**Dynamic Programming**

A method/approach for solving a complex problem by breaking it down into a collection of simpler subproblems, solving each of those subproblems just once, and storing their solutions.

Note:

* Most problems cannot be solved with it.
* The ones that can be solved with dynamic programming, it can make a huge difference in their performance.
* D.P. refers to finding an optimal program in the same way, like the term ***linear programming*** (A method to achieve the best outcome in a mathematical model whose requirements are represented by linear relationships. Linear programming is a special case of mathematical programming.). It has nothing to do with code. It’s just one of the design of algorithm.

Dynamic Programming only works on problems with **Optimal Substructure** & **Overlapping Subproblems**

**Overlapping Subproblems:**

A problem is said to have overlapping subproblems if it can be broken down into subproblems which are reused several times.

Those pieces should not be unique. They’re used several times.

For Example: Fibonacci Sequence

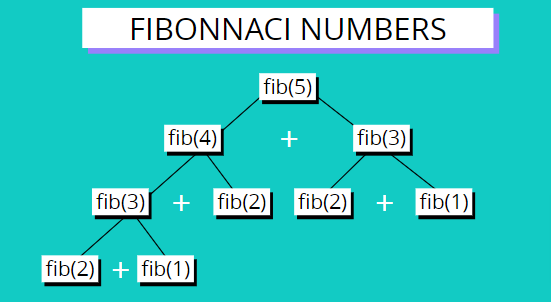
**Fibonacci Sequence:** It’s a sequence of numbers where every number is equal to the sum of the two previous numbers that came before it.

**Or**

Every number after the first two is the sum of the two preceding ones.

Ex:

1 1 2 3 5 8 13

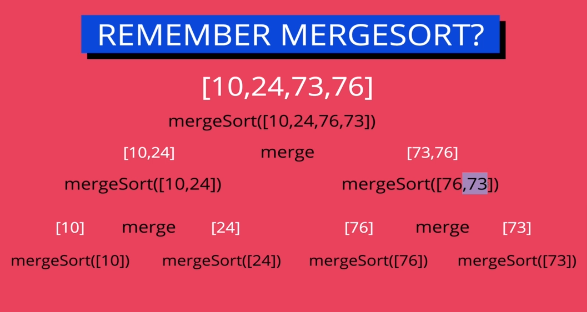


Here, the 3rd, 2nd & 1st Fibonacci numbers repeating several times.

*Overlapping sub problems*, means we need to look for repetition things where we’re repeating some sub problem.

So, *fibonnaic numbers* is an example of overlapping sub problems, where sub problems existed along with overlapping.

**Merge Sort:** Merge sort where we’ve an array and we split it up into smaller pieces in order to sort them and then we merge them back together.

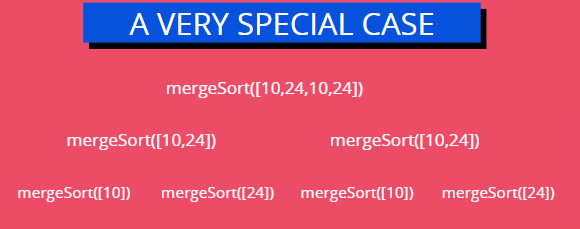


This involves sub problems (we’re taking a bigger problem and breaking it into smaller pieces, sub problems but there is no overlap (we’re not repeating the same thing, there’re no way we can reduce our duplication).

*Merge Sort*  is an example of no/non-overlapping sub problems, where sub problems existed but they doesn’t overlap.

And this structure often lends itself to the divide and conquer method or a pattern divide and conquer.

But this is not what we’re looking for. We want overlapping sub problems if we’re trying to use dynamic programming.

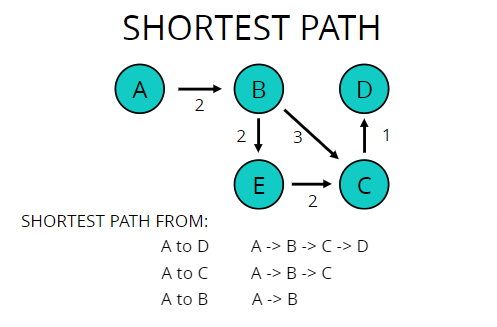
  
This a very special case of merge Sort where overlapping exist with sub problems.

So if for some reason we had data that looked like this with a ton of repetition, evenly spaced across an array, then we absolutely could use dynamic programming.

**Optimal Substructure:**

A problem is said to have optimal substructure if an optimal solution can be constructed from optimal solutions of its subproblems.

Example: Optimal *Substructure exists*



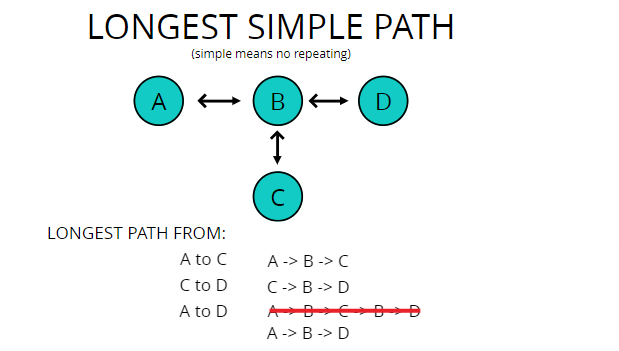
When we’re talking about the shortest path going from a start to an end, whatever that path is, any other poing along that path also is going to be the shortest path from the starts to that point.

So, from A, B, C to D, that includes the shortest path to C and the shortest path to B as well.

And if it didn’t it would no longer be the shortest path.

So that exhibits “*optimal substructure”*.

