**Dynamic programming** is all about breaking down an optimization problem into simpler sub-problems, and storing the solution to each sub-problem so that each sub-problem is solved only once.

**Sub-problems**

We said that, it is all about breaking down the original problem into simpler sub-problems. But what is a sub-problem?

For example lets think about a simple mathematical calculation:

3 + 7 + 8 + 1 + 5 + 2 + 3 + 7 + 8 + 8 + 8 + 1 = 61

We can divide it into sub-problems;

3 + 7 = 10

8 + 1 = 9

5 + 2 = 7

3 + 7 ---> We already calcualted it! It's 10.

8 + 8 = 16

8 + 1 ---> We already calcualted it! It's 9.

In Dynamic Programming (DP), we are storing the solutions of sub-problems so that we do not need to recalculate it later. This is called Memoization.

By finding the solutions for every single sub-problem, we can solve the original problem itself.

**Memoization**

Memoization refers to the technique of caching and reusing previously computed results. It is a top-down approach where we just start by solving the problem in a natural manner and storing the solutions of the sub-problems along the way.

Suppose that we want to find the Fibonacci number at a particular index of the sequence. So fibonacci(n) = nth element in the Fibonacci sequence.

This problem is normally solved with **Divide and Conquer** algorithm. There are 3 main parts in this technique:

**Divide** - divide the problem into smaller sub-problems of the same type

**Conquer** - solve the sub-problems recursively

**Combine** - combine all the sub-problems to create a solution to the original problem

Lets define a function which will be responsible for calculating each of the Fibonacci numbers up to some defined limit n.

def fibonacci(n):

if n == 0 or n == 1:

return n

else:

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

This first naive solution recursively calculates each number in the sequence from scratch. This method has O(2n) time complexity, which is really bad runetime. For example, calculating fibonacci(40) will take more than a minute!

This also happens to be a good example of the danger of naive recursive functions.

The main idea behind Memoization was to re-use already calculated sub-problem results in order to solve the original problem.

**Recursion Tree**

As you can see in the recursion tree, the same sub-problems occured more than once. For example fib(3) is occuring twice, fib(2) is occuring 3 times etc.

So, despite calculating the result of the same problem, again and again, we can store the results once and use them again whenever needed.

Let’s write the same code but this time by storing the terms we have already calculated.

memo = {0: 0, 1: 1}

def fibonacci\_memoization(n):

if n in memo.keys():

return memo[n]

else:

memo[n] = fibonacci\_memoization(n - 1) + fibonacci\_memoization(n - 2)

return memo[n]

By not computing the full recusrive tree on each iteration, we’ve essentially reduced the running time for fibonacci(40) from more than a minute to almost instant. But we are sacrificing memory for the speed. Dynamic programming basically trades time with memory.

**Tabulation**

The other way we could have solved the Fibonacci problem was by starting from the bottom i.e., start by calculating the 2nd term and then 3rd and so on and finally calculating the higher terms on the top of these, by using these values. This bottom-up approach is called Tabulation.

def fibonacci\_tabulation(n):

if n == 0:

return n

# pre-initialize array

f = [0] \* (n + 1)

f[1] = 1

for i in range(2, n + 1):

f[i] = f[i - 1] + f[i - 2]

return f[n]

**Memoization vs TebulationPermalink**

Generally speaking, memoization is easier to code than tabulation. We can write a memoriser wrapper function that automatically does it for us. With tabulation, we have to come up with an ordering.

Also, memoization is indeed the natural way of solving a problem, so coding is easier in memoization when we deal with a complex problem. Coming up with a specific order while dealing with lot of conditions might be difficult in the tabulation.

What is more, think about a case when we don’t need to find the solutions of all the subproblems. In that case, we would prefer to use the memoization instead.

Tabulation is faster, as you already know the order and dimensions of the table. Memoization is slower, because you are creating the table on the fly. Generally, memoization is also slower than tabulation because of the large recursive calls.

In memoization, table does not have to be fully computed, it is just a cache while in tabulation, table is fully computed.

If all sub-problems must be solved at least once, a bottom-up tabulated dynamic programming algorithm usually outperforms a top-down memoized algorithm by a constant factor.

**Problem: Climbing Stairs**[**Permalink**](https://emre.me/coding-patterns/staircase/#problem-climbing-stairs)

LeetCode 70 - Climbing Stairs [easy]

You are climbing a stair case. It takes n steps to reach to the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Note: Given n will be a positive integer.

Example 1:

Input: 2 Output: 2

Explanation: There are two ways to climb to the top.

--> 1 step + 1 step

--> 2 steps

Example 2:

Input: 3 Output: 3

Explanation: There are three ways to climb to the top.

