The method of dynamic programming, which is also sometimes referred to as "dynamic optimization" is an approach to solving complex problems by breaking them down into their smaller parts, and storing the results to these subproblems so that they only need to be computed once.

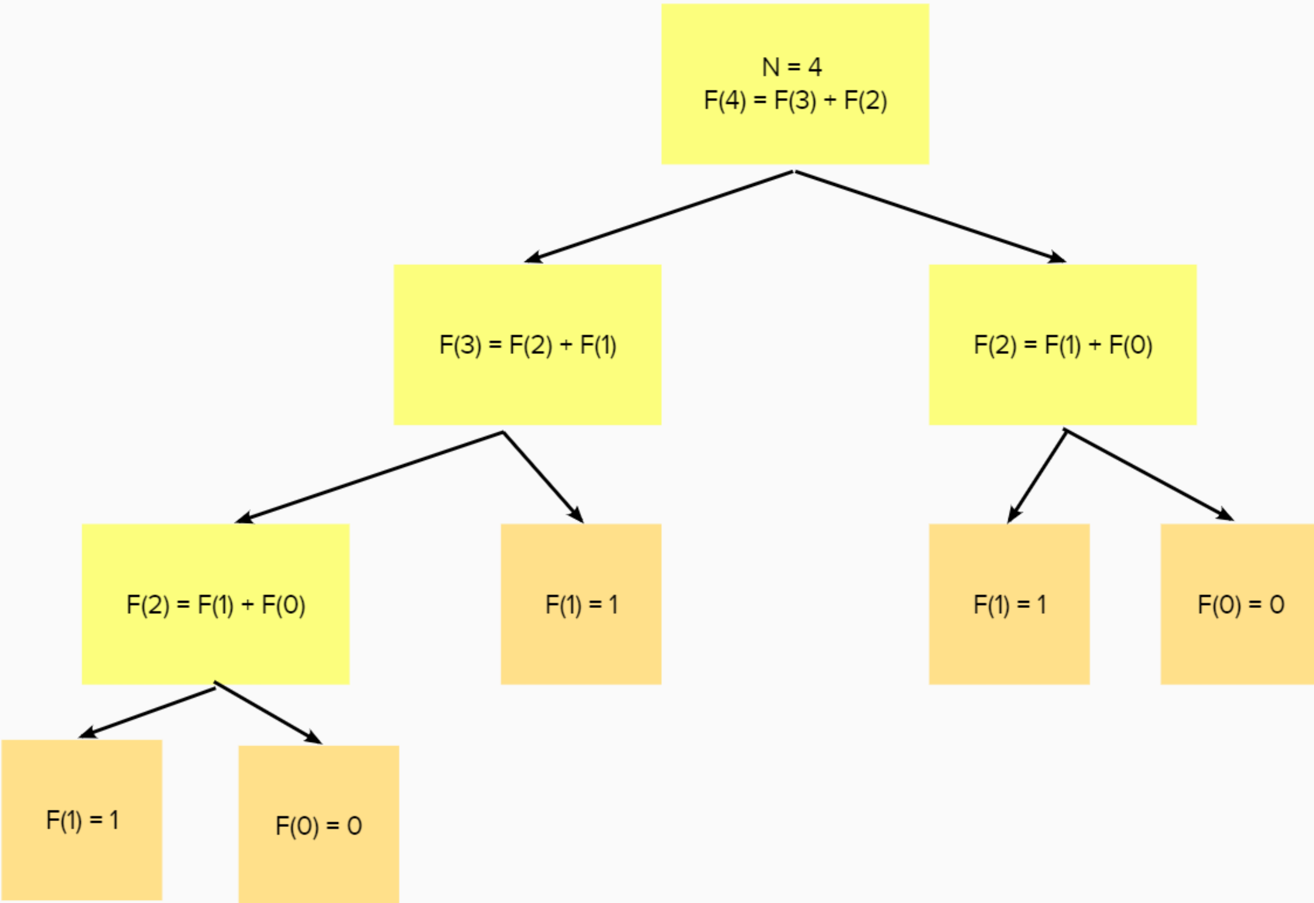
Divide & Conquer Algorithm	Greedy Algorithm	Dynamic Programming Algorithm
- Divides a problem into simpler versions of itself. - Applies solution for smaller subproblem to the larger problem. - Combines answers to subproblems (recursive). - For example: mergesort.	- Optimizes by making the choice that is the best at the moment. - Chooses the locally-optimal option, hoping it will lead to the globally-optimal solution. - For example: Dijkstra's algorithm	- Breaks a problem down into is sub-problems. - The subproblems are overlapping & recurring; DP will calculate them only once and save their values. - Sacrifices space to save time by remembering old subproblem values. - For example: memoized Fibonacci.

