The Power and Perils of Recursion

We'll cover the following

- Recursion with divide and conquer
- Recursive vs. iterative
- Drawback of recursion

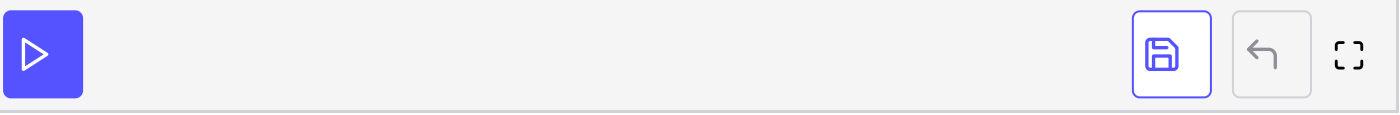
Recursion with divide and conquer

Using recursion we can apply the divide and conquer technique: solve a problem by implementing solutions to its subproblems. For example, here's a piece of Kotlin code to perform one implementation of the *quick sort* algorithm:

```
fun sort(numbers: List<Int>): List<Int> =
  if (numbers.isEmpty())
    numbers
  else {
    val pivot = numbers.first()
    val tail = numbers.drop(1)
    val lessOrEqual = tail.filter { e -> e <= pivot }
    val larger = tail.filter { e -> e > pivot }

    sort(lessOrEqual) + pivot + sort(larger)
  }

println(sort(listOf(12, 5, 15, 12, 8, 19))) //[5, 8, 12, 12, 15, 19]
```



quicksort.kts

The `sort()` function splits the given input into two parts, sorts the two parts separately, and, finally, merges the two solutions to create the overall solution. Kotlin readily supports general recursion, but it requires the return type for recursive functions—no type inference available for these.

Recursive vs. iterative

Recursion is highly expressive, though it takes a bit more effort to arrive at recursive solutions. Programmers in general, and beginners in particular, have trouble conceptualizing solutions recursively. But once it clicks, the joy is boundless. Let's look at some very simple recursive code:

```
import java.math.BigInteger

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

println(factorialRec(5)) //120
```

recursive.kts

The `factorialRec()` function returns `1` if the value given is zero or less. Otherwise it returns the product of the given input and a recursive call to itself.

The factorial may be implemented using an iterative solution as well, like so:

```
fun factorialIterative(n: Int) =
  (1..n).fold(BigInteger("1")) { product, e -> product * e.toBigInteger() }
```

The `fold()` function is much like the `reduce()` function we saw in Internal Iterators, except it takes an initial value in addition to the lambda argument. While you could use iteration, the recursive solution will help gain recognition and acceptance among programming geeks. So, let's explore that option further.

Drawback of recursion

For complex problems, the recursive solution, if possible, may be more elegant and expressive than iterative solutions. Sadly, recursion runs into an issue that iterative solutions don't suffer from: recursion grows the stack, and once it reaches dangerously large levels, the program may crash.

For example, here's the iterative solution for a large value:

```
println(factorialIterative(50000))
```

The code will have no trouble running. Now try the same with the recursive solution:

```
println(factorialRec(50000))
```

Though the recursive solution is elegant, it doesn't stand up to the challenge; the code fails at runtime when the input size is large.

```
java.lang.StackOverflowError
        at java.base/java.math.BigInteger.valueOf(BigInteger.java:1182)
        at Largerecursive.factorialRec(largerecursive.kts:4)
```

QUIZ

What is a drawback of using recursion?

Retake Quiz

Sigh, these geek cults aren't so easy to infiltrate. We need to up our skills a notch. Let's see how tail call optimization can help in the next lesson.