\[ F_n = \begin{cases} 0 & \text{if } n = 0 \\ 1 & \text{if } n = 1 \\ F_{n-1} + F_{n-2} & \text{otherwise} \end{cases} \]

\[
\text{RecFIBO}(n):
\begin{align*}
&\text{if } n = 0 \\
&\quad \text{return } 0 \\
&\text{else if } n = 1 \\
&\quad \text{return } 1 \\
&\text{else} \\
&\quad \text{return } \text{RecFIBO}(n-1) + \text{RecFIBO}(n-2)
\end{align*}
\]

\[ T(n) : \text{running time} \]
\[ T(n) = \begin{cases} T(n-1) + T(n-2) + 1 & (n \geq 2) \\ 1 & (n \leq 1) \end{cases} \]

solves to
\[ T(n) = 2F_{n+1} - 1 \]
memoization: remember past solutions so we don't recompute them

First try: Use a global array.

```
MEMFIBO(n):
    if n = 0
        return 0
    else if n = 1
        return 1
    else
        if F[n] is undefined
            F[n] ← MEMFIBO(n - 1) + MEMFIBO(n - 2)
        return F[n]
```
If we compute them in order anyway, just make it explicit. \( O(n) \) time
Dynamic Programming - Bellman in the 50's
Rod cutting

Given: n; the length of the rod
P[1..n]: P[i] price for a rod piece of length i

Goal: what is maximum total selling price for all ways of cutting up the rod?
Max Revenue(\( \omega \)) : max revenue we can get after selling pieces of a length \( \omega \) rod.

We want to ultimately compute Max Revenue(\( n \)).

\[
\text{Max Revenue}(\omega) = \begin{cases} 
0 & \text{if } \omega = 0 \\
\max_{1 \leq j \leq \omega} (P[j] + \text{Max Revenue}(\omega - j)) & \text{otherwise}
\end{cases}
\]
Need to compute Max Revenue id for all 0 ≤ i ≤ n.

Store in an array Max Revenue[0..n].

Depend on smaller arguments so loop i ← 0 to n.

\[\text{FastRodCutting}(n, P[1..n]):\]
\[
\text{Max Revenue}[0] \leftarrow 0
\]
for \(i \leftarrow 1\) to \(n\)
\[
\text{Max Revenue}[i] \leftarrow -\infty
\]
for \(j \leftarrow 1\) to \(i\)
\[
\text{if } P[j] + \text{Max Revenue}[i - j] > \text{Max Revenue}[i]
\]
\[
\text{Max Revenue}[i] \leftarrow P[j] + \text{Max Revenue}[i - j]
\]
return Max Revenue[n]

\(O(n^2)\)
Dynamic Programming
(recursion without repetition)

1) Formulate/solve the problem recursively.
   a) Specify: Give a precise definition of the subproblems you're solving.

6) Then give recursive solution.

HARD PART
2) The easy part: speeding it up

a) Identify the subproblems: what values can the arguments take? (0 ≤ i ≤ n)

b) Choose a memoization data structure. (usually an array indexed by the recurrence parameters)

c) Identify dependencies: what needs to be done before you can solve any given subproblem?
d) Find a good evaluation order.

e) Find space & running time:
   Space: size of memo structure
   Time: usually \( \mathcal{O}(n^2) \) subproblems.
      \( \mathcal{O}(\text{time per subproblem}) \)

f) Write the code:
   (nested for loops)
   ("copy-paste" the recurrence)
DON'T BE GREEDY
(use dynamic programming)
Subset Sum:
Given: Set $X$ of positive integers $\times$ integer $T$.
Question: Is there a subset of $X$ summing to $T$?

$SS(i, t)$: True if some subset of $X(1..i)$ sum to $t$.

Want to ultimately compute $SS(n, T)$. 
Assume \( \dot{\omega} \geq 0 \) and \( t \geq 0 \).

\[
SS(\dot{\omega}, t) = \begin{cases} 
\text{True} & \text{if } t = 0 \\
\text{False} & \text{if } \dot{\omega} = 0 \text{ and } t > 0 \\
SS(\dot{\omega} - 1, t) & \text{if } \dot{\omega} > 0 \\
SS(\dot{\omega} - 1, t - x[\omega]) & 0 < t < x[\omega] \\
\text{otherwise} & \end{cases}
\]

Use \( x[i] \) → \( x(\dot{\omega}) \)

Don't use \( x(\dot{\omega}) \) → \( x(\dot{\omega}) \)

\( 0 \leq i \leq n \)

\( 0 \leq t \leq T \)

Store in \( SS[0..n, 0..T] \).

Go in increasing order of \( i \) parameter.
Space: $O(nT)$

Time: $O(1) \cdot O(nT) = O(nT)$