Dynamic programming is similar to divide + conquer in that it solves a problem by dividing it into sub-problems. However, in the dynamic programming paradigm, the larger problem is solved by solving and remembering overlapping sub-problems, which are reused repeatedly in the process.

Dynamic programming is similar to the greedy algorithm paradigm in that both approaches use an optimal substructure, where the optimal solution will hold the optimal solution for the subproblems within it. However, in dynamic programming, we find the optimal solution for every single sub-problem, and choose the best option. In the greedy algorithm, we only solve one sub-problem, based on an initial greedy choice.

* Dijkstra's algorithm is considered to be a greedy algorithm because it picks the vertex to which there is a shortest path <u>currently</u> <u>known</u>.

→ It doesn't exhaustively search through all of the "subproblems" of the graph. Instead, it iteratively chooses the best vertex to visit based on edge weight, making the greedy choice.


N = 4
F(4) = F(3) + F(2)
F(3) = F(2) + F(1)
F(2) = F(1) + F(0)
F(2) = F(1) + F(0)
F(1) = 1
F(1) = 1
F(0) = 0
F(1) = 1
F(0) = 0

how to think about dynamic programming:

→ How would we solve 5+5+5+5?

* we'd add them up! 5+5+5+5=20 ✓

→ What if we added another 5? Would we add them up again in the same way?

5+5+5+5+5?

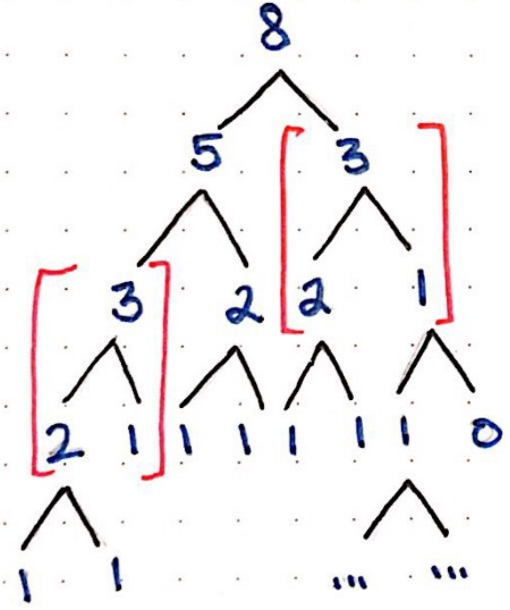
* no! we already <u>know</u> that 5+5+5+5 is 20, because we already solved it and remember our answer. So: 20+5=25 ✓

Dynamic programming allows us to avoid repeating ourselves and repeating bits of work by remembering partial portions of problems that we have already solved along the way.

Fibonacci sequence: 0, 1, 1, 2, 3, 5, 8, 13, 21 ...

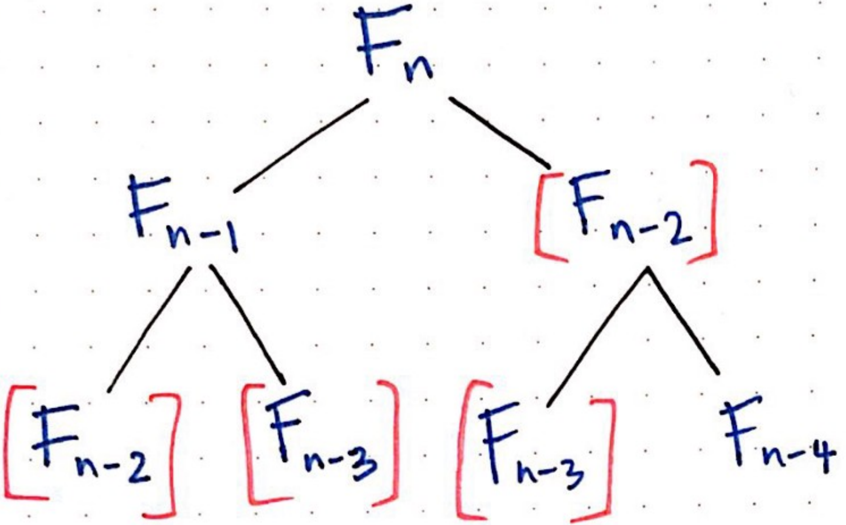
* derive any number by finding the two numbers that come before it, and summing them.

