PROGRAMMING PARADIGMS AND LANGUAGES-RECURSION

ROVSHAN AHMADLI

RASHAD MAHMUDOV

WHAT IS PROGRAMMING PARADIGM?

A programming paradigm is a fundamental style of computer programming, serving as away of building the structure and elements of computer programs.

In other words, programming paradigms are a way to classify programming languages based on their features.

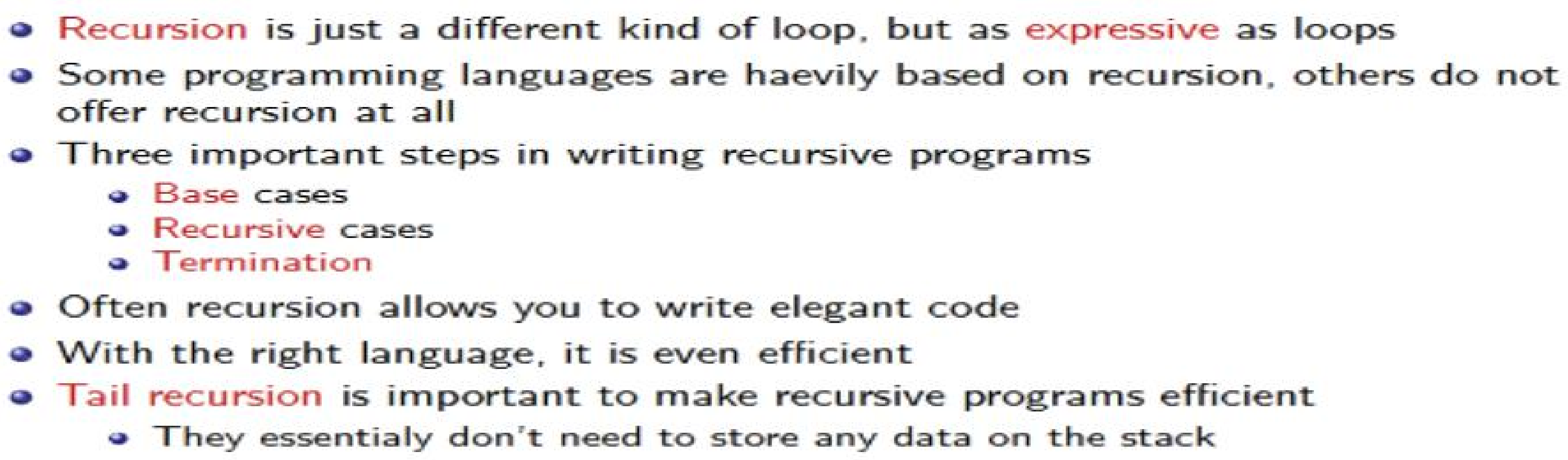
Languages can be classified into multiple paradigms.

1.-Some paradigms are concerned mainly with implications for the execution model of the language such as allowing side effects

2.-Other paradigms are concerned mainly with the way that code is organized, such as groping a code into units along with the state that is modified by the code.

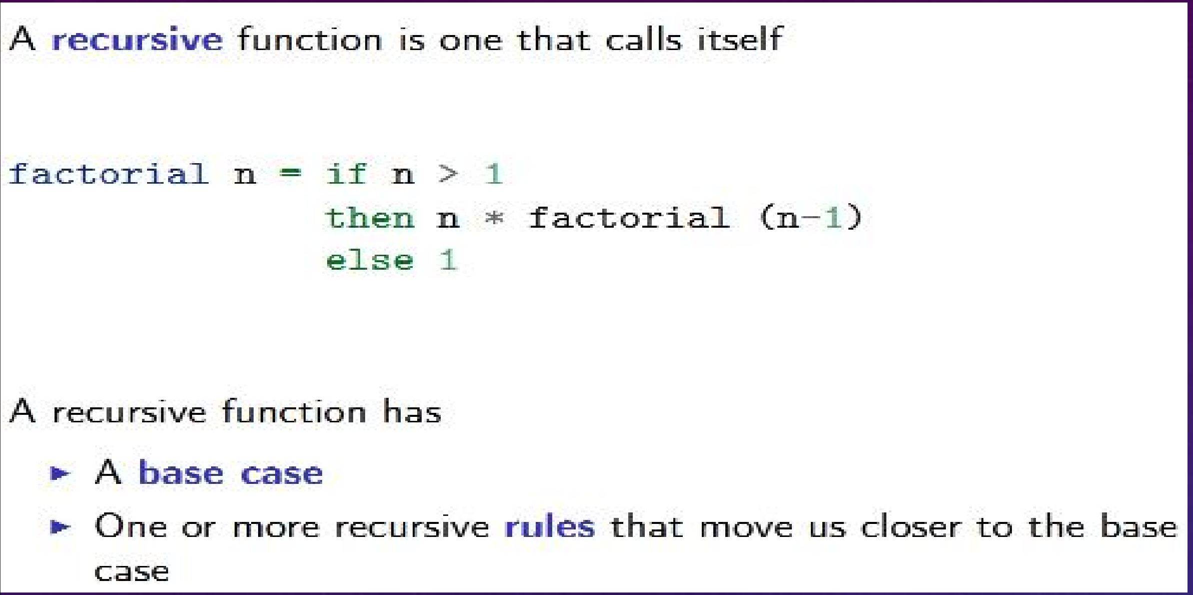
3.-Yet others are concerned mainly with the style of syntax and grammar.

GENERAL ABOUT RECURSION



- Recursion is just a different kind of loop, but as expressive as loops
- Some programming languages are haevily based on recursion, others do not offer recursion at all
- Three important steps in writing recursive programs
  - Base cases
  - Recursive cases
  - Termination
- Often recursion allows you to write elegant code
- With the right language, it is even efficient
- Tail recursion is important to make recursive programs efficient
  - They essentialy don't need to store any data on the stack

WHAT IS RECURSION?