Example: Optimal *Substructure doesn’t exists*

*There is no Optimal Substructure*

Longest Simple Path where *simple* means is that there’s no repeating of a vertex, because if we did allow repetition it might create a loop by repeatedly moving over the same vertex.

For example:

If we find the longest simple path from A to D.

It would be A🡪 B 🡪 D instead A 🡪B 🡪 C 🡪 B 🡪 D because here repetition is occurring.

So if this followed optimal if it had optimal substructure, we could combine those (longest path from A to C & C to D) and we just say, the optimal longest solution to D from A is A, B, C, C, B, D.

But that’s not the case because we can’t repeat, so longest path doesn’t exhibit an optimal substructure.

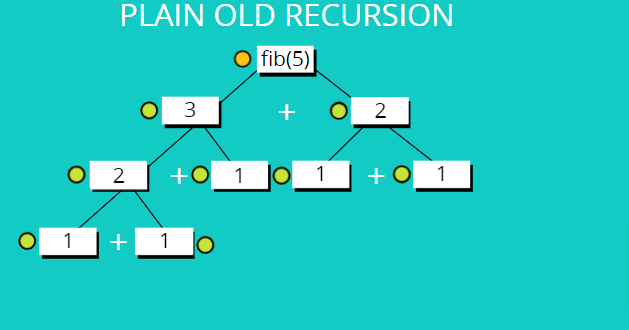
**FIBONACCI SEQUENCE:**

Plain Old Recursion

Fib(n) = Fib(n-1) + Fib(n-2)

Fib(2) is 1

Fib(1) is 1



**Code:**

function fib(n) {

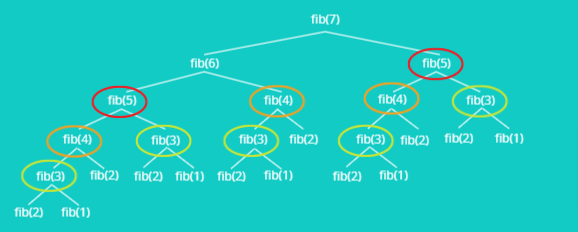
if(n<=2) return 1;

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

}

**BIG O Notation:**

Time Complexity: **O(2^n)** i.e. Exponential



So, the real problem is that we’re repeating things over and over.

This means that we can use Dynamic Programming. These are all sub problems and there’s overlap, there’s repetition.

**Solution:**

What if we could remember these old values?

That would be the whole point of dynamic programming,

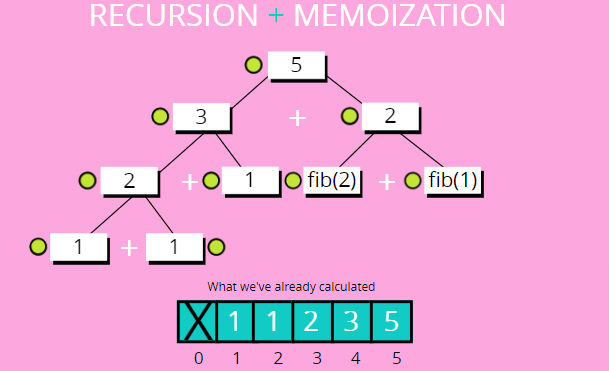
“Using past knowledge to make solving a future problem easier”

So, Memoization comes into play to solve this issue.

**Memoization (Top- Down)  
(One way of doing Dynamic Programming)**

Storing the results of expensive function calls and returning the cached result when the same inputs occur again.

The idea is that we have some structure to store data in an array or an object usually and then we do whatever expensive function call.



Ex:

Let’s say fib(5), and then we store it in that array so we can come back. And next time, if there’s another fib(5) call, instead of doing all of the work, we simply look in our table or array, whatever it is to see if there’s a fib(5) in there already. And if there we just use that piece. So we remember we’ve already solved it and we cashed it and we don’t have to do it again.

**Code:**

function fib(n, memo = []) {

if(memo[n]!==undefined) return memo[n];

if(n<=2) return 1;

let result = fib(n-1, memo) + fib(n-2, memo);

memo[n] = result;

return result;

}

fib(5);

Note:

Fibonacci numbers grow so quickly that numbers like integers in JavaScript can actually hold them and then we start to lose accuracy.

**BIG O Notation:**

Time Complexity: **O(n)** i.e. Linear

Note:

**Drawback of Memoization:**

At some point of Number, it will give an error like Maximum call stack size exceeded.

Like fib\_memo(10000);

**BOTTOM-UP**

We’ve been working TOP-Down before on Fibonacci Sequence.

So bottom-up means that we would start with fib(1) and fib(2) and add them together and then go to fib(3) and add that until we get to fib(7).

So the strategy that we’re going to use it tabulation.

**TABULATION (Bottom-Up Approach)  
(Another way of doing Dynamic Programming)**

Storing the result of a previous result in a "table" (usually an array) Usually done using **iteration**.

Better **space complexity** can be achieved using tabulation

So it tabulation it’s usually done using iteration, using a loop.

And we start at the bottom of whatever we’re trying to solve the smallest sub problem, and then we store the results in some sort of table, hence the term tabulation, usually an array.

So we store the data in an array and then we loop and move forward and we add things together or we do something with the data we’re not always adding, but in the case of Fibonacci, we will be.

**Code:**

function fib\_tab(n) {

if(n<=2) return 1;

let fibNums = [0, 1, 1];

for(let i=3; i<=n; i++){

fibNums[i] = fibNums[i-1] + fibNums[i-2];

}

return fibNums[n];

}

**BIG O Notation:**

Time Complexity: **O(n)** i.e. Linear

**Note:**

In TABULATION, fib\_tab(10000) give Infinity unlike any error in MEMOIZATION.