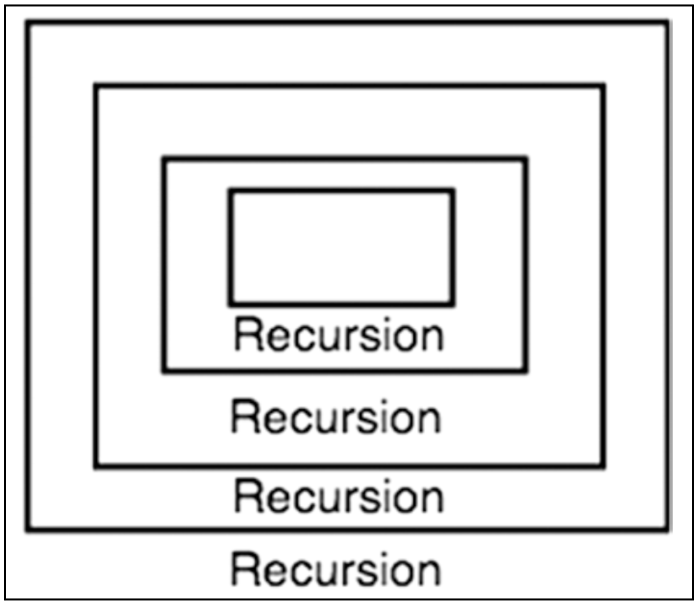
Basic Recursion

What is recursion?

Recursion is a way by which any function can call itself which is called a recursive function. A recursive method solves a problem by calling a copy of itself to work on a smaller problem. This is called the **recursion step**. The recursion step can result in many more such recursive calls. It is important to ensure that the recursion terminates. Each time the function calls itself with a slightly simpler version of the original problem. The sequence of smaller problems must eventually converge on the **base case**. Recursion is important because you will see it again and again in the implementation of various data structures.



Important points about recursion:-

1. Recursive algorithms have two types of cases, recursive cases and base cases.
2. Every recursive function case must terminate at a base case.
3. Generally, iterative solutions are more efficient than recursive solutions [due to the overhead of function calls].
4. A recursive algorithm can be implemented without recursive function calls using a stack, but it’s usually more trouble than it's worth. That means any problem that can be solved recursively can also be solved iteratively.
5. For some problems, there are no obvious iterative algorithms.

6. Some problems are best suited for recursive solutions while others are not.

Let’s see the following function, which prints numbers counting down from n to 0 as an example:

```
function countDownToZero(n)
{ // base case. Stop at 0
if (n < 0) {
return; // stop the function }
else {
console.log(n);
countDownToZero(n - 1); // count down 1
}
}
countDownToZero(12);
```

countDownToZero is a **Recursive Function** where at each step it is calling itself but with a lesser value of n. To end this infinite loop we have to define a base case which terminates the loop, the base case in this example is **n < 0.** Now, let’s look at the recursive case which is in the else part where we are printing the output and reducing the value of n by 1 to the next call of the function.If a negative number is given as the input, it will not print that number because of the base case. In addition to a base case, this recursive function also exhibits the divide-and-conquer method.

Divide-and-Conquer Method

In computer science, the divide-and-conquer method is when a problem is solved by solving all of its smaller sub-problems. With the countdown example, counting down from 2 can be solved by printing 2 and then counting down from 1. Here, counting down from 1 is the part solved by “dividing and conquering.” It is necessary to make the problem smaller to reach the base case.