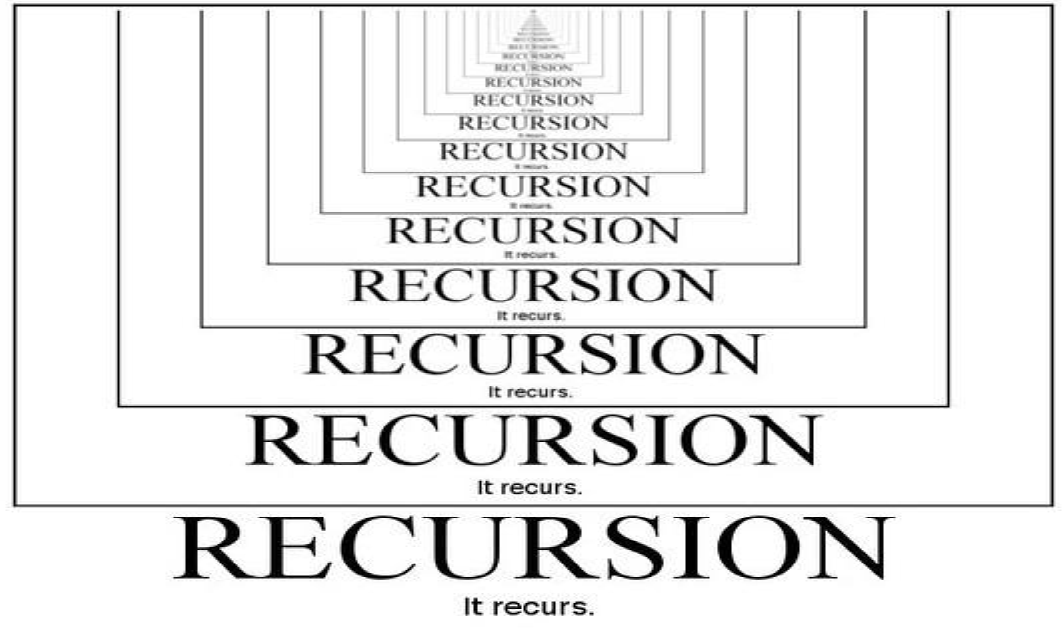


A **recursive** function is one that calls itself

```
factorial n = if n > 1
              then n * factorial (n-1)
              else 1
```

A recursive function has

- A **base case**
- One or more recursive **rules** that move us closer to the base case



- **Recursion** in computer science is a method of solving a problem where the solution depends on solutions to smaller instances of the same problem. Such problems can generally be solved by iteration, but this needs to identify and index the smaller instances at programming time

- Recursion is a powerful technique for thinking about a process

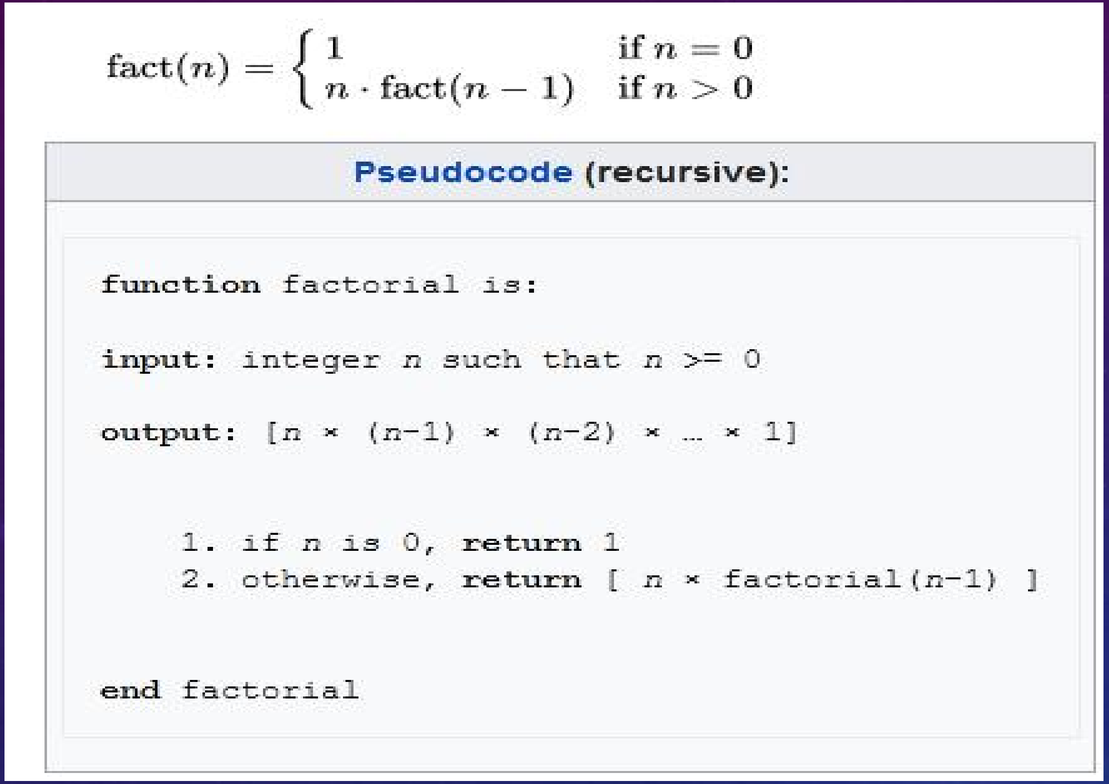
- It can be used to simulate a loop, or for many other kinds of applications
- In recursion, a function or procedure calls itself

TYPES OF RECURSION:

- 1. Single recursion and multiple recursion : Recursion that only contains a single self-reference is known as **single recursion**, while recursion that contains multiple self-references is known as **multiple recursion**. Standard examples of single recursion include list traversal, such as in a linear search, or computing the factorial function, while standard examples of multiple recursion include tree traversal, such as in a depth-first search.
- 2. Indirect recursion: Most basic examples of recursion, and most of the examples presented here, demonstrate ***direct* recursion**, in which a function calls itself. *Indirect* recursion occurs when a function is called not by itself but by another function that it called (either directly or indirectly).
- 3. Anonymous recursion: Recursion is usually done by explicitly calling a function by name. However, recursion can also be done via implicitly calling a function based on the current context, which is particularly useful for anonymous functions, and is known as anonymous recursion.

