Applying Memoization to Dynamic Programming

We'll cover the following

- Dynamic programming
- A regular solution
- A memoized solution

Dynamic programming

Dynamic programming is an algorithmic technique that uses memoization to make the execution of recursive calls highly efficient. By caching and reusing the results of a function call, dynamic programming eliminates repetitive recursive calls to functions. So computations that may have exponential time complexity may execute with linear time complexity, thanks to memoization.

In the previous section, we used memoization to compute the Fibonacci number and saw how the technique greatly reduced the computation time. Let's apply the `Memoize` delegate we created to solve a well-known problem in dynamic programming—the rod-cutting problem.

Unlike the Fibonacci number, where there's one number for a given position, a category of problems called *optimization problems* may have multiple possible solutions. A user may pick one among the possible solutions, but we have to explore the different solutions to facilitate that. Dynamic programming is often used to recursively explore the possible solutions for optimization problems. Let's explore how our solution in the previous section applies to the rod-cutting problem, which is an optimization problem.

A regular solution

Given prices for different lengths of a rod, the objective of the problem is to find the maximum revenue a seller can make for a given rod length by cutting it. For example, given the prices in dollars, 2, 4, 6, 7, 10, 17, 17, for lengths of 1, 2, ..., 7 units (inches or centimeters depending on the units of measure you use), find the

maximum revenue for a length of 4 units. If the seller were to sell the rod of length

4 as is, the revenue will be $7. However, if cut into four, each of length 1 unit, the revenue will be $8. Likewise, if the seller cuts it into two equal parts, the revenue will be $8. Cutting into two pieces of length 1 and 3 units, incidentally, will yield the same revenue. Thus, there are three solutions, each yielding the same maximum revenue for length of 4 units. But if further cutting the sub-length of 3 units, for example, into smaller pieces may yield a better revenue, we should explore that as well. Thus, the problem nicely fits into a recursive solution, illustrated in the following pseudocode:

```
maxPrice(length) =
  max {
                maxPrice(1) + maxPrice(length - 1),
                maxPrice(2) + maxPrice(length - 2),
                ...,
                maxPrice(length - 1) + maxPrice(1), price[length]
}
```

This is a maximum of maximum computation; that is, it takes the maximum price for each combination of cuts that total the length and finds the maximum among them.

A memoized solution

This solution can benefit from memoization. The computation of `maxPrice(3)`, for example, involves computing `maxPrice(2)` and `maxPrice(1)`. And, in turn, computing `maxPrice(2)` again involves computing `maxPrice(1)`. Memoization will remove such redundancies in computation. Let's turn the above pseudocode into Kotlin code. The `Memoize` delegate used in the following code is the one we created in the previous section.

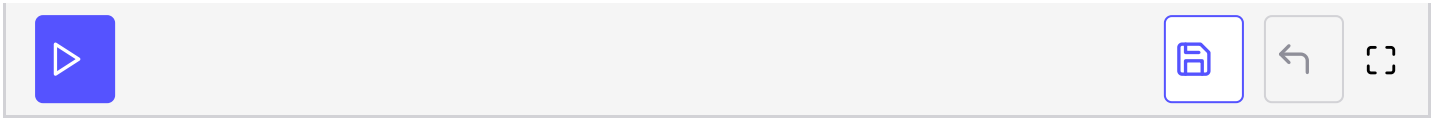
```
val prices = mapOf(1 to 2, 2 to 4, 3 to 6, 4 to 7, 5 to 10, 6 to 17, 7 to 17)

val maxPrice: (Int) -> Int by Memoize { length: Int ->
  val priceAtLength = prices.getOrDefault(length, 0)

  (1 until length).fold(priceAtLength) { max, cutLength ->
    val cutPrice =  maxPrice(cutLength) + maxPrice(length - cutLength)

    Math.max(cutPrice, max)
  }
}
```

```
for (i in 1..7) {
  println("For length $i max price is ${maxPrice(i)}")
}
```



cutrod.kts

The `prices` variable holds an immutable map of lengths and prices. The advantage of holding the prices in a `Map` instead of a `List` is that price isn't required for every single length. For a given length we can look up the price using the length as a key in a `Map` instead of an index in a `List`. That gives more flexibility. The `maxPrice` variable refers to a lambda that takes an `Int` for length and returns the maximum price for that length, as `Int`. The delegate that intercepts the call to the `maxPrice` variable computes the maximum of maximum using the `fold()` method of `IntRange`. For every value `cutLength` in the range from 1 to one less than the given length, we pick either the previously computed maximum value or the price we can get by cutting the rod into two pieces of `cutLength` and `length - cutLength`, whichever is higher. The iteration of `fold()` starts with the price for the given length, uncut, as the initial value.

We exercise the `maxPrice()` function for lengths from 1 to 7 to find the maximum price for each length. The output from the code is here:

```
For length 1 max price is 2
For length 2 max price is 4
For length 3 max price is 6
For length 4 max price is 8
For length 5 max price is 10
For length 6 max price is 17
For length 7 max price is 19
```

The output reflects the maximum price, which is either the same or higher than the price the seller will get if the rod of any length were sold uncut.

We've seen how the algorithm breaks the solution into recursive calls to solving the subproblems. However, the recursive calls involve repetitive invocation of functions for the same input. By using memoization, we avoid these repetitive recursive calls, thus greatly reducing the computation complexity of the code.

The next lesson concludes the discussion for this chapter.