# Tail Call Optimization

**We'll cover the following**

## What is tail call optimization?

Consider the code you write to be a procedure and the generated bytecode that will eventually run to be a process. The `factorialIterative()` function is an iterative procedure and is compiled into and will run as an iterative process—no surprise there. Likewise, `factorialRec()` is a recursive procedure and is compiled into, and run as, a recursive process, exactly what we'd expect there as well. However, the real gain, as explained in Structure and Interpretation of Computer Programs, is when a recursive procedure can be compiled into an iterative process. This approach will bring the best of both worlds—the code can be expressed as recursion, but it can enjoy the runtime behavior of iterations. So, no stack overflow errors.

That's intriguing—compiling recursion into iteration—that's exactly what the `tailrec` annotation will instruct the Kotlin compiler to do. Let's rewrite the `factorialRec()` function to make use of that technique.

## Using `tailrec`

As a first step, we'll annotate the function with the `tailrec` keyword.

```
tailrec fun factorialRec(n: Int): BigInteger =
  if (n <= 0) 1.toBigInteger() else n.toBigInteger() * factorialRec(n - 1)
```

Good try, but that doesn't work—we need to take a few more steps.

```
120
```

```
recursivetail.kts:4:1: warning: a function is marked as tail-recursive
 but no tail calls are found

tailrec fun factorialRec(n: Int): BigInteger =
^
recursivetail.kts:5:56: warning: recursive call is not a tail call
  if (n <= 0) 1.toBigInteger() else n.toBigInteger() * factorialRec(n - 1)
                                                       ^
```

Kotlin is too polite here and gives a warning, but remember to treat warnings as errors as discussed in Sensible Warnings. If we try to run the function for large inputs, this version's fate will be the same as the older version of `factorialRec()`, a runtime error. Kotlin will optimize recursion to iteration only when the call is in tail position. Let's discuss that further.

Looking at the code `n.toBigInteger() * factorialRec(n - 1)`, we may be tempted to think that `factorialRec()` is executed last, but it's the multiplication operation that kicks in before the function call returns. This operation waits for a call to `factorialRec()` to finish, thus increasing the stack size on each recursive call. A *tail call* is where the recursive call is truly the last operation in the function. Let's rewrite the function `factorialRec()` and rename it as `factorial()` along the way.

```
import java.math.BigInteger

tailrec fun factorial(n: Int,
  result: BigInteger = 1.toBigInteger()): BigInteger =
    if (n <= 0) result else factorial(n - 1, result * n.toBigInteger())

println(factorial(5)) //120
```

factorial.kts

Running this code produces the expected result, and there are no warnings. That tells us that the Kotlin compiler was happy to optimize the function into an iteration, quietly behind the scenes.

Exercise the function for a large input value:

```
println(factorial(50000)) //No worries
```

The code will run just fine and produce, as expected, a very large output.

## Behind the scene #

The successful execution is a proof of optimization, but as a curious programmer you may want to explicitly see the effect rather than assume based on empirical results. That's fair—to see the optimization at work, let's create a Factorial class and place the recursive version and the tail recursive version of factorial into it:

```
import java.math.BigInteger

object Factorial {
  fun factorialRec(n: Int): BigInteger =
    if (n <= 0) 1.toBigInteger() else n.toBigInteger() * factorialRec(n - 1)

  tailrec fun factorial(n: Int,
    result: BigInteger = 1.toBigInteger()): BigInteger =
      if (n <= 0) result else factorial(n - 1, result * n.toBigInteger())
}
```

Factorial.kt

Now to compile the code locally and view the generated bytecode use these commands:

```
kotlinc-jvm Factorial.kt
javap -c -p Factorial.class
```

An excerpt of the generated bytecode is shown here:

```
Compiled from "Factorial.kt" public final class Factorial {
public final java.math.BigInteger factorialRec(int); Code:
...
      38: invokevirtual #23
                 // Method factorialRec:(I)Ljava/math/BigInteger;
...
44: invokevirtual #27
// Method java/math/BigInteger.multiply:(...)
47: dup
48: ldc
#29 // String this.multiply(other)
...
public final java.math.BigInteger factorial(int, java.math.BigInteger); Code:
...
       7: ifgt          14
      10: aload_2
      11: goto          76
...
```

```
56: invokevirtual #27
// Method java/math/BigInteger.multiply:(...) ...

      73: goto          0
      76: areturn
```

In the bytecode for `factorialRec()`, the bytecode instruction `invokevirtual` is used to recursively call `factorialRec()`, and then a call to `multiply()` on `BigInteger` follows. That shows that the recursive procedure has been compiled into a recursive process.

On the other hand, the bytecode for `factorial()` doesn't have any `invokevirtual` recursive calls. Instead, it has calls to `ifgt` and `goto` that jump around to different parts of the function. That's evidence that the recursive procedure was compiled to an iterative process—good job, Kotlin.

The `tailrec` optimization works only for recursions that can be expressed as tail recursion. To use `tailrec`, we had to rewrite `factorialRec()` as `factorial()` so the recursion would appear as the last expression that's evaluated. If the recursion is complex, that may not be easy or even possible.

QUIZ

Q Can the `tailrec` optimization be used for any recursive problem?

Retake Quiz

Tail call optimization keeps the number of levels in the stack under control by converting the recursion to iteration. That has an impact on efficiency, but we can make execution faster by returning a stored value rather than repetitively calling functions. That’s the solution we’ll explore in the next lesson.