Memoization

Memoization ensures that a function doesn't run for the same inputs more than once by keeping a record of the results for given inputs (usually in a dictionary).

For example, a simple recursive function for computing the nth fibonacci number:

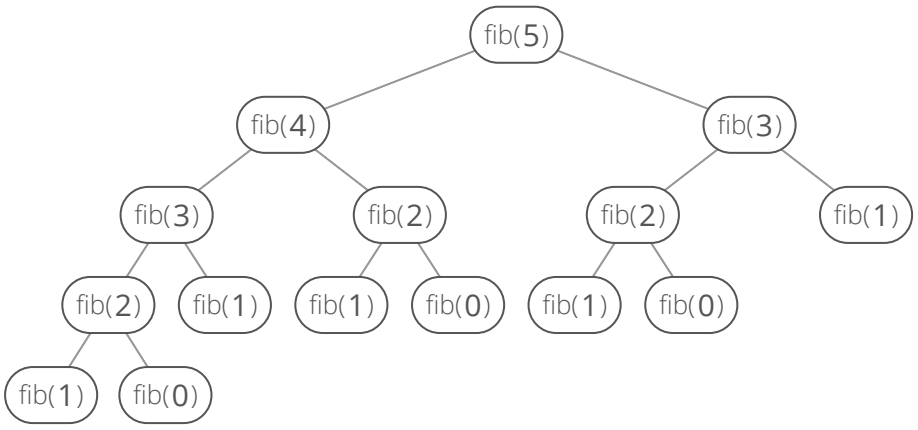
C# (beta)

```
public int FibRecursive(int n)
{
    // Edge case
    if (n < 0) {
        throw new ArgumentOutOfRangeException(nameof(n),
            "Index was negative. No such thing as a negative index in a series.");
    }

    // Base cases
    if (n == 0 || n == 1)
    {
        return n;
    }

    Console.WriteLine($"Computing FibRecursive({n})");
    return FibRecursive(n - 1) + FibRecursive(n - 2);
}
```

Will run on the same inputs multiple times:

```
// Output for FibRecursive(8)
Computing FibRecursive(8)
Computing FibRecursive(7)
Computing FibRecursive(6)
Computing FibRecursive(5)
Computing FibRecursive(4)
Computing FibRecursive(3)
Computing FibRecursive(2)
Computing FibRecursive(2)
Computing FibRecursive(3)
Computing FibRecursive(2)
Computing FibRecursive(4)
Computing FibRecursive(3)
Computing FibRecursive(2)
Computing FibRecursive(2)
Computing FibRecursive(5)
Computing FibRecursive(4)
Computing FibRecursive(3)
Computing FibRecursive(2)
Computing FibRecursive(2)
Computing FibRecursive(3)
Computing FibRecursive(2)
Computing FibRecursive(6)
Computing FibRecursive(5)
Computing FibRecursive(4)
Computing FibRecursive(3)
Computing FibRecursive(2)
Computing FibRecursive(2)
Computing FibRecursive(3)
Computing FibRecursive(2)
Computing FibRecursive(4)
Computing FibRecursive(3)
Computing FibRecursive(2)
Computing FibRecursive(2)
21
```

We can imagine the recursive calls of this function as a tree, where the two children of a node are the two recursive calls it makes. We can see that the tree quickly branches out of control:



To avoid the duplicate work caused by the branching, we can wrap the function in a class that stores an instance variable, `_memo`, that maps inputs to outputs. Then we simply:

1. Check `_memo` to see if we can avoid computing the answer for any given input, and
2. Save the results of any calculations to `_memo`.

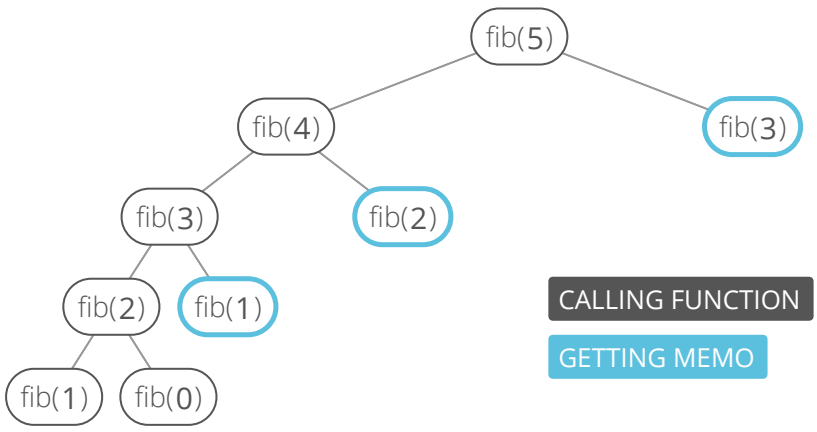
C# (beta)

```
public class Fibber
{
    private Dictionary<int, int> _memo = new Dictionary<int, int>();

    public int Fib(int n)
    {
        // Edge case
        if (n < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(n),
                "Index was negative. No such thing as a negative index in a series.");
        }

        // Base cases
        if (n == 0 || n == 1)
        {
            return n;
        }

        // See if we've already calculated this
        if (_memo.ContainsKey(n))
        {
            Console.WriteLine($"Grabbing _memo[{n}]");
            return _memo[n];
        }

        Console.WriteLine($"Computing Fib({n})");
        int result = Fib(n - 1) + Fib(n - 2);

        // Memoize
        _memo[n] = result;

        return result;
    }
}
```

We save a bunch of calls by checking the memo:

```
// Output of new Fibber().Fib(8)
Computing Fib(8)
Computing Fib(7)
Computing Fib(6)
Computing Fib(5)
Computing Fib(4)
Computing Fib(3)
Computing Fib(2)
Grabbing _memo[2]
Grabbing _memo[3]
Grabbing _memo[4]
Grabbing _memo[5]
Grabbing _memo[6]
21
```

Now in our recurrence tree, no node appears more than twice:



Memoization is a common strategy for **dynamic programming** problems, which are problems where the solution is composed of solutions to the same problem with smaller inputs (as with the fibonacci problem, above). The other common strategy for dynamic programming problems is **going bottom-up (/concept/bottom-up)**, which is usually cleaner and often more efficient.

See also:

- Overlapping Subproblems (/concept/overlapping-subproblems)
- Bottom-Up Algorithms (/concept/bottom-up)