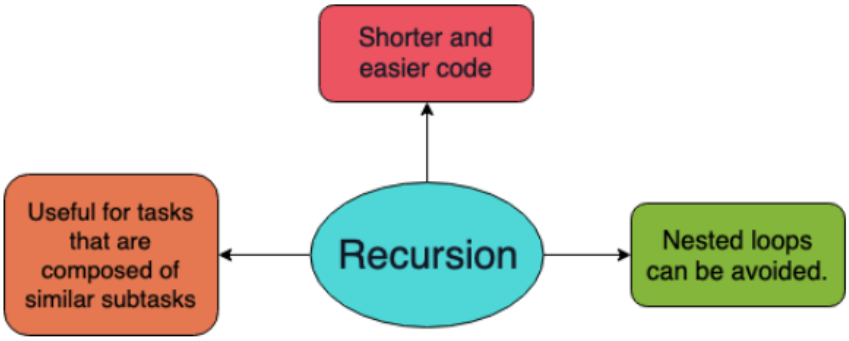
Recursion

In computer science, recursion occurs when a function calls itself within its declaration.

It is not an algorithm or pattern, it is more of a concept to write more efficient algorithms using various data structures (e.g., arrays, linked lists, trees, etc.) and other algorithms and patterns.



Why use recursion?

In general, the recursive code helps us avoid complex nested loops and is most useful for tasks that can be defined in terms of similar subtasks.

Recursion has two fundamental aspects: the **base case** and the **recursive step**.

When using iteration, we rely on a counting variable and a Boolean condition.

For example, when iterating through the values in a list, we would increment the counting variable until it exceeded the length of the dataset.

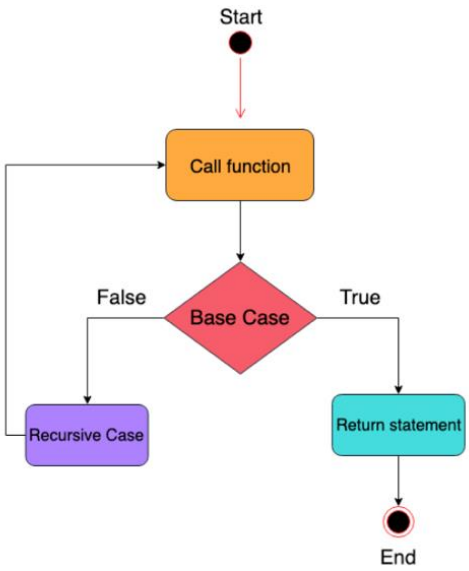
Recursive functions have a similar concept, which we call the **base case**. The base case dictates whether the function will recurse, or call itself. Without a base case, it's the iterative equivalent to writing an infinite loop.

Because we're using a call stack to track the function calls, your computer will throw an error due to a *stack overflow* if the base case is not sufficient.

Format of a Recursive Function

Each recursive function consists of 2 parts:

- **Base Case**: The base case is where further calls to the same function stop, i.e, it does not make any subsequent recursive calls.
- **Recursive Step**: The recursive case is where the function calls itself again and again until it reaches the base case.



In a recursive program, the solution to the bigger problem is expressed in terms of the smaller problems, until the smallest problem reaches the base case.

```
function RecursiveFunction() {
  // Base Case
  if (<baseCaseCondition>) {
    return <some value>;
  }
  // Recursive Case
 else {
    return <recursive call and any other task>;
 }
}
```

Since a recursive approach requires the function invoking itself with different arguments there needs to be a way to track the multiple function invocations with different arguments.

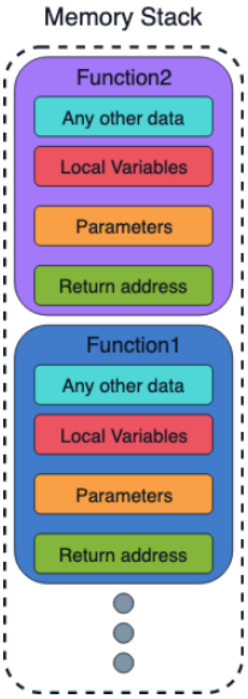
The **call stack** and **execution contexts** make this possible.

Stacks are regions of memory where data is added or removed in a last-in-first-out (LIFO) manner. Each program has a reserved region of memory referred to as its stack.

The execution context is memory allocated for each function to store the variables needed in a function. There is a function execution context and a global execution context.