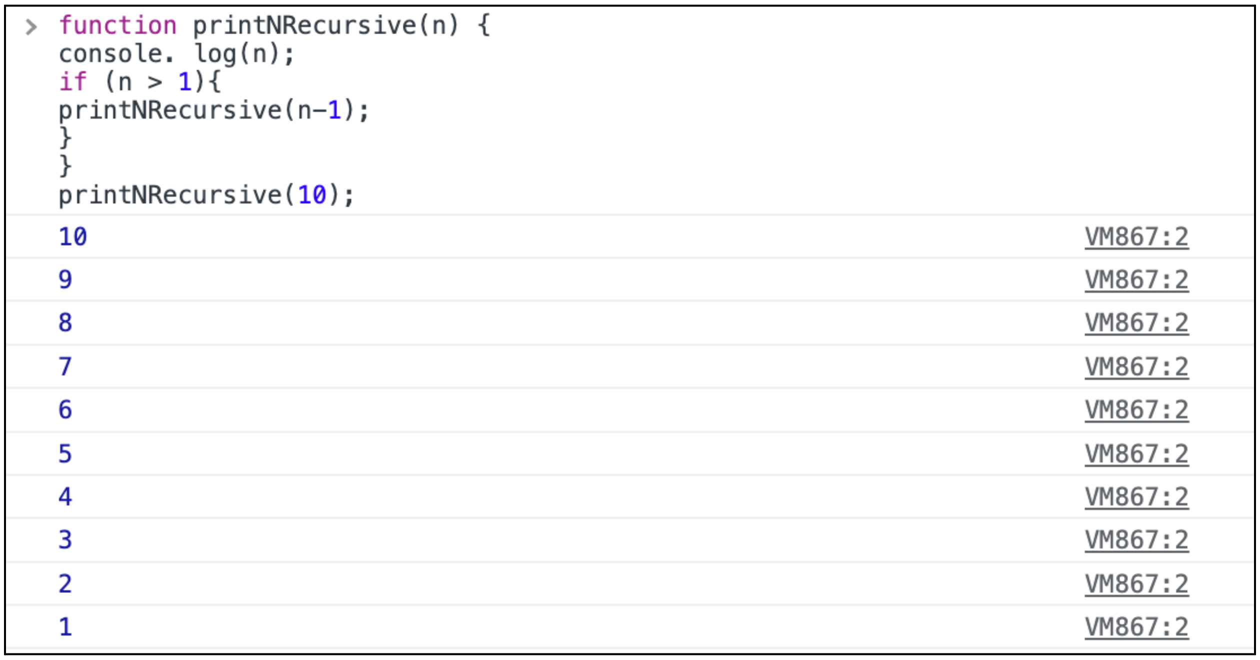
What will happen if we do not put a correct base case?
If the recursive call does not converge to a base case, a stack overflow occurs. Each call of our recursive function is stored in a recursive stack and this stack calls can only be upto certain limit after which the stack is filled completely and no more calls can be made due to which stack overflow occurs.

Recursive Call Stack Memory

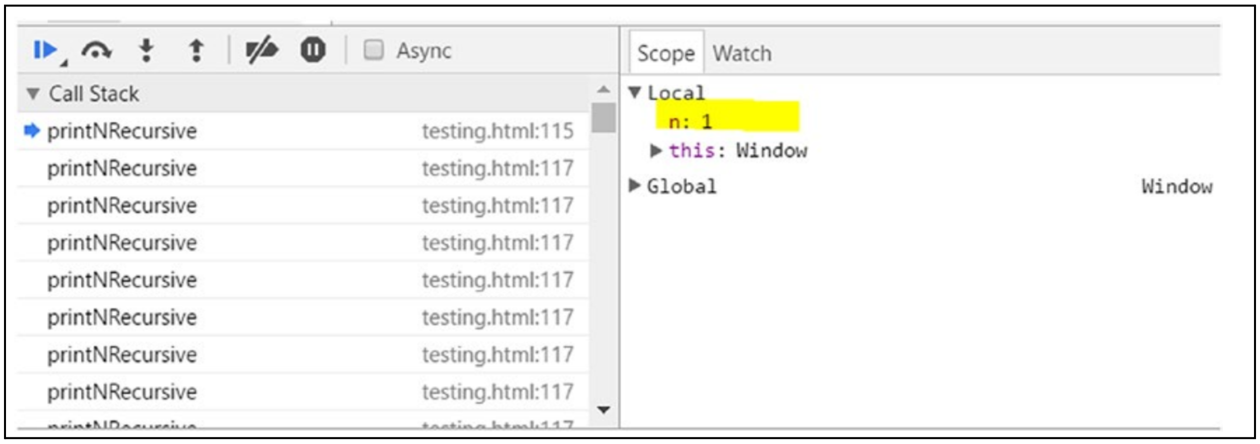
When a recursive function calls itself, that takes up memory, and this is really important in Big-O space complexity analysis.
For example, this simple function for printing from n to 1 recursively takes O(n) in space:

```
function printNRecursive(n) {
console. log(n);
if (n > 1){
printNRecursive(n-1);
}
}
printNRecursive(10);
```

