A recursive function by definition is a function which elegantly calls itself during execution to solve a certain task. In general in computer science and programming it is considered one of the more elegant ways to solve tasks because their execution is so beautiful, they basically solve the problems themselves, without invoking an external function nor relying on other methods. It creates the image that the problem is so easy to solve, it solves itself! The first example shown during the video is that of the factorial. A factorial of any number refers to the multiplication of all positive integers starting from 1, as 0 would undoubtedly just make any factorial 0, up until and including that number. It is represented by an exclamation mark and the highest number of the series. An example would be !5 (factorial of 5), this would represent 5\*4\*3\*2\*1 and would equal 120.

Now to be able to solve any factorial or to create the function which would solve a factorial operation of any number n (where 0<n), we first have to define the two types of cases that can occur during any recursive function: the recursive case and the base case. Recursive functions rely on a special type of logic to solve problems, they use a method which involves reevoking itself various times until a certain scenario and condition is met. The condition that must be fulfilled is what we call the base case as it is what must be fulfilled to stop the method from re-invoking itself and instead returns the first value or makes the first operation which will then be sent to the last invocation of the function and that way keep going up until all the recursive cases which had built themselves up on the stack are completed.

Taking as an example the factorial function again. When the method is first called, an integer value n is passed as a parameter. This number n defines the largest value or the number we call the factorial of: the operation to solve would then be !n (factorial of n). To solve this using a recursive function we would first have to find the base case of the function, which in this case would be !1 = 1. Why? Because if you think about the definition of a factorial, the fact that it multiplies all the integers before that also implies that the factorial of any number(n) would be the same as n multiplied by the factorial of the number before that because factorial of n-1 includes the multiplication of all previous integers. Therefor we know that !n = n\*!n-1, an example would be !2 = 2\*!1. This is what is necessary to create the recursive case or create a recursive function. Now since we know that this is true and that !1=1, we now have both necessary cases, base and recursive, and therefor only need to write a function which calls itself over and over again, in every turn decreasing the value of n up until it reaches one, where the base case is called, returning the first value and later solving all the rest. One could say that this procedure is very similar to a loop, and in fact it is: a simple for loop would solve the problem in n executions, and yet it is not as elegant as the recursion simply because the recursive method does not require the creation of a counter variable or anything similar, it just simply needs to keep recalling itself until it has the solution.

Later in the video, another case of a recursive method is shown, a Fibonacci sequence solver. The Fibonacci sequence as a matter of fact is a sequence which builds on the idea that, starting from 0 and then 1, every following integer is found by adding the two previous number together, resulting in the shape of n = n-1 + n-2 if n referred to the position in the sequence. The Fibonacci sequence in the video is used to convey the idea that often there are not 1 but many base cases which must be accounted for, in the Fibonacci sequence it refers to the fact that the first two integers in the series, 0 and 1, both must be stated as base cases seeing as none of them can be inferred with the recursive case n = n-1 + n-2. Furthermore, base cases are not the only ones of which there can be various, recursive cases can also occur in abundance, take the Collatz conjecture as an example. The Collatz conjecture is speculates that all positive integers can be brought back down to 1, meaning can be reduced back down to 1, if either they are divided by 2(n/2) or multiplied by 3 and then 1 added (3n +1). This makes sense seeing as 2 is the most used prime number and 3n+1 simply multiplies by the second most common prime number. The moment n is a multiple of two the Collatz conjecture solves itself by dividing by 2 repeatedly so what the Collatz conjecture really proves is that any positive integer can become a multiple of two if multiplied by 3n+1 enough and divided by 2 enough times. Seeing as now there are two recursive cases, the question arises as to when each would be used. The answer to this lies in the condition. Each must have a condition which must be fulfilled, for n/2 it would be that the integer is even and for 3n+1 that it is odd. Finally, the question arises as to what the base case is, we have already mentioned it but it would be 1. With that we can easily write a recursive function.

Now as to a last question many might ask themselves, why is the Collatz conjecture a conjecture and not a theorem, implying that a conjecture can onlybecome a theorem when it is rigorously proved by mathematical reasoning. The answer to that lies in the fact that many numbers involve large amounts of steps before they can be solved, even numbers like 27 would take 111 steps to complete and would climb to numbers as high as 9232 before being solved. As numbers increase we come to a point where modern day computers simply do not have the computer power to solve these conjectures and elevate them to the standard of theorem, but that day will hopefully come.

fact(n){

if(n==1) return 1;

return n\*fact(n-1);

}

fib(n){

if(n==1) return 1;

if(n==2) return 2;

return fib(n-1) + fib(n-2);

}

collatz(n){

if(n ==1) return 0;

if(n%2 ==0) return 1+collatz(n/2);

return 1+collatz(3n+1);

}