So, for each recursive call, a new function execution context is created to store the data need for that function invocation to execute.

Memory allocation in recursion



When a function is called, its memory is allocated on the **stack**. When a function executes, it adds its state data to the top of the stack; when the function exits this data is removed from the stack.

A recursive function calls itself, therefore, the memory for a called function is allocated on top of memory allocated for calling function.

Remember, a different copy of local variables is created for each function call. When the base case is reached, the child function returns its value to the function from which it was called. Then, this child function's stack frame is removed.

Stack overflow error in recursion

When we call a recursive function, the return address and arguments are pushed onto the call stack. The stack is finite, so if the recursion is too deep, you'll eventually run out of stack space. This is also called the stack overflow in recursion.

Popular reasons for stack overflow error:

- Missing a base case.
- Incorrect base case function call.
- Recursive function needs multiple base cases.

Recognizing a recursive problem

Most of the recursive problems used in interviews are famous, so we recognize them by name.

For example, problems such as Fibonacci numbers, summing a list of numbers, greatest common divisor, the factorial of a number, recursive Binary Search, reversing a string, and so on are well-known recursive problems.

But what do all these problems have in common?
Answer: All these problems can be built off of **sub-problems**. In other words, we say that we can express the value returned by a method in terms of other values returned by that method.

Typically, such problems include the words list top/last n ..., compute the nth or all..., count/find all solutions that ..., generate all cases that ..., and so on.

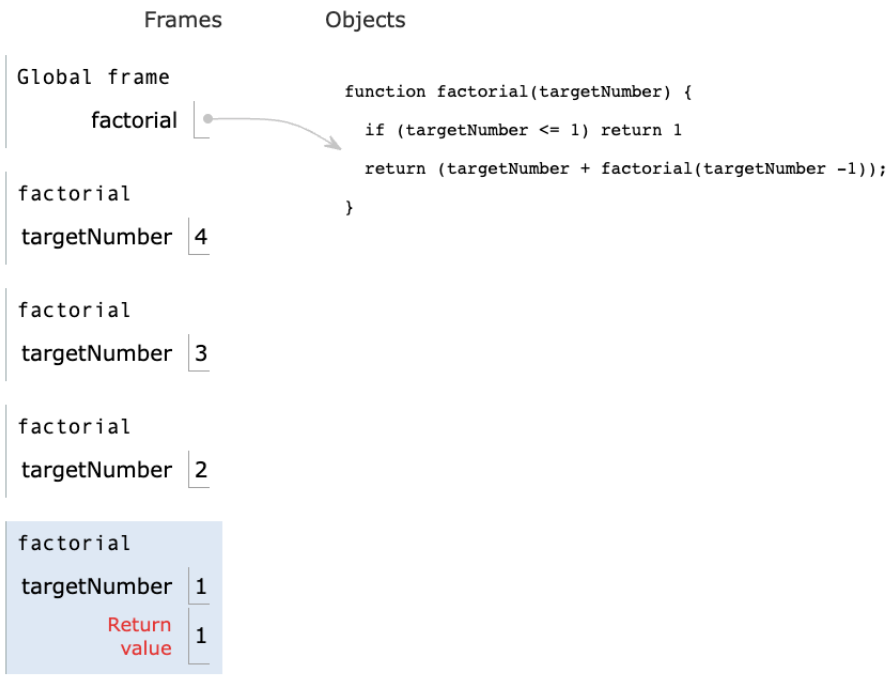
Calculating the factorial of a number

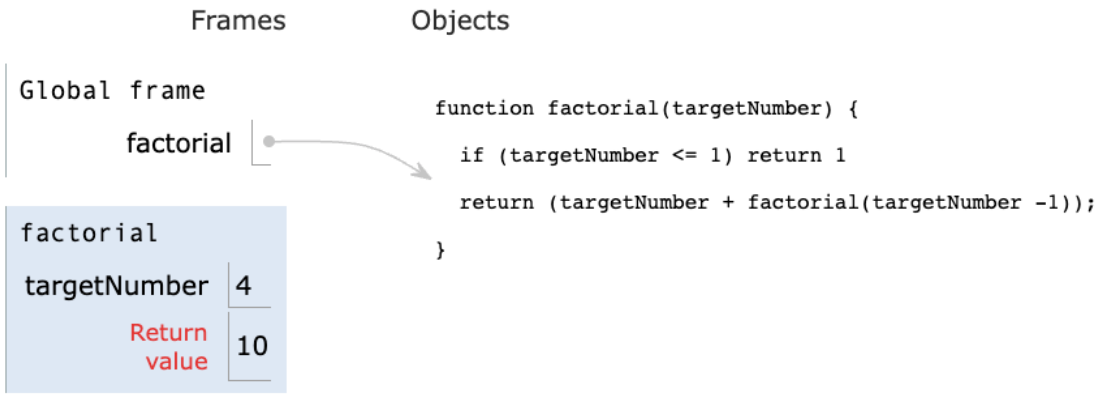
A common recursive algorithm is calculating the factorial of a number. A factorial is the product of an integer and all the positive integers less than it.