RECURSIVE CLASSIC PROGRAMS

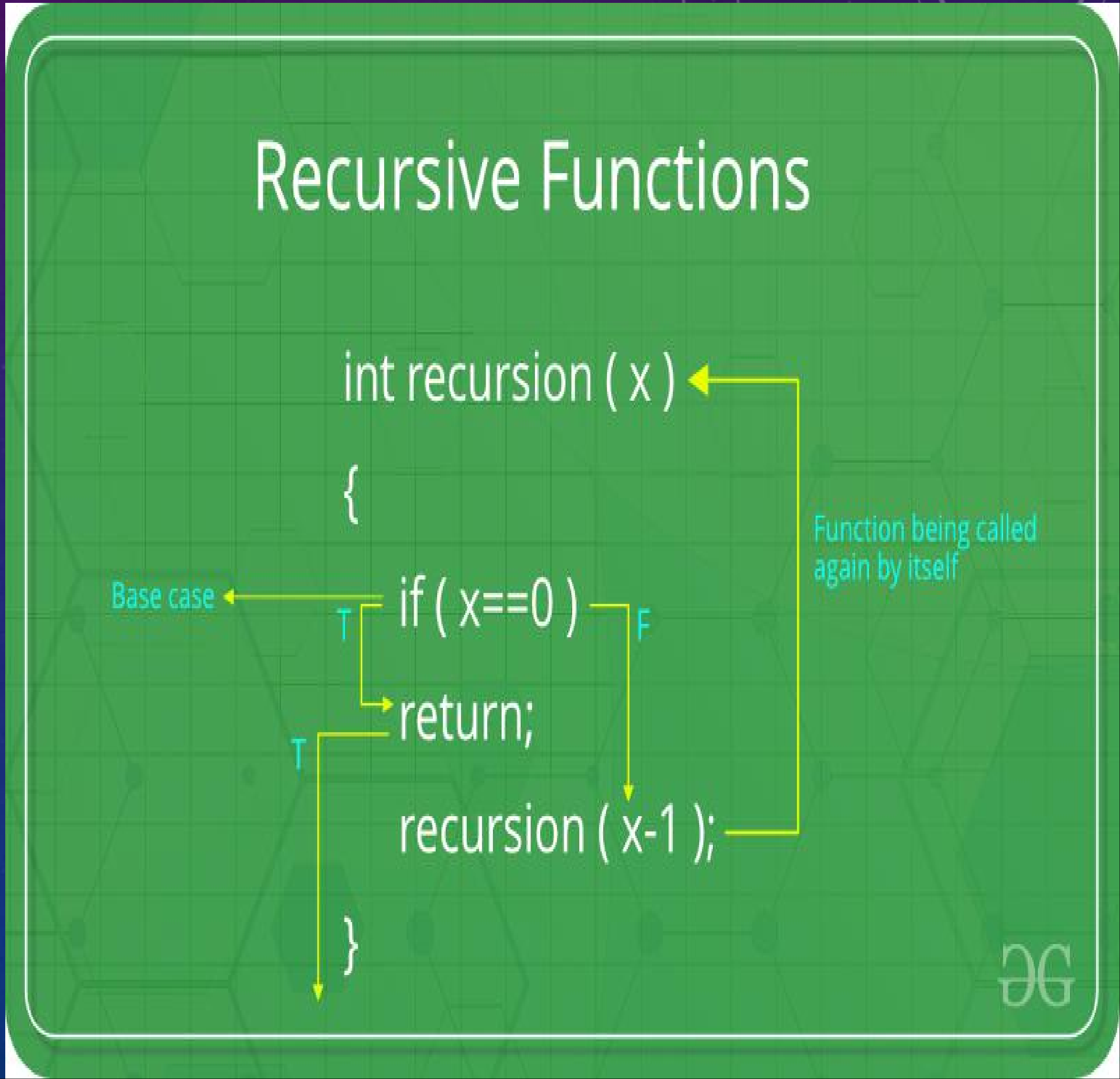
- A classic example of a recursive procedure is the function used to calculate the factorial of a natural number:



$$\mathrm{fact}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \cdot \mathrm{fact}(n-1) & \text{if } n > 0 \end{cases}$$

Pseudocode (recursive):