--> 1 step + 1 step + 1 step

--> 1 step + 2 steps

--> 2 steps + 1 step

**Brute Force Solution**

At every step, we have two options: either climb **1** step or **2** steps.

def climbStairs(self, n: int) -> int:

# we don't take any steps, so there is only 1 way

if n == 0:

return 0

# we can take one step to reach the end, and this is the only way

if n == 1:

return 1

# we can take one step twice or take two steps to reach the end

if n == 2:

return 2

# if we take one step, we are left with "n - 1" steps

take1step = self.climbStairs(n - 1)

# if we take two steps, we are left with "n - 2" steps

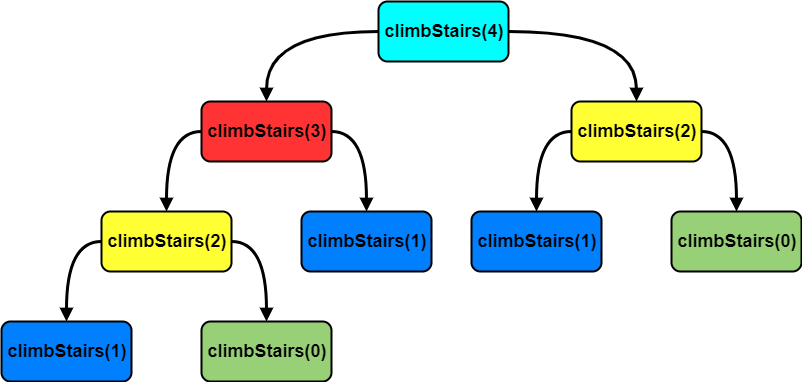
take2steps = self.climbStairs(n - 2)

return take1step + take2steps

**Time Complexity**: **O(2N)** because we are making **2** recursive calls in the same function.

**Space Complexity**: **O(N)** which is used to store the recursion stack.

But can we do better? To be able to understand this, lets visualize the recursion stack.



We can clearly see from colors that there are lots of *overlapping sub-problems* that we don’t need to calculate them again and again.

**Top-down Dynamic Programming with Memoization**[**Permalink**](https://emre.me/coding-patterns/staircase/#top-down-dynamic-programming-with-memoization)

We can use an array to store already solved sub-problems.

def climbStairs(self, n: int) -> int:

dp = [0 for x in range(n+1)]

return self.climbStairs\_recursive(dp, n)

def climbStairs\_recursive(self, dp, n):

# we don't take any steps, so there is only 1 way

if n == 0:

return 0

# we can take one step to reach the end, and this is the only way

if n == 1:

return 1

# we can take one step twice or take two steps to reach the end

if n == 2:

return 2

if dp[n] == 0:

# if we take one step, we are left with "n - 1" steps

take1step = self.climbStairs\_recursive(dp, n - 1)

# if we take two steps, we are left with "n - 2" steps

take2steps = self.climbStairs\_recursive(dp, n - 2)

dp[n] = take1step + take2steps

return dp[n]

**Time Complexity**: **O(N)** because [memoization](https://emre.me/algorithms/dynamic-programming/" \l "memoization) array dp[n+1] stores the results of **all** *sub-problems*. We can conclude that we will not have more than n + 1 *sub-problems*.

**Space Complexity**: **O(N)** which is used to store the recursion stack.

**Bottom-up Dynamic Programming with Tabulation**[**Permalink**](https://emre.me/coding-patterns/staircase/#bottom-up-dynamic-programming-with-tabulation)

Let’s try to populate our dp[] array in a bottom-up fashion. As we see from the recursion stack visualization, each climbStairs(n) call is the sum of the **two** previous calls.

We can use this fact while populating our dp[] array.

def climbStairs(self, n: int) -> int:

dp = [0 for x in range(n+1)]

dp[0] = 1

dp[1] = 1

for i in range(2, n+1):

dp[i] = dp[i-1] + dp[i-2]

return dp[n]

**Time Complexity**: **O(N)**

**Space Complexity**: **O(N)** which is used to store the recursion stack.

**Memory Optimization**

As we can see from the visualization of the recursive stack and other solutions, to be able to calculate the **n**, you need the value of last two combinations: **n-1** and **n-2**.

These two values are enough and we don’t need to store all other past combinations.

def climbStairs(self, n: int) -> int:

if n <= 0:

return 0

if n == 1:

return 1

if n == 2:

return 2

first = 1 # how many step possibilities there are with 1 stairs

second = 2 # how many step possibilities there are with 2 stairs

third = 0

for \_ in range(2, n):

third = first + second

first = second # walk up first to second

second = third # walk up second to third

return third

OR more shortly;

def climbStairs(self, n: int) -> int:

a = b = 1

for \_ in range(n):

a, b = b, a + b

return a

**Time Complexity**: **O(N)**

**Space Complexity**: **O(1)**

**How to identify?**

[Staircase](https://emre.me/coding-patterns/staircase) pattern is very useful to solve [Dynamic Programming](https://emre.me/algorithms/dynamic-programming/) problems involving *minimum/maximum steps*, *jumps*, *stairs*, *fibonacci numbers* etc. to reach a target.

**Similar LeetCode Probleems**

* [LeetCode 62 - Unique Paths [*medium*]](https://leetcode.com/problems/unique-paths/) [LeetCode 91 - Decode Ways [*medium*]](https://leetcode.com/problems/decode-ways/)
* [LeetCode 509 - Fibonacci Number [*easy*]](https://leetcode.com/problems/fibonacci-number/) [LeetCode 746 - Min Cost Climbing Stairs [*easy*]](https://leetcode.com/problems/min-cost-climbing-stairs/)
* [LeetCode 1155 - Number of Dice Rolls With Target Sum [*medium*]](https://leetcode.com/problems/number-of-dice-rolls-with-target-sum/)