Memoization

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

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

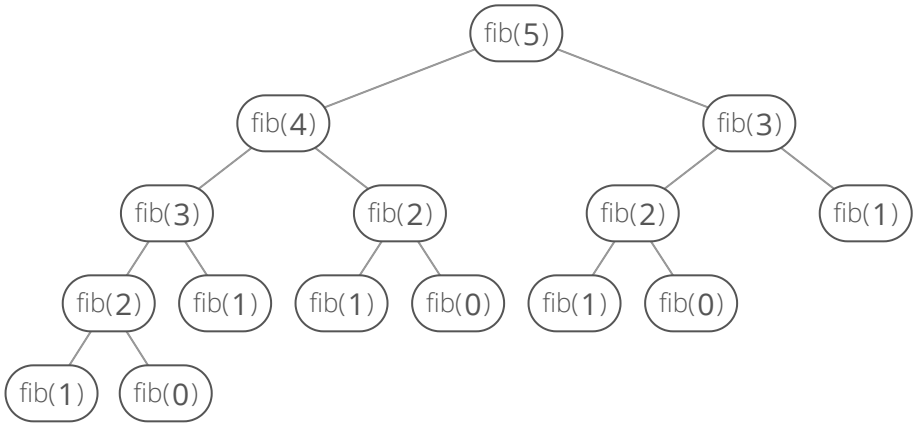
Objective-C

```
NSInteger ICKFib(NSInteger n) {
    NSCAssert(n >= 0, @"Input must be non-negative");

    if (n == 0 || n == 1) {
        return n;
    }
    NSLog(@"computing ICKFib(%ld)", (long)n);
    return ICKFib(n - 1) + ICKFib(n - 2);
}
```

Will run on the same inputs multiple times:

```
// output for ICKFib(5)
... computing ICKFib(5)
... computing ICKFib(4)
... computing ICKFib(3)
... computing ICKFib(2)
... computing ICKFib(2)
... computing ICKFib(3)
... computing ICKFib(2)
... 5
```

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 ivar, `_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`.

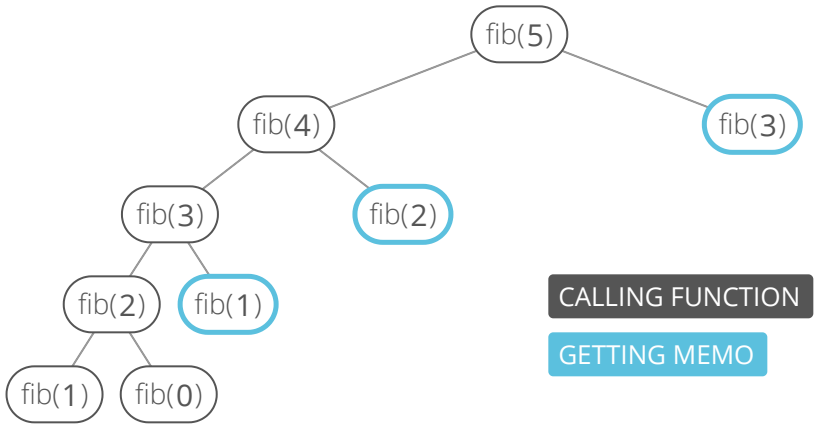
Objective-C

```
@interface ICKFibber : NSObject {
    NSMutableDictionary<NSNumber *, NSNumber *> *_memo;
}
- (NSInteger)fib:(NSInteger)n;
@end

@implementation ICKFibber

- (instancetype)init {
    if (self = [super init]) {
        _memo = [NSMutableDictionary new];
    }

    return self;
}

- (NSInteger)fib:(NSInteger)n {

    // edge case
    NSAssert(n >= 0, @"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[@(n)]) {
        NSLog(@"grabbing _memo[%ld]", (long)n);
        return _memo[@(n)].integerValue;
    }

    NSLog(@"computing fib:%ld", (long)n);
    NSInteger result = [self fib:(n - 1)] + [self fib:(n - 2)];

    // memoize
    _memo[@(n)] = @(result);

    return result;
}
@end
```

We save a bunch of calls by checking the memo:

```
// output of [[ICKFibber new] fib:5]
... computing fib:5
... computing fib:4
... computing fib:3
... computing fib:2
... grabbing _memo[2]
... grabbing _memo[3]
... 5
```

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.

← course home (/table-of-contents)

Next up: Bottom-Up Algorithms → (/concept/bottom-up?course=fc1&section=dynamic-programming-recursion)