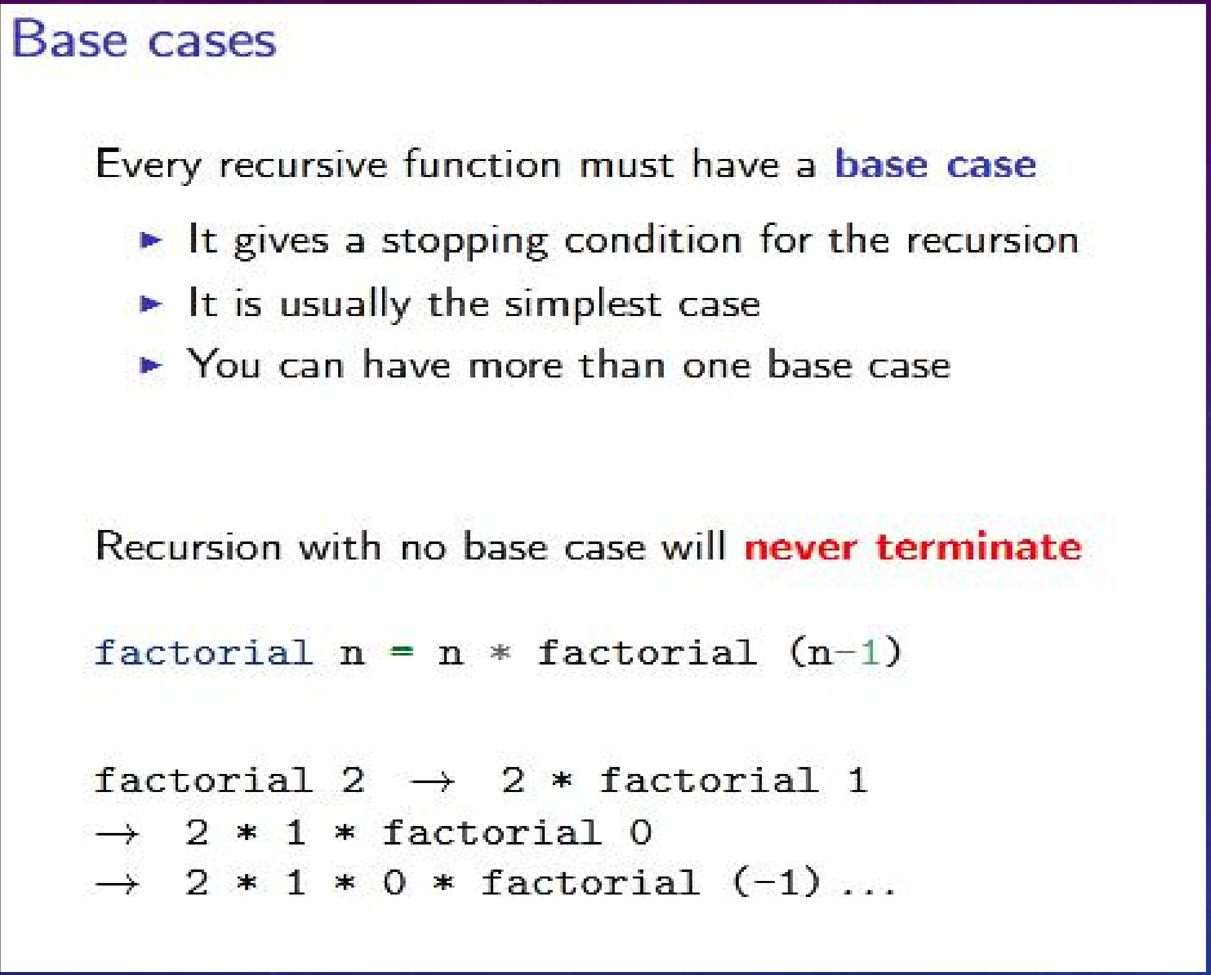
```
function factorial is:

input: integer n such that n >= 0

output: [n × (n-1) × (n-2) × … × 1]


    1. if n is 0, return 1
    2. otherwise, return [ n × factorial(n-1) ]


end factorial
```



Recursive Functions

```
int recursion ( x )
{
    if ( x==0 )
    return;
    recursion ( x-1 );
}
```

CHARACTERISTICS OF RECURSION



Base cases

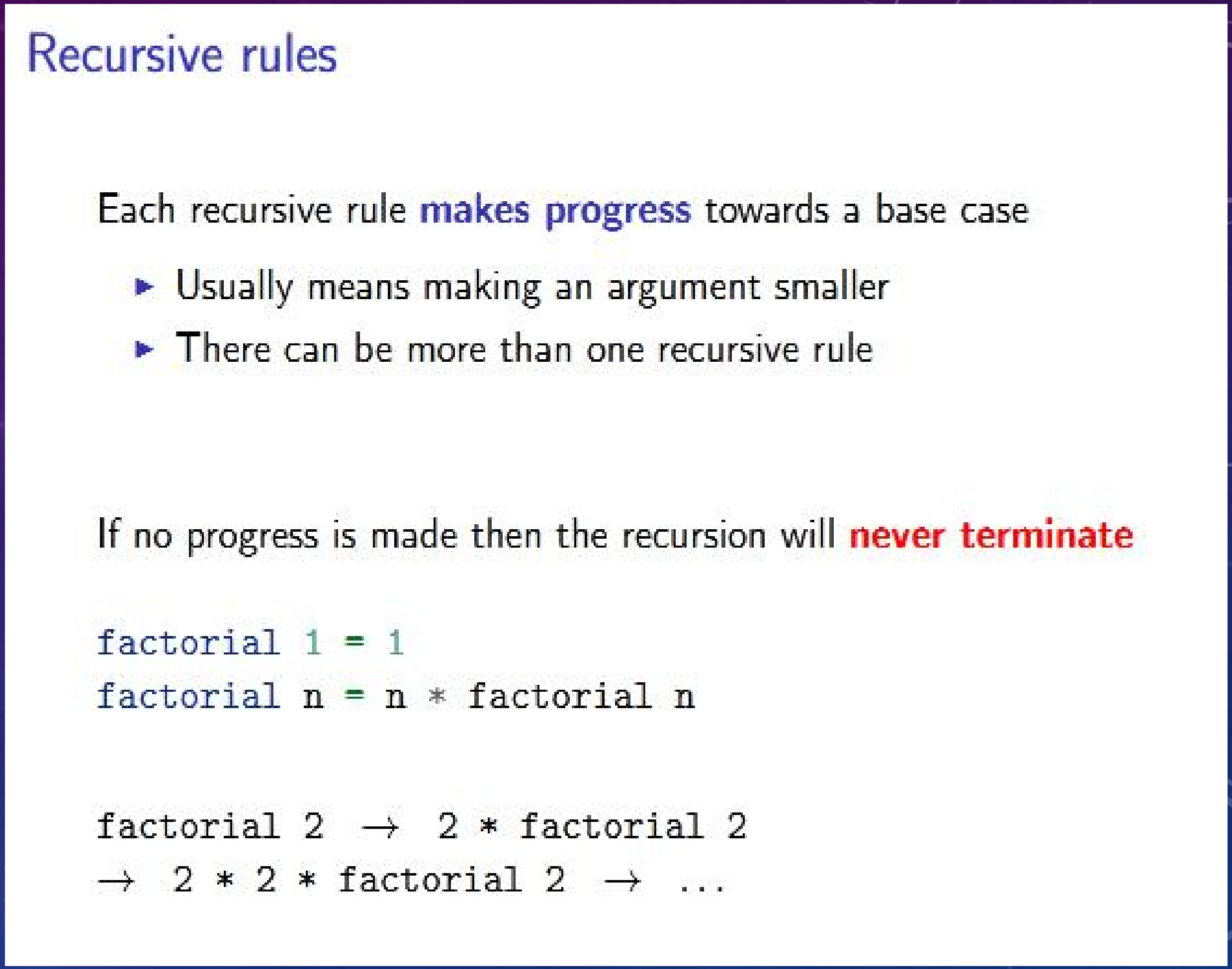
Every recursive function must have a **base case**

