**Dynamic Programming**

A way to solve problems by breaking it down into a collection of sub-problems, solving each sub-problems only once and storing their solutions in case the same sub-problem occurs next time.

An optimization technique using caching.

If you have something to cache, then you can use dynamic programming.

Dynamic Programming = Divide & Conquer + Memoization

**When to use Dynamic Programming?**

* Can be divided into subproblem
* Recursive Solution
* Are there repetitive subproblems?
* Memoize subproblems – if there are repetitive subproblems

**Caching**

A way of speeding up programs by holding data so that it can be easily accessed.

**Memoization**

A form of caching.

Reduces the time and space complexity.

Creating a cache where it can be accessed easily:

let cache = {};

function memoizedAddTo80(n) {

    if (n in cache) {

        return cache[n];

    } else {

        console.log('long time');

        cache[n] = n + 80;

        return cache[n];

    }

}

console.log('1', memoizedAddTo80(5));

console.log('2', memoizedAddTo80(5));

console.log('3', memoizedAddTo80(5));

Output:

long time // ‘long time’ will be printed the first time calling the method because there is

1 85 // nothing in the cache yet. After the first call to memoizedAddTo80 function it

2 85 // will print ‘long time’ then every call after that will just produce ‘85’ because

3 85 // the value has been added to the cache.

Dynamic Programming with Fibonacci Sequence

//0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233...

let calculations = 0;

function fibonacci(n) { //O(2^n)

    if (n < 2) {

        return n

    }

    return fibonacci(n - 1) + fibonacci(n - 2);

}

// Using dynamic programming by creating a cache, we are decreasing our time complexity and increasing our space complexity

function fibonacciMaster() { //O(n)

    // creating a hash table for the cache

    let cache = {};

    return function fib(n) {

        calculations++;

        if (n in cache) {

            return cache[n];

        } else {

            if (n < 2) {

                return n;

            } else {

                cache[n] = fib(n - 1) + fib(n - 2);

                return cache[n];

            }

        }

    }

}

We use dynamic programming to reduce the time complexity from O(2^n) to O(n)

However, we increased our space complexity

Bottom up – Dynamic Programming

// Bottom up approach to dynamic programming using Fibonacci Sequence

function fibonacciMaster2(n) {

    let answer = [0, 1];

    for (let i = 2; i <= n; i++) {

        answer.push(answer[i - 2] + answer[i - 1]);

    }

    return answer.pop(); // pop will return the last item in the array

}

console.log('DP2', fibonacciMaster2(100));

**Dynamic Programming Questions:**

<https://leetcode.com/problems/house-robber/>

<https://leetcode.com/problems/best-time-to-buy-and-sell-stock/>

<https://leetcode.com/problems/climbing-stairs/>