You can run this on any browser or any JavaScript engine and will see the result as shown in the figure.



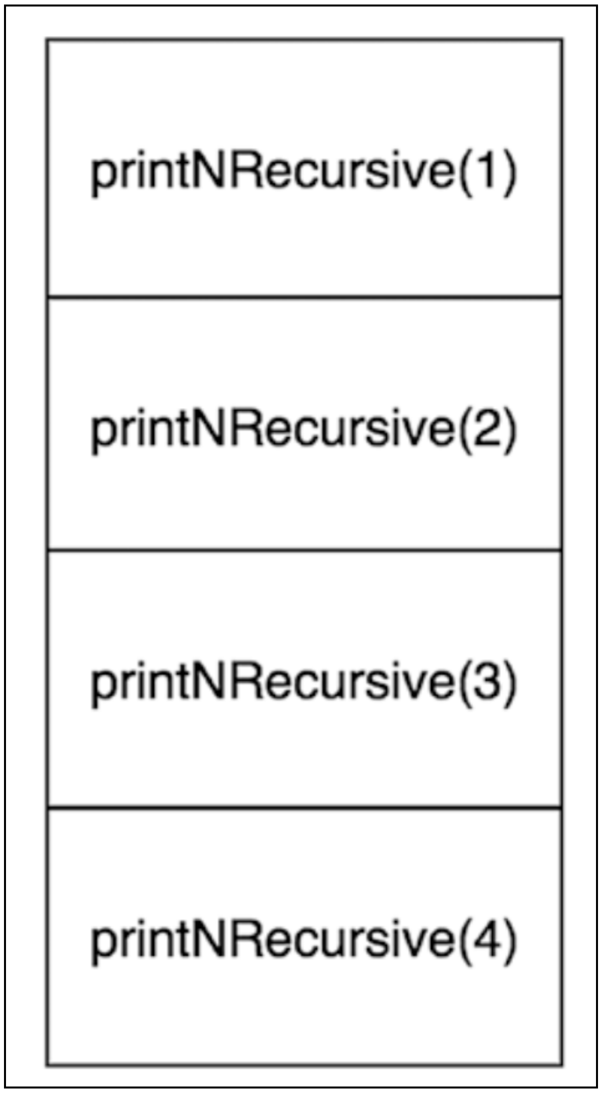
```
> function printNRecursive(n) {
  console. log(n);
  if (n > 1){
  printNRecursive(n-1);
  }
  }
  printNRecursive(10);
  10                                                    VM867:2
  9                                                     VM867:2
  8                                                     VM867:2
  7                                                     VM867:2
  6                                                     VM867:2
  5                                                     VM867:2
  4                                                     VM867:2
  3                                                     VM867:2
  2                                                     VM867:2
  1                                                     VM867:2
```



```
Call Stack
  printNRecursive          testing.html:115
  printNRecursive          testing.html:117
  printNRecursive          testing.html:117
  printNRecursive          testing.html:117
  printNRecursive          testing.html:117
  printNRecursive          testing.html:117
  printNRecursive          testing.html:117
  printNRecursive          testing.html:117

Scope  Watch
▼ Local
    n: 1
  ▶ this: Window
▶ Global                   Window
```

is



As shown in above figures, each recursive call must be stored in memory until the base case is resolved. Recursive algorithms take extra memory because of the call stack.

