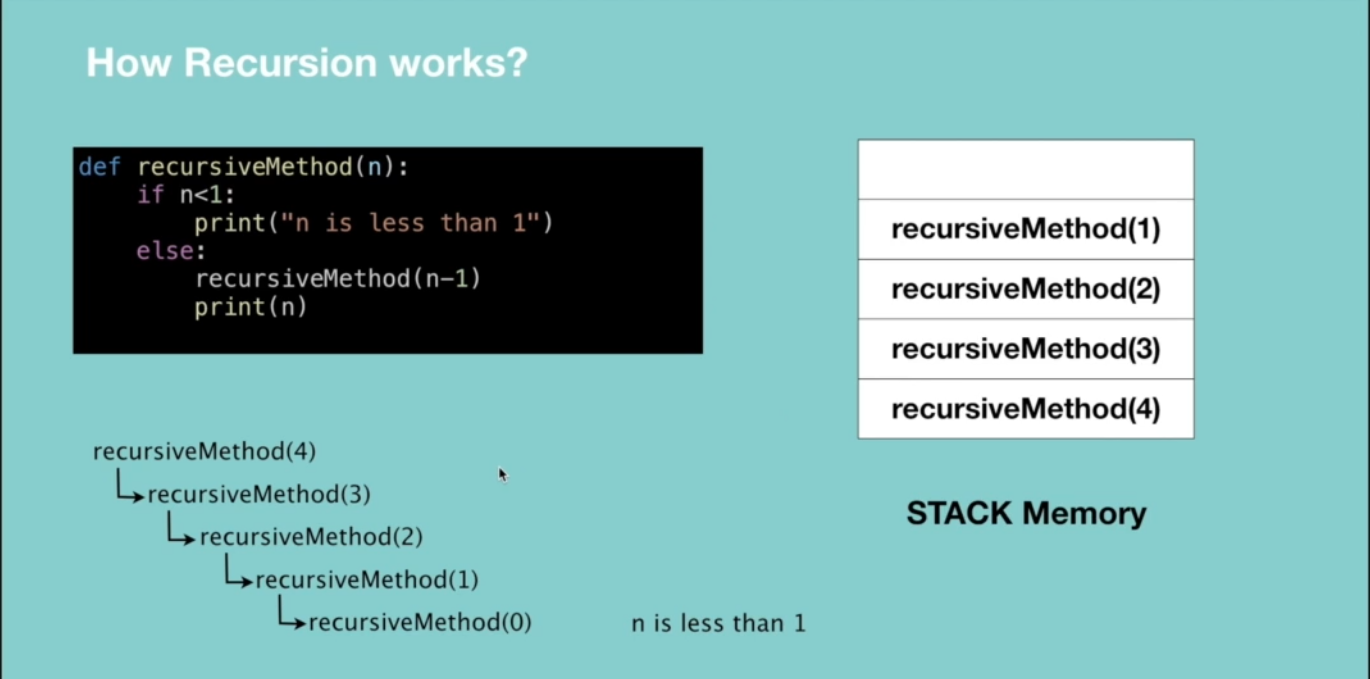
**How Recursion works**

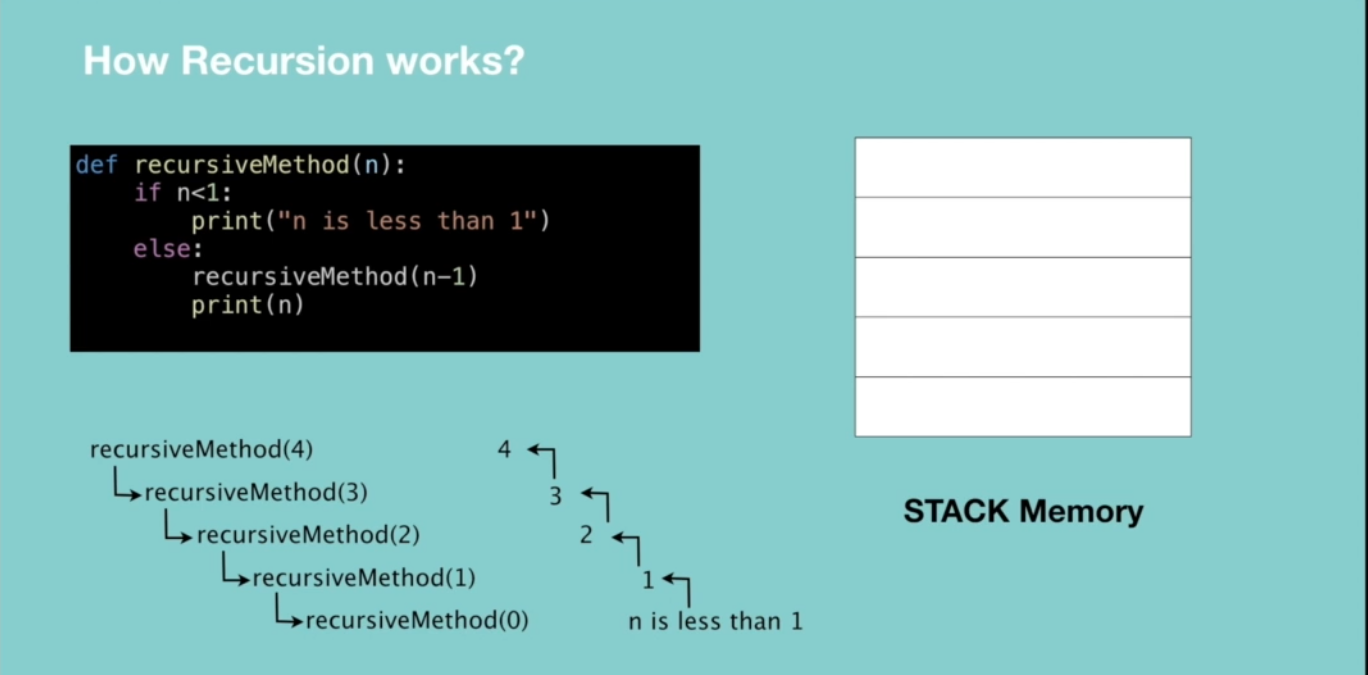
When designing a recursive call, we have to consider two conditions:

* A condition where a method calls itself with similar values.
* A condition to exit from the infinite loop.



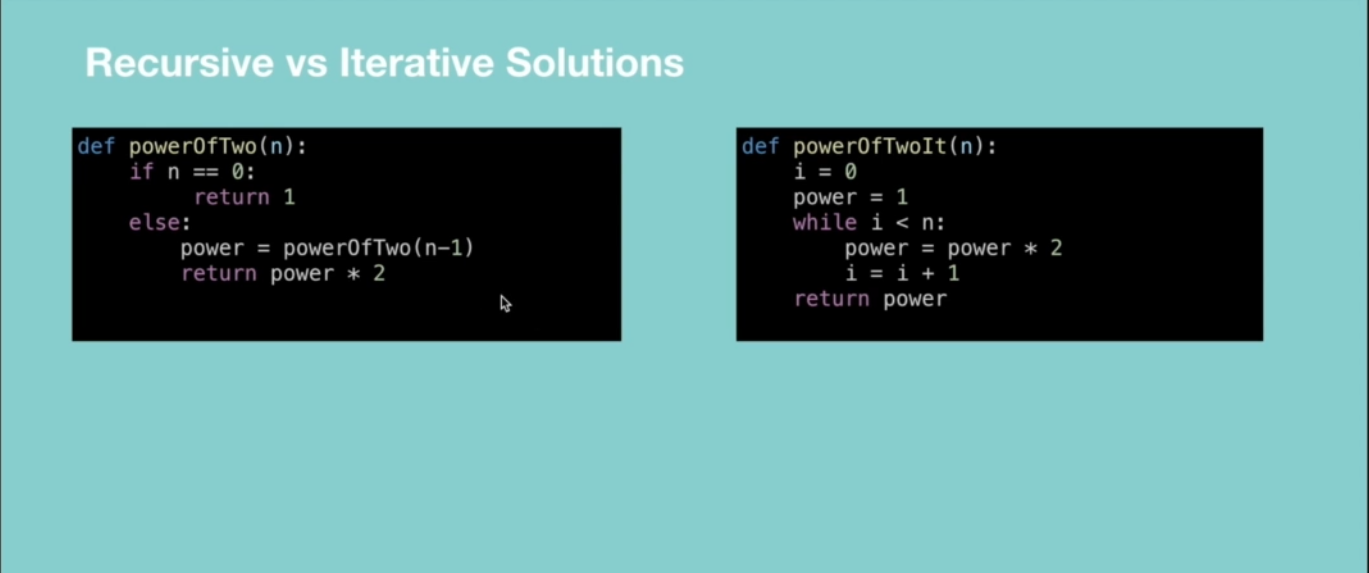




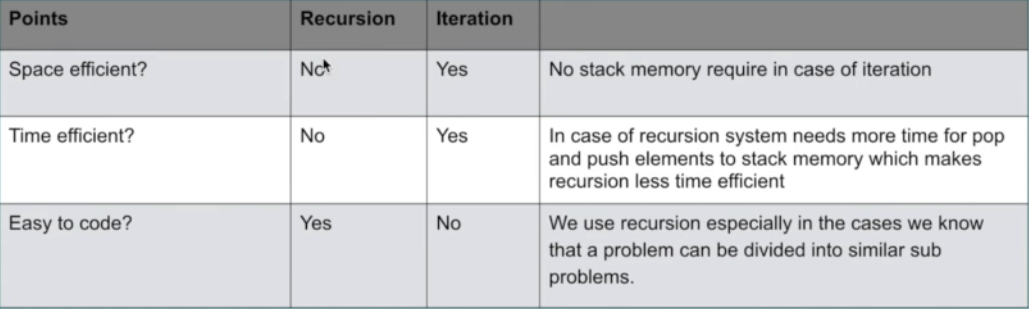


**Recursive vs Iterative solutions.**

All recursive methods can be implemented iteratively.