For example, 4! (factorial) is 4x3x2x1 = 24; (12x2x1)

Here, the smallest starting value is 1! Therefore, this is our **base case**. So, the function input reaches 1, we will reach our base case and exit the function returning **10**.

```
1  function factorial(targetNumber) {
2
3    if (targetNumber <= 1) return 1
4
5    return (targetNumber + factorial(targetNumber -1));
6
7  }
8
9  console.log(factorial(4))
```





Note: Use the [JavaScript coding tutor] to visualize your code.

Head versus tail recursion

There are two kinds of recursion: head and tail recursion. The difference is the position of the recursive call:

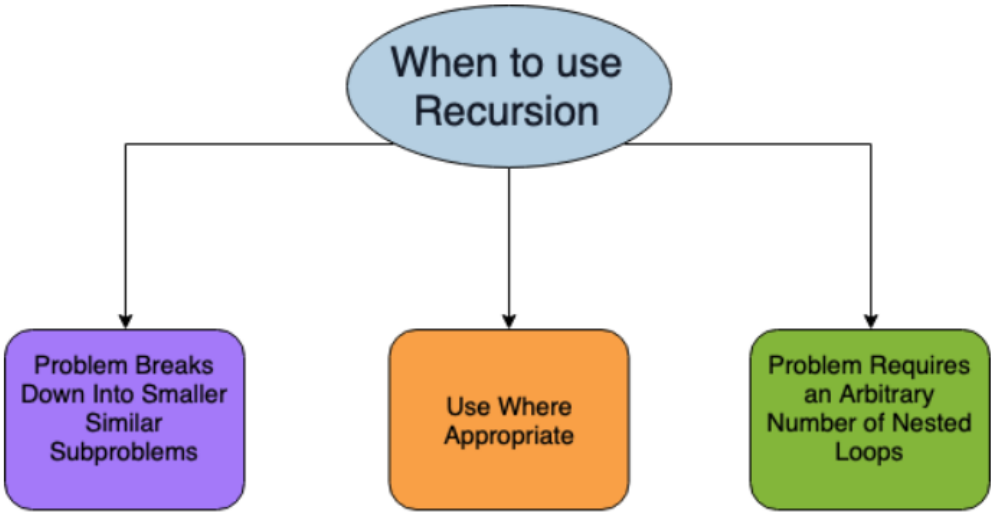
Head recursion

The recursive call is **not** the function' last statement, and other statements/expressions are executed/evaluated after it.

Tail recursion

All non-recursive expressions are evaluated before the recursive call. Generally, this is the most common version of recursion.

When to use recursion



In general, problems that can be solved with a non-recursive code can also be solved by recursion. However, some problems can only be solved recursively.

- Recursion should be used where you feel like it is appropriate, or it just feels natural.
- When a problem can be broken down into smaller subproblems.
- Have to nest an arbitrary number of loops, which may make your code complicated. Such problems can be more easily solved using recursion.

When not to use recursion

Which to choose — recursion or iteration — depends highly on the problem you want to solve and in which environment your code runs.

Recursion is often the preferred tool for solving more abstract problems, and iteration is preferred for more low-level code. Iteration might provide better runtime performance, but recursion can improve your productivity as an engineer.

	Recursion	Iteration
Implementation	Self-calling function	Loop
State	Stored on Stack	control variables (.e.g. a loop index)
Progress	Towards base condition	Towards control value condition
Termination	Base condition reached	Control variable condition reached
Verbosity	↘	↗
If not terminated	`StackOverflowError`	endless loop
Overhead	↗	↘
Performance	↘	↗