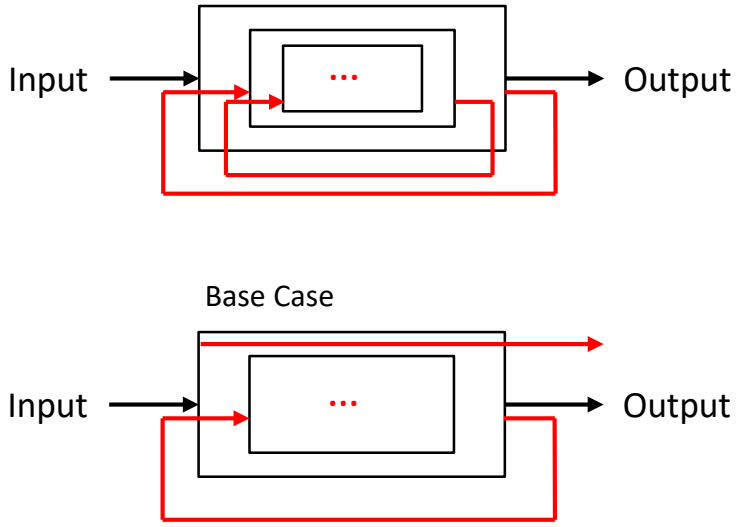
Dynamic Programming

Recursive Algorithm

- One of the central ideas of computer science
- Depends on solutions to smaller instances of the same problem (= sub-problem)
- Function to call itself (it is impossible in the real world)
- Factorial example
  - $n! = n \cdot (n-1) \cdots 2 \cdot 1$



Dynamic Programming

- Dynamic Programming: general, powerful algorithm design technique

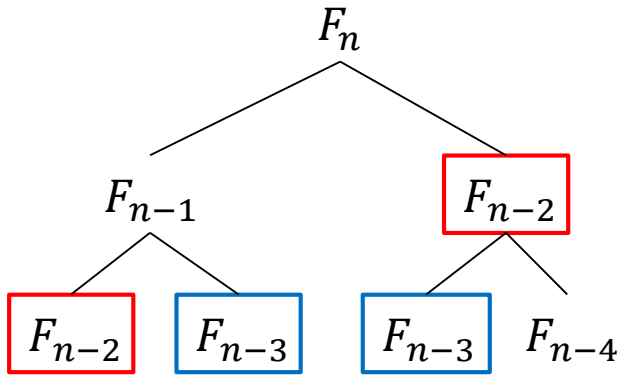
- Fibonacci numbers:

$$F_1 = F_2 = 1$$
$$F_n = F_{n-1} + F_{n-2}$$

Naïve Recursive Algorithm

```
fib(n) :
        if n ≤ 2 :  f = 1
        else :  f = fib(n − 1) + fib(n − 2)
        return f
```

- It works. Is it good?



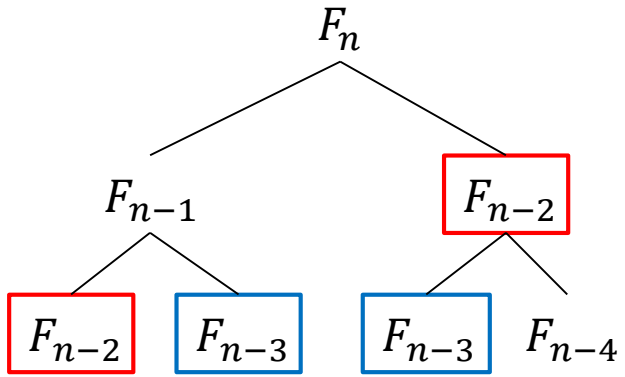
Memorized Recursive Algorithm

```
memo = [ ]
    fib(n) :
        if n in memo : return memo[n]

        if n ≤ 2 : f = 1
        else : f = fib(n − 1) + fib(n − 2)

        memo[n] = f
        return f
```

- Benefit?
  - fib(n) only recurses the first time it's called



Dynamic Programming Algorithm

- Memorize (remember) & re-use solutions to subproblems that helps solve the problem
- DP ≈ recursion + memorization

Example 1: Climbing a Stair

- You are climbing a stair case. Each time you can either make 1 step, 2 steps, or 3 steps. How many distinct ways can you climb if the stairs has $n = 30$ steps?

Example 2: Knapsack Problem

- Burglar (or thief) can carry at most 20 kg (= maximum capacity = 20)
- Quickly decide which item to carry

items	1	2	3	4	5	6
weight	10	9	4	2	1	20
value	175	90	20	50	10	200

- Approaches
  - Guess
  - Exhaustive search if possible
  - “smarter way” → recursive or dynamic programming

$$\text{key ideas} = \text{original problem} \rightarrow \begin{cases} \text{subproblem} \rightarrow \begin{cases} \text{subproblem} \rightarrow \\ \text{subproblem} \rightarrow \end{cases} \\ \text{subproblem} \rightarrow \begin{cases} \text{subproblem} \rightarrow \\ \text{subproblem} \rightarrow \end{cases} \end{cases}$$

Example 2: Knapsack Problem

- "smarter way" → recursive or dynamic programming

items	1	2	3	4	5	6
weight	10	9	4	2	1	20
value	175	90	20	50	10	200

Suppose we have the following function:

```
[value, taken] = chooseBest(items(1:6),maxWeight)
```

1) item 1 is not taken

```
[v_1,t_1] = chooseBest(items(2:6),maxWeight)
```

2) item 1 is taken

```
[v_2,t_2] = chooseBest(items(2:6),maxWeight - weights(1))
```

```
v_2 = v_2 + values(1)
```

```
t_2 = [items(1),t_2]
```