Memoization

A theoretical question about the pros and cons of memoization.

Exercise:

What is memoization? What are its benefits? What is the necessary condition for using memoization? Illustrate the benefits of memoization using an example.

Remark:

Expect these types of questions when bridging theory with practice. You need to know what you are doing, and you have to write code that runs properly. The task itself is up to your interpretation, so choose the easiest possible solution.

Solution:

Memoization is an optimization technique often used with recursion. We create a lookup table that contains the mapping between the input and the output of the function. If the input has been calculated before, we find the corresponding output in the lookup table and return it. If the input has not been calculated, we calculate it and insert it into the lookup table.

There are several benefits of memoization. First, if the computation of the return value of a function takes a lot of time, substituting it with a simple lookup saves a lot of time. Second, we may save a lot of recursive calls, as computing the value may imply entering into the same sequence of recursive calls. In general, if we are interested in computing the same value over and over again, we have a case for memoization.

The necessary condition of using memoization is that the function has to be *deterministic*. This means that the input values should always determine the return value of the function regardless of the external context.

We will use the Fibonacci function to illustrate memoization. The original

We will use the Fibonacci function to illustrate memoization. The original Fibonacci function can be implemented like this:

```
let fibonacci = n =>
    n <= 1 ? n :
    fibonacci( n - 1 ) + fibonacci( n - 2 );

console.log(fibonacci(7))
```

Let’s add a global counter to count the number of function calls. We will just use a global variable now. Remember, our goal is to focus on the solution, there is no need to beautify the solution by adding complexity. Using proxies may be more elegant, but it’s a minefield in an interview situation, where one bad step may lead to your elimination.

```
let callCount = 0;
let fibonacci = n => {
    callCount += 1;
    return n <= 1 ? n :
           fibonacci( n - 1 ) + fibonacci( n - 2 );
}

callCount = 0;
fibonacci( 15 );
console.log( callCount );
//> 1973
```

Wow! We needed 1973 recursive calls to compute the Fibonacci value of 15. Let’s see how we can do better by using memoization.

```
let callCount, memoMap;
let fibonacci = n => {
    callCount += 1;
    if ( memoMap.has( n ) ) {
        return memoMap.get( n );
    }
    let result = n <= 1 ? n : fibonacci( n - 1 ) + fibonacci( n - 2 );
    memoMap.set( n, result );
    return result;
}

memoMap = new Map();
callCount = 0;
fibonacci( 15 );
console.log( callCount );
console.log( callCount );
//> 29
```

Cool! We just need 29 recursive calls instead of 1973. That’s an improvement.

But we can do better.

You remember the description of the use cases for memoization? The first sentence of the solution states: “Memoization is an optimization technique often used with recursion”. Then I continued describing a recursive solution.

We will now realize that memoization may be useful without recursion. Imagine there is a service in the form of a function, and we call this function quite often. Memoization acts as a cache to retrieve the values that have been calculated. Caching requires a more complex data structure, because the size of the cache is often limited, while in this implementation, the size of the memo map is unlimited.

We will demonstrate the use of caching by trying to compute `fibonacci(12)`, by using the values already in the memo map after the previous exercise:

```
memoMap = new Map();
callCount = 0;
fibonacci( 15 );
console.log( callCount );
//> 29

callCount = 0;
fibonacci( 12 );
console.log( callCount );
//> 1
```

Why is the call count one? Because `fibonacci(12)` had been calculated as a result of calculating `fibonacci(15)`. It was a mandatory step in the computation. All we need to do is to look up one value in the lookup table.

Similarly, when calculating `fibonacci(16)`, we can reuse the lookup table containing `fibonacci(15)` and `fibonacci(14)`, so we end up with just three calls:

```
callCount = 0;
fibonacci( 16 );
console.log( callCount );
//> 31
```

This is a thorough answer illustrating memoization.