- It gives a stopping condition for the recursion
- It is usually the simplest case
- You can have more than one base case

Recursion with no base case will **never terminate**

```
factorial n = n * factorial (n-1)
```

```
factorial 2  →  2 * factorial 1
→  2 * 1 * factorial 0
→  2 * 1 * 0 * factorial (-1) ...
```



Recursive rules

Each recursive rule **makes progress** towards a base case

- Usually means making an argument smaller
- There can be more than one recursive rule

If no progress is made then the recursion will **never terminate**

```
factorial 1 = 1
factorial n = n * factorial n
```

```
factorial 2  →  2 * factorial 2
→  2 * 2 * factorial 2  →  ...
```

EXAMPLE: THE HANDSHAKE PROBLEM

There are n people in a room, and each person shakes hands once with every other person.

What is the total number h(n) of handshakes?

Recursive solutions seems very natural

2 persons : $h(2) = 1$

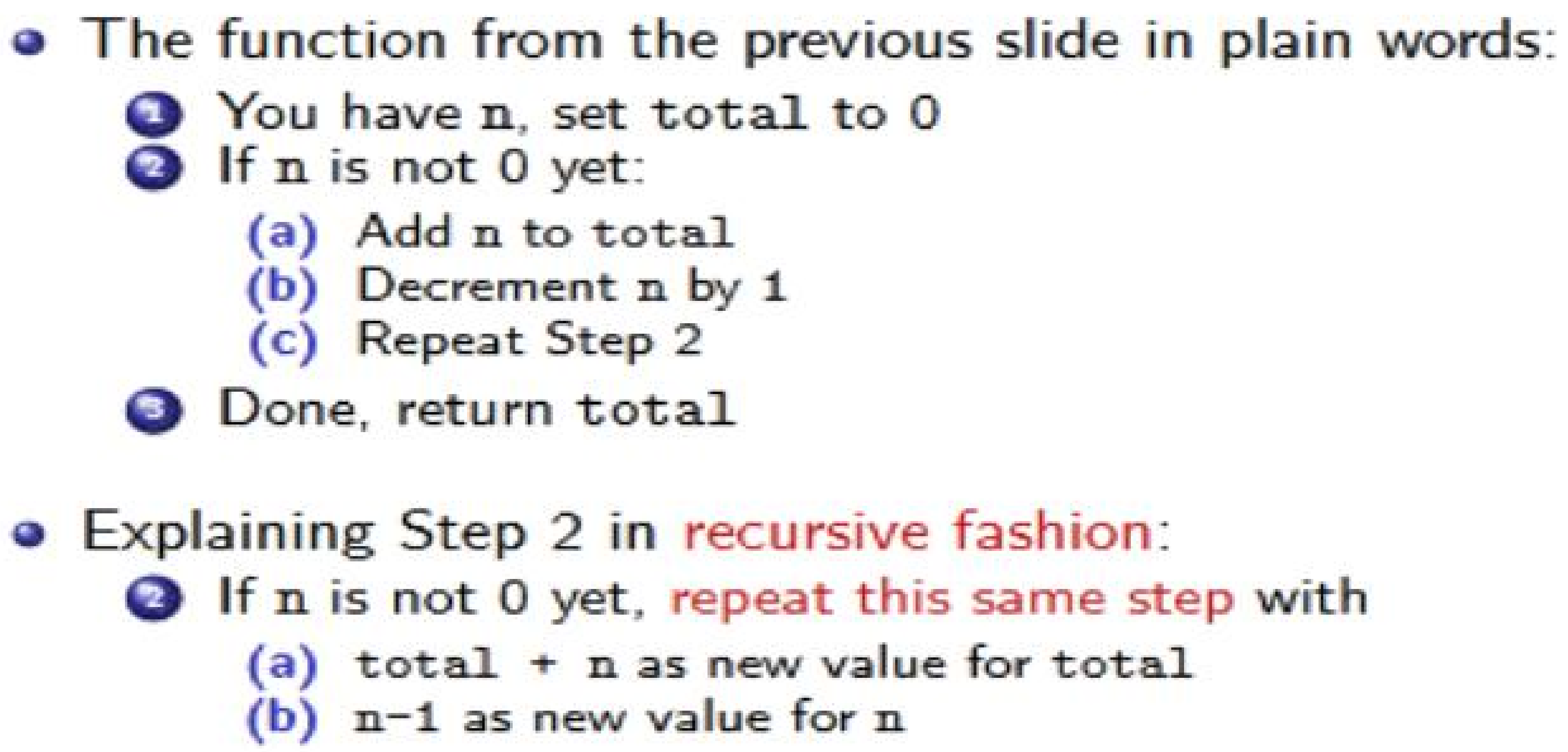
3 persons : $h(3) = h(2) + 2$

npersons:$h(n) = h(n-1) + (n-1)$

That is, then-th person shakesn−1 hands in addition to the number of handshakes of the previousn−1 people