* In recursion, a conditional statement determines the termination of the function while in iterative solutions a controlled variable value decides the termination.
* In recursion, infinite recursion can lead to a system crash while infinite iterations consume CPU cycles.
* Recursion repeatedly invokes the mechanism consequently overhead of method calls, this is expensive in both processor time and memory space while iteration does not.
* Recursive algorithm can be very space inefficient. i.e. if your algorithm resources to the depth of N, it uses at least O(N) memory hence it’s better to implement recursive algorithm iteratively.



**When to Use/Avoid Recursion?**

**When to use it?**

* When we use memoization in recursion – this reduces the time complexity.
* 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. Note when you are developing a mobile app which should run on low memory devices, then here recursion is not advisable. If you’re designing a critical system like an airbag in the car, don’t use recursion.
* When we need a quick working solution instead of efficient one e.g. factorial and Fibonacci numbers.
* When traverse a tree

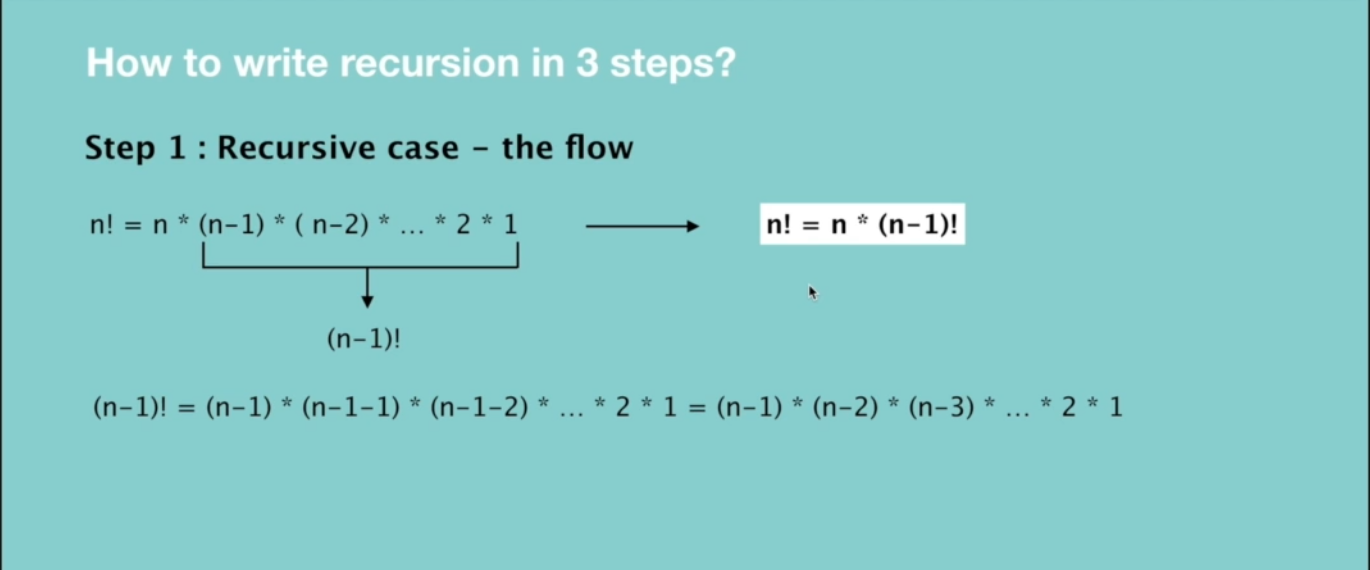
**When avoid it?**

* If time and space complexity matter 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.

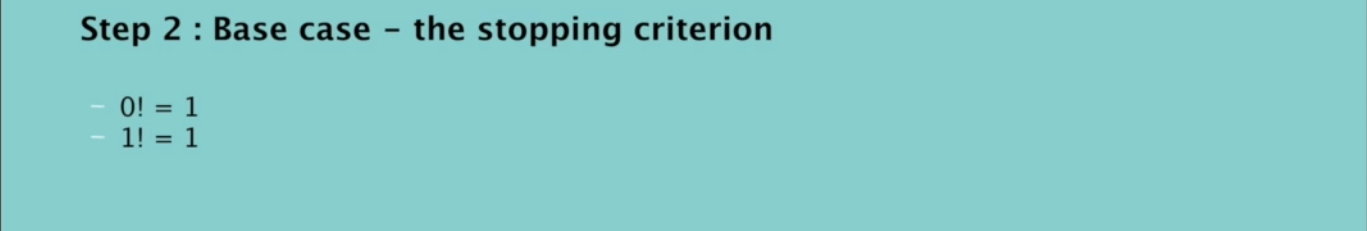
**How to write recursion in 3 steps**

Factorial => the product of all positive integers less than or equal to n.

**Step 1: Recursive case – the flow**



**Step 2: Base case – the stopping criteria**



**Step 3: Unintentional case - the constraint**



