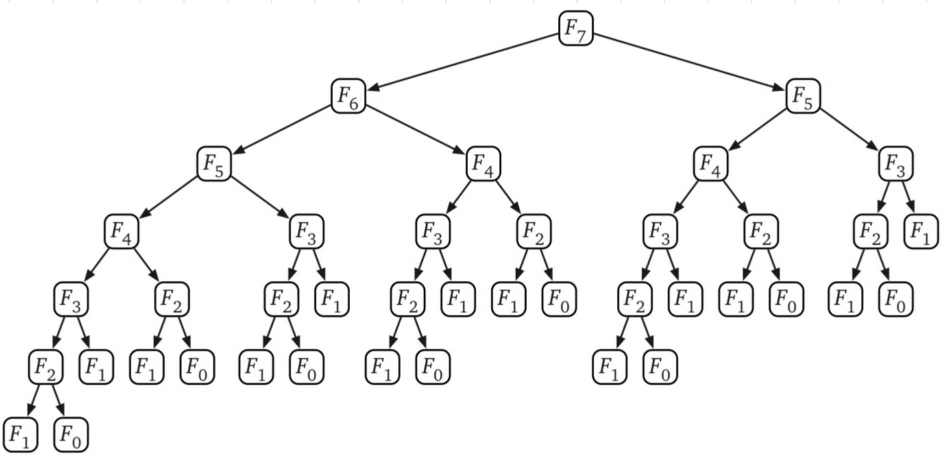
$$F_0 = 0 \qquad F_1 = 1 \qquad F_n = F_{n-1} + F_{n-2}$$

```
RecFibo(n):
  if n = 0
      return 0
  else if n = 1
      return n
  else
      return RecFibo(n − 1) + RecFibo(n − 2)
```

$$T(n) = T(n-1) + T(n-2) + 1$$
$$= 2F_{n+1} - 1$$
$$= \Theta(\phi^n) \qquad \phi = \frac{\sqrt{5}+1}{2} \approx 1.62$$



Memoization: remember results of subproblems in case we use them again

$F[0 \ldots]$: global memoization 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]
```

$O(n)$ space

time?

$F[i-2]$

Observation: when we compute $F[i]$, we already computed $F[i-1]$

⇒ we're computing them in increasing order!

```
ITERFIBO(n):
    F[0] ← 0
    F[1] ← 1
    for i ← 2 to n
        F[i] ← F[i−1] + F[i−2]
    return F[n]
```

$O(n)$ time!

dynamic programming

```
ITERFIBO2(n):
    prev ← 1
    curr ← 0
    for i ← 1 to n
        next ← curr + prev
        prev ← curr
        curr ← next
    return curr
```

optimization problem:

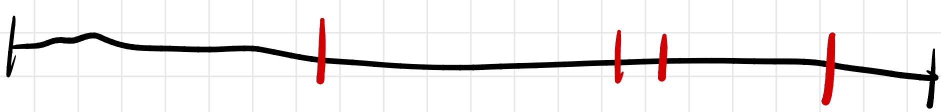
many different valid/feasible solutions, but...

each solution has a value

return one of maximum/minimum value

Rod Cutting:

Given a non-negative integer n & an array of integers $P[1..n]$.



n units of steel rod

Want to cut rod into integer length pieces for resale. Sale a pieces of length i to get $P[i]$ USD.

Goal: Want a list of positive integers $i_1, i_2, \ldots i_k$

s.t. $\sum_{j=1}^{k} i_j = n$

maximizing $\sum_{j=1}^{k} P[i_j]$.

Today: just focus on max value (revenue)

We're making a sequence of decisions.
Let's focus on the first one & let recursion tell us the consequences, so we know what first choice is best.

What is length of first piece?

Suppose we choose length j ...

left with rod of length $n-j$...

$CutRod(i) :=$ max revenue from cutting a rod of length $i \leq n$.

if j is correct first choice

$$CutRod(i) = P[j] + CutRod(i-j)$$

But which j?

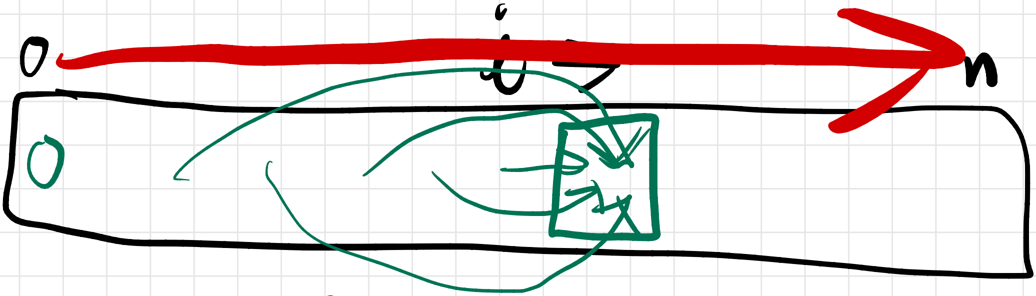
$$\text{CutRod}(i) = \begin{cases} \max_{1 \le j \le i} \{ P[j] + \text{CutRod}(i-j) \} & \text{if } i > 0 \\ 0 & \text{otherwise} \end{cases}$$

CutRod(n) is the optimal value we want!

Backtracking: try each option to make exactly one decision, using recursion to learn about quality / consequences of the decision

Optimal substructure: optimal solution to instance incorperates optimal solutions to subproblems

Use an array CutRod[0..n].



CutRod

So fill array in ___ order.
increasing

RODCUTTING$(n, P[1..n])$:
  $CutRod[0] \leftarrow 0$
  for $i \leftarrow 1$ to n
    $R[i] \leftarrow 0$
    for $j \leftarrow 1$ to i
      if $P[j] + R[i-j] > R[i]$
        $R[i] \leftarrow P[j] + R[i-j]$
  return $R[n]$

$O(n^2)$

Dynamic programming is *not* about filling in tables.
It's about smart recursion!

See Erickson 3.4.

1) Formulate problem recursively.

a) Say <u>in English</u> what the recursive subproblems solve.

+ say what parameters give solution to original problem

b) Give recursive solution (with base cases)

2) Build solutions from bottom up.

a) identify subproblems
what are possible parameter values? ($0 \le i \le n$ for CutRod)

b) choose a data structure

c) Identify dependencies.

d) Find an evaluation order.

e) Analyze space & running time.

space: size of structure

time: (usually) # subproblems × time per subproblem

f) write down the algorithm (for loops?)