Same as sum of $1 + 2 + \cdots + (n-1) = n \cdot (n-1)/2$

EXPLAINING RECURSION



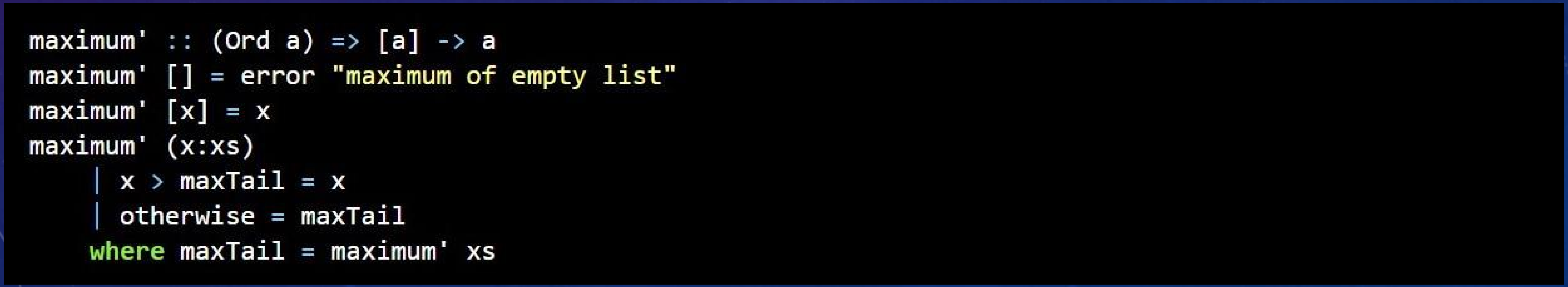
- The function from the previous slide in plain words:
  1. You have `n`, set `total` to 0
  2. If `n` is not 0 yet:
     - (a) Add `n` to `total`
     - (b) Decrement `n` by 1
     - (c) Repeat Step 2
  3. Done, return `total`
- Explaining Step 2 in recursive fashion:
  2. If `n` is not 0 yet, repeat this same step with
     - (a) `total + n` as new value for `total`
     - (b) `n-1` as new value for `n`

FIBONACCI NUMBERS WITH RECURSION

- Recursion is actually a way of defining functions in which the function is applied inside its own definition. Definitions in mathematics are often given recursively. For instance, the fibonacci sequence is defined recursively.
- First, we define the first two fibonacci numbers non-recursively. We say that F(0) = 0 and F(1) = 1, meaning that the 0th and 1st fibonacci numbers are 0 and 1, respectively. Then we say that for any other natural number, that fibonacci number is the sum of the previous two fibonacci numbers. So F(n) = F(n-1) + F(n-2). That way, F(3) is F(2) + F(1), which is (F(1) + F(0)) + F(1).
- Because we've now come down to only non-recursively defined fibonacci numbers, we can safely say that F(3) is 2. Having an element or two in a recursion definition defined non-recursively (like F(0) and F(1) here) is also called the **edge condition** and is important if you want your recursive function to terminate. If we hadn't defined F(0) and F(1) non recursively, you'd never get a solution any number because you'd reach 0 and then you'd go into negative numbers. All of a sudden, you'd be saying that F(-2000) is F(-2001) + F(-2002) and there still wouldn't be an end in sight!

MAXIMUM AWESOME

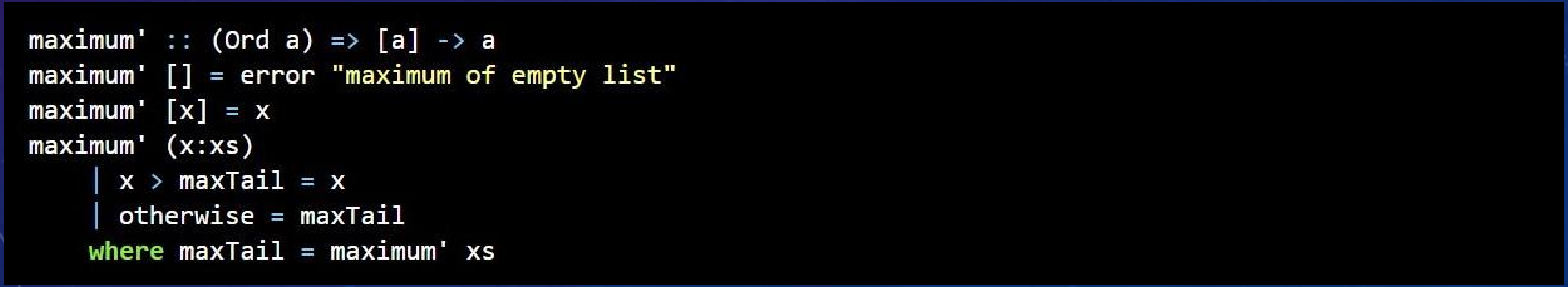
- The maximum function takes a list of things that can be ordered (e.g. instances of the Ord typeclass) and returns the biggest of them. Think about how you'd implement that in an imperative fashion. You'd probably set up a variable to hold the maximum value so far and then you'd loop through the elements of a list and if an element is bigger than then the current maximum value, you'd replace it with that element. The maximum value that remains at the end is the result. That's quite a lot of words to describe such a simple algorithm!
- Now let's see how we'd define it recursively. We could first set up an edge condition and say that the maximum of a singleton list is equal to the only element in it. Then we can say that the maximum of a longer list is the head if the head is bigger than the maximum of the tail. If the maximum of the tail is bigger, well, then it's the maximum of the tail. That's it! Now let's implement that in Haskell.



```haskell
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

MAXIMUM AWESOME

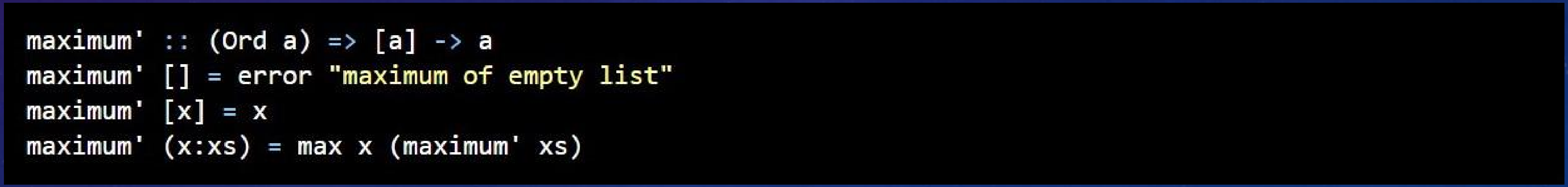
- As you can see, pattern matching goes great with recursion! Most imperative languages don't have pattern matching so you have to make a lot of if else statements to test for edge conditions. Here, we simply put them out as patterns. So the first edge condition says that if the list is empty, crash! Makes sense because what's the maximum of an empty list? I don't know. The second pattern also lays out an edge condition. It says that if it's the singleton list, just give back the only element.
- Now the third pattern is where the action happens. We use pattern matching to split a list into a head and a tail. This is a very common idiom when doing recursion with lists, so get used to it. We use a where binding to define maxTail as the maximum of the rest of the list. Then we check if the head is greater than the maximum of the rest of the list. If it is, we return the head. Otherwise, we return the maximum of the rest of the list.