Note:- Recursive functions have an additional space complexity cost that comes from the recursive calls that need to be stored in the operating system's memory stack. The stack is accumulated until the base case is solved. In fact, this often why an iterative solution may be preferred over the recursive solution. In the worst case, if the base case is implemented incorrectly, the recursive function will cause the program to crash because of a stack overflow error that occurs when there are more than the allowed number of elements in the memory stack.

Let's now see another example of the Fibonacci sequence.

The Fibonacci sequence is a list of infinite numbers, each of which is the sum of the past two terms (starting with 1).
1, 1, 2, 3, 5, 8, 13, 21

```
function getNthFibo(n) {
if (n <= 1) {
return n;
} else {
return getNthFibo(n - 1) + getNthFibo(n - 2);
}
}
```

Base case: The base case for the Fibonacci sequence is that the first element is 1.
Divide and conquer: By definition of the Fibonacci sequence, the Nth Fibonacci number is the **sum of the (n-1)th and (n-2)th Fibonacci numbers**. This implementation has a **time complexity of O(2^n)**.

Master Theorem

The master theorem states the following:

Given a recurrence relation of the form **T(n)=aT(n/b)+O(n^c)** where a>=1 and b>=1, there are three cases.

a is the coefficient that is multiplied by the recursive call. b is the "logarithmic" term, which is the term that divides the n during the recursive call. Finally, c is the polynomial term on the nonrecursive component of the equation.

The first case is when the polynomial term on the non recursive component O(n^c) is smaller than log_b(a).

Case 1: $c<\log_b(a)$ then $T(n)=O(n((\log_b(a)))$).

For example, $T(n)=8T(n/2)+1000n^2$

Identify a,b,c: a=8,b=2,c=2

Evaluate: $\log_2(8)=3$.$c<3$ is satisfied.

Result: $T(n)=O(n^3)$

The second case is when c is equal to $\log_b(a)$.

Case 2: $c = \log_b(a)$ then $T(n)=O(n^c\log(n))$.

For example, T(n)=2T(n/2)+10n.

Identify a,b,c:a=2,b=2,c=1

Evaluate: $\log_2(2)=1$,c=1 is satisfied.

Result: $T(n)=O(n^c\log(n))=T(n)=O(n^1\log(n))=T(n)=O(n\log(n))$

The third and final case is when c is greater than $\log_b(a)$.

Case 3: $c>\log_b(a)$ then $T(n)=O(f(n))$

For example, $T(n)=2T(n/2)+n^2$.

Identify a,b,c: $a=2,b=2,c=2$

Evaluate: $\log_2(2)=1, c>1$ is satisfied.

Result: $T(n)=f(n)=O(n^2)$

This section covered a lot about analyzing the time complexity of recursive algorithms. Space complexity analysis is just as important. The memory used by recursive function calls should also be noted and analyzed for space complexity analysis.

Summary

Recursion is a powerful tool to implement complex algorithms. Recall that all recursive functions consist of two parts: the base case and the divide-and-conquer method (solving subproblems).

Analyzing the Big-O of these recursive algorithms can be done empirically (not recommended) or by using the master theorem. Recall that the master theorem needs the recurrence relation in the following form: $T(n) = aT(n/b) + O(n^c)$. When using the master theorem, identify a, b, and c to determine which of the three cases of the master theorem it belongs to.

Finally, when implementing and analyzing recursive algorithms, consider the additional memory caused by the call stack of the recursive function calls. Each recursive call requires a place in the call stack at runtime; when the call stack accumulates n calls, then the space complexity of the function is $O(n)$.