**Dynamic Programming**

# Concept

## When to use DP

1. The problem can be broken down into "overlapping subproblems" - smaller versions of the original problem that are re-used multiple times.
2. The problem has an "optimal substructure" - an optimal solution can be formed from optimal solutions to the overlapping subproblems of the original problem.

It is commonly used for optimization problems (find min/max), counting problems (How many ways are there to…) and the possibility of doing something: is it possible to reach a certain point.

* What is the minimum cost of doing...
* What is the maximum profit from...
* How many ways are there to do...
* What is the longest possible...
* Is it possible to reach a certain point...

# Framework for DP Problems

1. A function or DS that will compute/contain the answer to the problem for every given state
2. A recurrent relation for transition between states.
3. Base case(s) so that our recurrence relation does not go on infinitely.

# Top Down (Memoization)

DS to visualize it is a tree. You start from the root and the number of branches from every node is the number of recursive calls in each state.

You start from the root and keep transitioning using recurrence relation till you reach base case.

Every state is a subproblem to the original one, the first call to the function is the main problem, then with each transition you take a smaller subproblem till you reach base case.

Each state depends on the result of the next state, then takes decision (optimization/count) then returns this decision to the previous state and so on.

# Bottom Up (Tabularization)

DS to visualize it is a table, you keep filling this table starting with the base case and then looping over all the state variables you fill the current state using the previously filled state.

The question that needs to be answered is which state I am coming from.

The current subproblem is the filled table so far.

You return the last state that you reach, which is the final value for all state variables (the original problem).

# Top-down to Bottom-up

1. Start with a completed top-down implementation.
2. Initialize an array dpdp that is sized according to your state variables. For example, let's say the input to the problem was an array numsnums and an integer kk that represents the maximum number of actions allowed. Your array dpdp would be 2D with one dimension of length nums.lengthnums.length and the other of length kk. The values should be initialized as some default value opposite of what the problem is asking for. For example, if the problem is asking for the maximum of something, set the values to negative infinity. If it is asking for the minimum of something, set the values to infinity.
3. Set your base cases, same as the ones you are using in your top-down function. Recall in House Robber, dp(0) = nums[0]dp(0) = nums[0] and dp(1) = max(nums[0], nums[1])dp(1) = max(nums[0], nums[1]). In bottom-up, dp[0] = nums[0]dp[0] = nums[0] and dp[1] = max(nums[0], nums[1])dp[1] = max(nums[0], nums[1]).
4. Write a for-loop(s) that iterate over your state variables. If you have multiple state variables, you will need nested for-loops. These loops should **start iterating from the base cases**.
5. Now, each iteration of the inner-most loop represents a given state, and is equivalent to a function call to the same state in top-down. Copy the logic from your function into the for-loop and change the function calls to accessing your array. All dp(...)dp(...) changes into dp[...]dp[...].
6. We're done! dpdp is now an array populated with the answer to the original problem for all possible states. Return the answer to the original problem, by changing return dp(...)return dp(...) to return dp[...]return dp[...].