```haskell
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs)
    | x > maxTail = x
    | otherwise = maxTail
    where maxTail = maximum' xs
```

MAXIMUM AWESOME

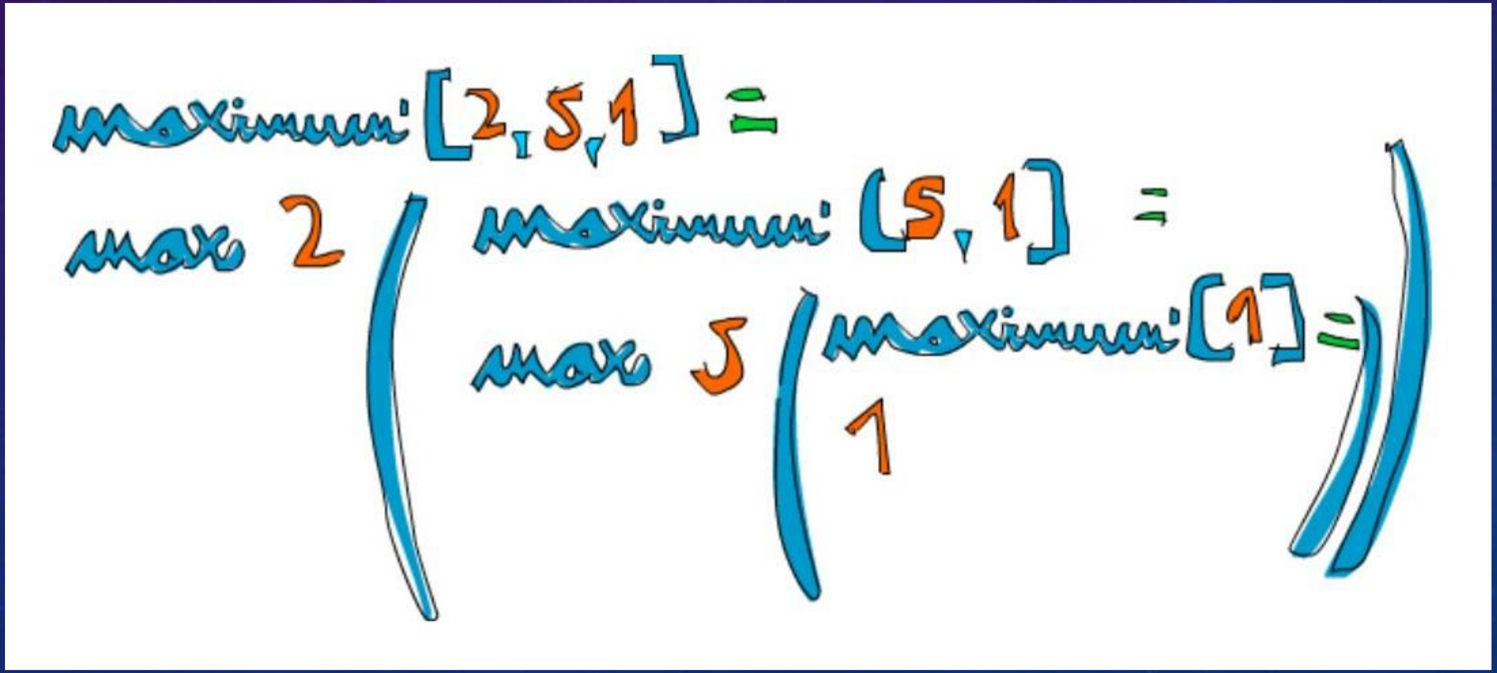
- Let's take an example list of numbers and check out how this would work on them: [2,5,1]. If we call maximum' on that, the first two patterns won't match. The third one will and the list is split into 2 and [5,1]. The where clause wants to know the maximum of [5,1], so we follow that route. It matches the third pattern again and [5,1] is split into 5 and [1]. Again, the where clause wants to know the maximum of [1]. Because that's the edge condition, it returns 1. Finally! So going up one step, comparing 5 to the maximum of [1] (which is 1), we obviously get back 5. So now we know that the maximum of [5,1] is 5. We go up one step again where we had 2 and [5,1]. Comparing 2 with the maximum of [5,1], which is 5, we choose 5.



