1. Dynamic Programming (DP) (Bellman, 1950s)

- General, powerful alg. design technique
- Especially good for <u>optimization problems</u>
- DP ≈ "careful brute force"
- DP ≈ subproblems + reuse

"The Rabbit Problem"

- Start with a pair of baby rabbits
- Each pair matures in one month, and gives birth to another pair every month starting with the second month
- How many pairs of rabbits we have at k months?

1, 1, 2, 3, 5, 8, ...

$F_1 = F_2 = 1$ <u>Fibonacci Numbers</u>

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

"pair of rabbits currently"   "how many pairs ready to give birth"

Goal: Compute F_n

Naive Recursion:

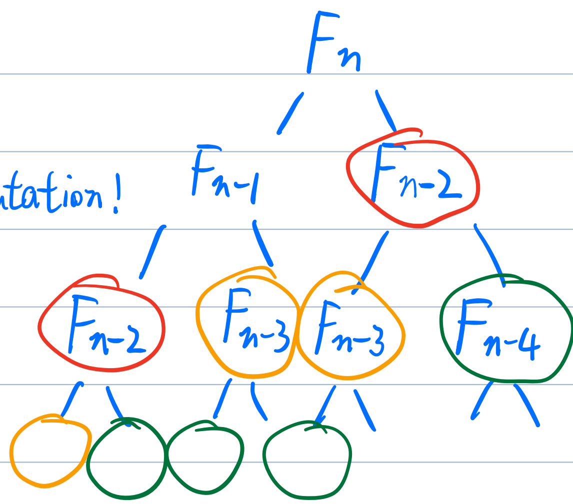
```
fib(n):
  If n≤2:
    Return 1
  Return fib(n-1)+fib(n-2)
```

Correctness: ✓

Runtime: $T(n) = T(n-1) + T(n-2) + \Theta(1)$

$$\geq 2T(n-2) + \Theta(1) \Rightarrow T(n) \geq 2^{n/2}$$

We're redoing a lot of computation!
Maybe we can memoize these results.



Memoized DP 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
```

* You can do this to any recursive function (and Python has it built in as of version 3.2)

Correctness: ✓

Runtime:

- fib(k) only recurses the first time it is called.
- A memoized call takes $O(1)$ time.
- # of non-memoized calls is n:

$$f(1), f(2), \dots, f(n)$$

$$\Rightarrow O(n) \text{ runtime}$$

DP: memoize (remember)

& reuse solutions to subproblems

DP ≈ recursion + memoization

$\Rightarrow$ time = # subproblems · (time per subproblem)

Not counting recursions!
So no recurrences!

v.s. Divide and Conquer:

Also have subproblems, but there they're usually <u>disjoint</u>, and we never encounter the same subproblem more than once, so memoization doesn't really help.

Bottom-up DP algorithm:

```
fib = { }
For k = 1 to n:
    If k ≤ 2:
        f = 1
    Else:
        f = fib[k-1] + fib[k-2]
    fib[k] = f
Return fib[n]
```

Correctness: ✓

Runtime: $\Theta(n)$

Space: n (can be improved to constant)

Any DP alg. can be converted into a bottom-up alg.

5 Steps for DP:

① Define Subproblems (#)

② Guess part of a solution (#)

③ Recurrence (time/subproblem)

④ Recurse + memoize or bottom-up

time = # subproblems · time/subproblem

⑤ Solve original problem

2. Rod Cutting

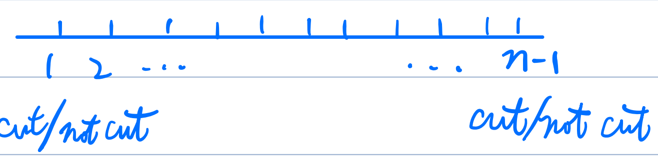
Given a n-feet rod, we want to cut it and sell it according to the following prices

length i	1	2	3	4	5	6	7	8	9
price p_i	1	5	6	9	10	17	17	20	24

Greedy?
Pick i with largest $\frac{p_i}{i}$. X e.g. 9

Brute Force: Check all possible cuttings!

$\# = 2^{n-1}$



We want to break this into smaller subproblems

$$r_n = \max(p_n,\ r_1 + r_{n-1},\ r_2 + r_{n-2},\ \dots,\ r_{n-1} + r_1)$$

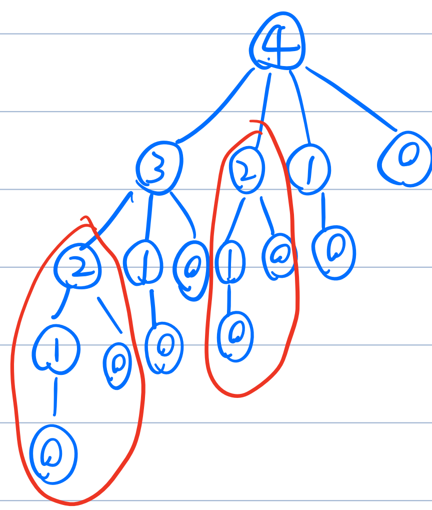
$\uparrow$ don't cut at all — $\uparrow$ make cutting after 1 — $\uparrow$ cutting after 2

Or, we can decompose with a first piece of length i, and then a remainder of length $n-i$.

$$r_n = \max(p_1 + r_{n-1},\ p_2 + r_{n-2},\ \dots,\ p_{n-1} + r_1,\ p_n)$$

```
CutRod(n):
    If n==0:
        Return 0
    q = -∞
    For i=1 to n:
        q = max(q, p_i + CutRod(n-i))
    Return q
```

(*)

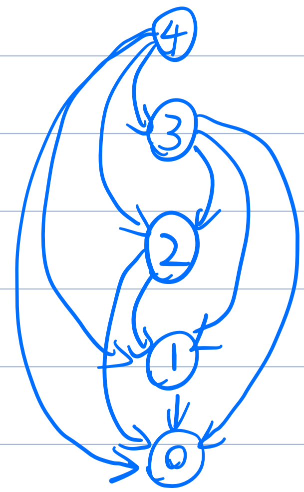


Runtime: $T(n) = 1 + \sum_{i=0}^{n-1} T(i) \Rightarrow T(n) = 2^n$ (prove by Induction)

Not so surprising since we are kinda brute-forcing here.

Speed up? Memoization!

```
CutRodMemo(n):
    If memo[n] exists:
        Return memo[n]
    (*)
    memo[n] = q
    Return q
```



Runtime: We need to compute CutRodMemo(1), ..., CutRodMemo(n), each takes $O(n)$, but recursive calls are free! $\Rightarrow O(n^2)$

$$\left.\begin{array}{l}\#\text{ of subproblems} = n \\ \text{time/subproblem} = O(n)\end{array}\right\} \Rightarrow O(n^2)$$

Bottom-up version:

```
CutRod(n)
    r[0] = 0
    For i = 1 to n
        q = -∞
        For j = 1 to i
            q = max(q, p_j + r[i-j])
        r[i] = q
    Return r[n]
```

Again, running time is $O(n^2)$.

Note: This outputs the revenue only. What if we want to output how to cut?
Memorize "the cut" for each r[i].