→ in other words: $F_n = F_{n-1} + F_{n-2}$



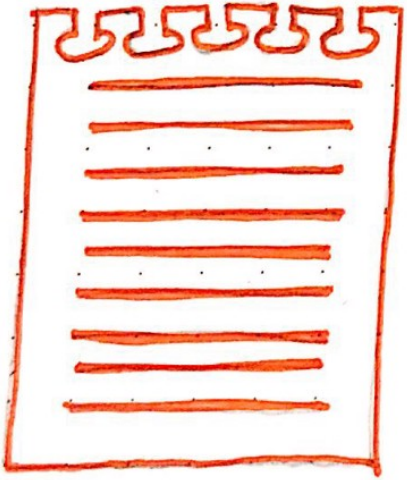
* Fibonacci can be solved iteratively, but it lends itself well to recursive implementation.

* Even in a recursive implementation we end up solving for the same values, recursively, more than once!



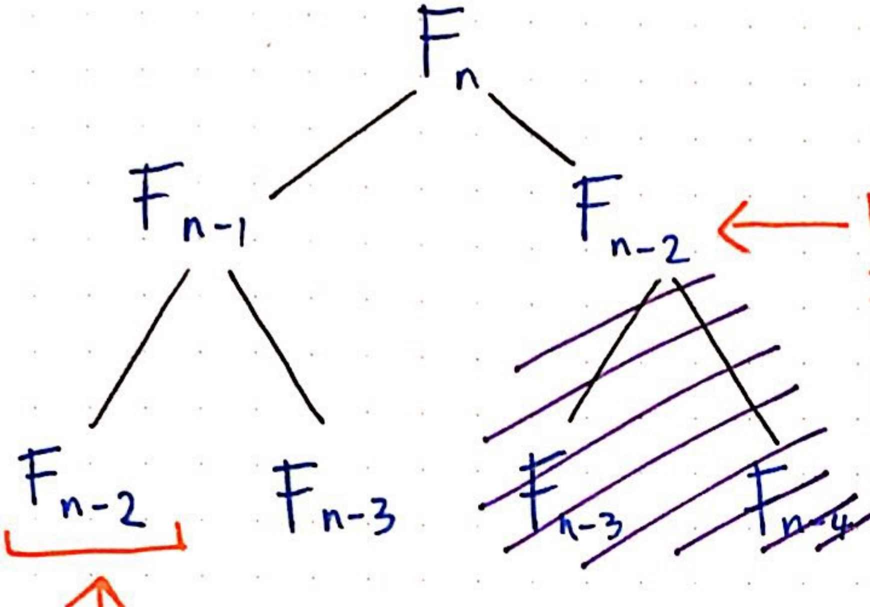
Why are we recalculating F_{n-2} and F_{n-3} so many times?! We can do better. We can use **dynamic programming!**

* We can use memoization to remember the problems that we have seen before and already solved so as not to resolve them for no good reason!



→ Memoization is like taking notes on a memo pad.

→ When we solve a problem by breaking it into subproblems, we check to see if that subproblem has already been solved before. If so, there is no need to recompute it!



When we go to calculate F_{n-2}, we check whether we have already solved for it.

As it turns out, we <u>have</u> solved for it! And we know its value since we memoized it!

* Since we needed to solve for F_{n-2} when we first solved for F_{n-1}, we don't need to recalculate this. Notice how memoization allowed us to solve for F_n, but also allowed us to eliminate half of the tree in the process.

→ Most recursive algorithms can be easily memoized, and thus, made so much more efficient!

top down	bottom up
- start with the large, complex problem, and build a solution for it by understanding how to build it/break it down into smaller sub-problems, smaller solutions - memoization as we break down the problem into parts: solve F_{n-1}, then F_{n-2}, then F_{n-3}...	- start with the smallest solutions, the smallest subproblems, and then build up each solution until we arrive at the solution to the larger subproblem. - Solve the Fibonacci sequence starting with 0 and 1 first, memoizing as we build our way up to whichever Fibonacci number we're trying to find.

* A benefit to the bottom up dynamic programming approach (DP) is that we can save space since we're working our way up. We only need to really memoize the last 2 values, which means we can achieve constant space O(1).