```haskell
maximum' :: (Ord a) => [a] -> a
maximum' [] = error "maximum of empty list"
maximum' [x] = x
maximum' (x:xs) = max x (maximum' xs)
```

FUNCTION MAX

- An even clearer way to write this function is to use max. If you remember, max is a function that takes two numbers and returns the bigger of them.
- How's that for elegant! In essence, the maximum of a list is the max of the first element and the maximum of the tail.

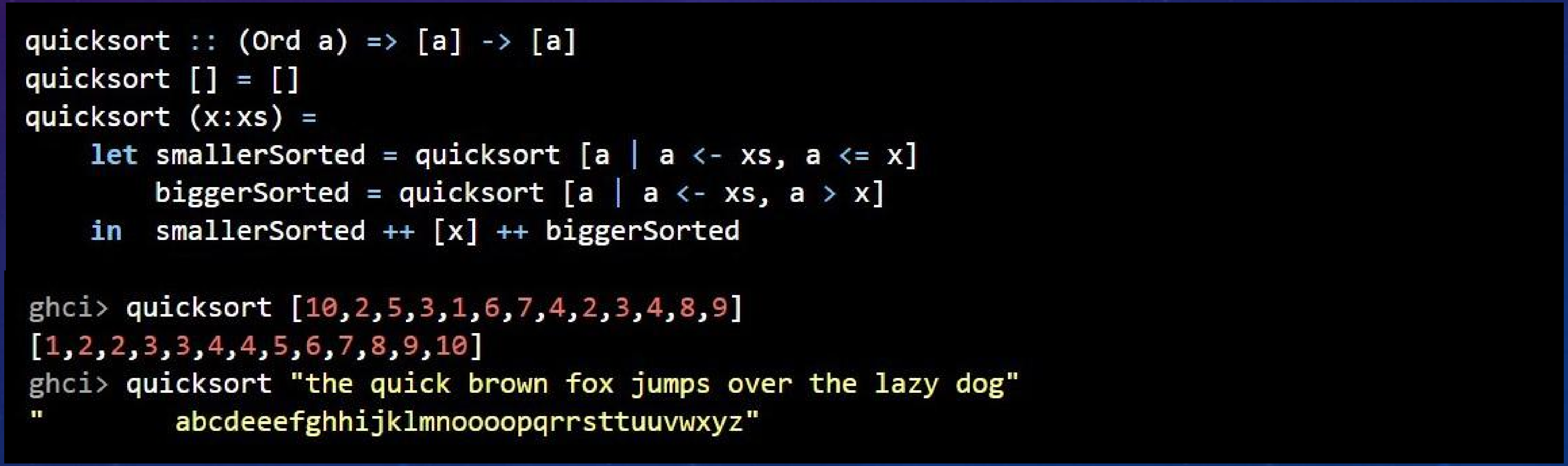


QUICK SORT!

- We have a list of items that can be sorted. Their type is an instance of the Ord typeclass. And now, we want to sort them! There's a very cool algoritm for sorting called quicksort. It's a very clever way of sorting items. While it takes upwards of 10 lines to implement quicksort in imperative languages, the implementation is much shorter and elegant in Haskell. Quicksort has become a sort of poster child for Haskell. Therefore, let's implement it here, even though implementing quicksort in Haskell is considered really cheesy because everyone does it to showcase how elegant Haskell is.
- So, the type signature is going to be quicksort :: (Ord a) => [a] -> [a]. No surprises there. The edge condition? Empty list, as is expected. A sorted empty list is an empty list. Now here comes the main algorithm: a sorted list is a list that has all the values smaller than (or equal to) the head of the list in front (and those values are sorted), then comes the head of the list in the middle and then come all the values that are bigger than the head (they're also sorted).

QUICK SORT!

- Notice that we said sorted two times in this definition, so we'll probably have to make the recursive call twice! Also notice that we defined it using the verb is to define the algorithm instead of saying do this, do that, then do that .... That's the beauty of functional programming! How are we going to filter the list so that we get only the elements smaller than the head of our list and only elements that are bigger? List comprehensions. So, let's dive in and define this function.

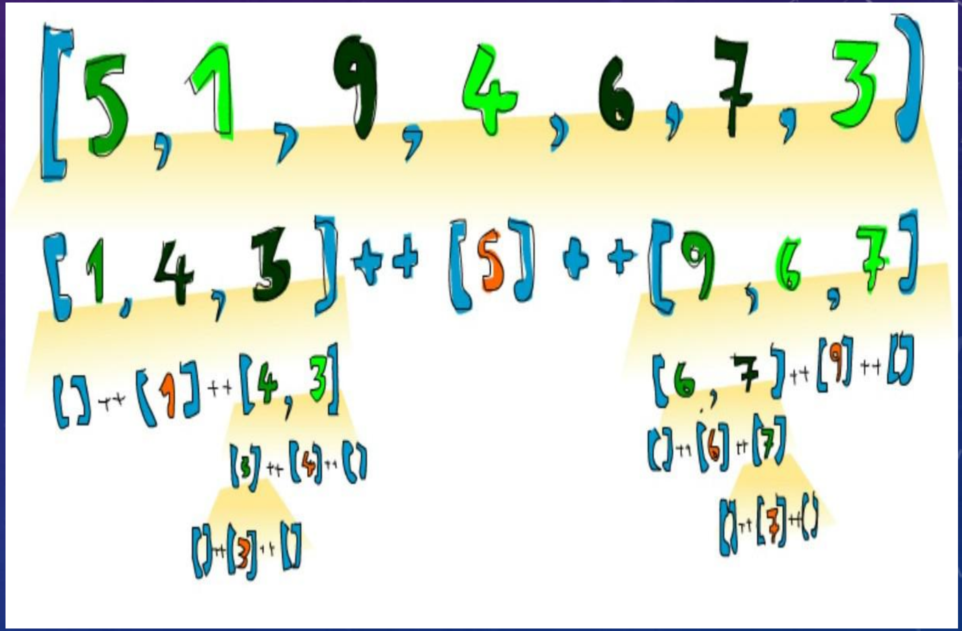


```haskell
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) =
    let smallerSorted = quicksort [a | a <- xs, a <= x]
        biggerSorted = quicksort [a | a <- xs, a > x]
    in  smallerSorted ++ [x] ++ biggerSorted
```

```
ghci> quicksort [10,2,5,3,1,6,7,4,2,3,4,8,9]
[1,2,2,3,3,4,4,5,6,7,8,9,10]
ghci> quicksort "the quick brown fox jumps over the lazy dog"
"        abcdeeefghhijklmnoooopqrrsttuuvwxyz"
```

QUICK SORT!

- So if we have, say [5,1,9,4,6,7,3] and we want to sort it, this algorithm will first take the head, which is 5 and then put it in the middle of two lists that are smaller and bigger than it. So at one point, you'll have [1,4,3] ++ [5] ++ [9,6,7]. We know that once the list is sorted completely, the number 5 will stay in the fourth place since there are 3 numbers lower than it and 3 numbers higher than it. Now, if we sort [1,4,3] and [9,6,7], we have a sorted list! We sort the two lists using the same function. Eventually, we'll break it up so much that we reach empty lists and an empty list is already sorted in a way, by virtue of being empty.



CONCLUSION

- Recursion is important to Haskell because unlike imperative languages, you do computations in Haskell by declaring what something is instead of declaring how you get it. That's why there are no while loops or for loops in Haskell and instead we many times have to use recursion to declare what something is.
- Apart from these examples, recursion is also used in solving some standard problems like traversals (inorder/preorder/postorder), towers of Hanoi, BFS traversal, etc.

THANK YOU FOR ATTENTION