

# Fibonacci Numbers

$$F_0 = 0, F_1 = 1$$

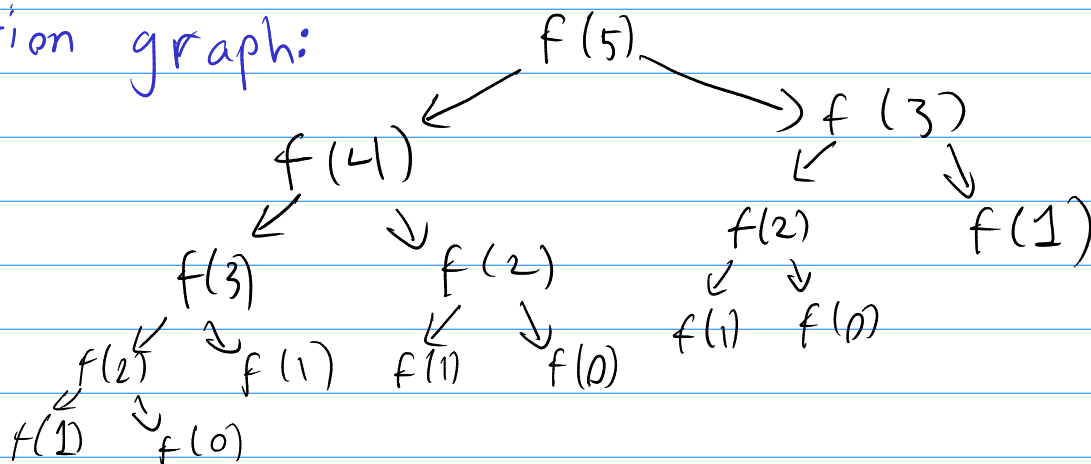
$$F_n = F_{n-1} + F_{n-2}$$

How quickly can we compute the  $n^{\text{th}}$  Fibonacci number?

Naive recursion:

```
def f(n):
    if n <= 1: return n
    return f(n-1) + f(n-2)
```

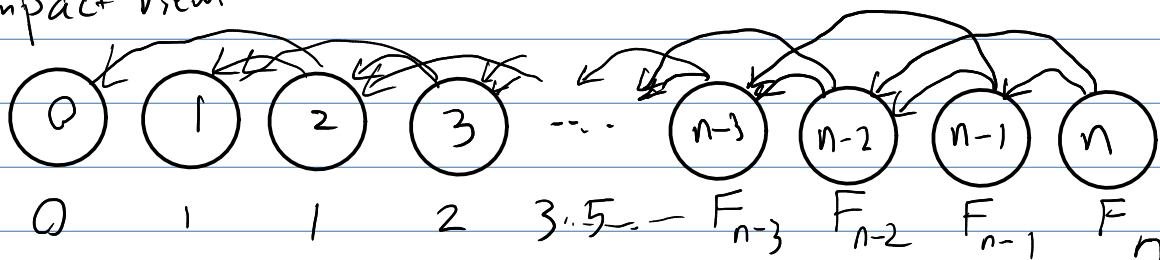
Computation graph:



More compact view:

"state"  $n$

$f(n)$



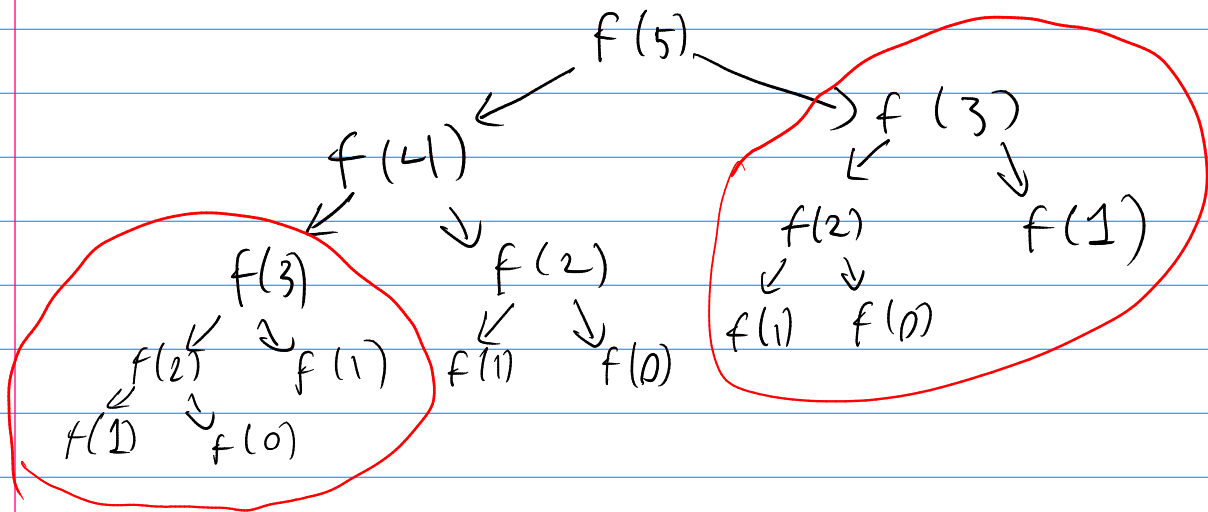
Returns to  $f(1)$  again and again

$$T(n) = T(n-1) + T(n-2) + \text{Something} \geq 1$$

$$\Rightarrow T(n) \geq F_n > 2^{n/2} \quad (\approx 1.6^n)$$

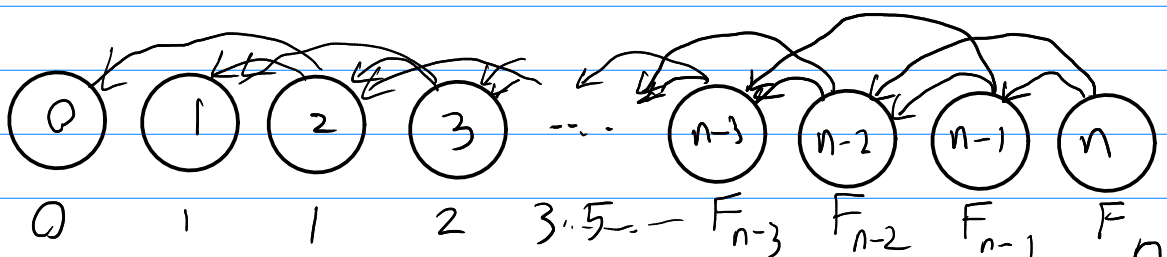
How can we avoid this repetition?

Only compute each  $F_i$  once.



$f(3)$  never changes. Can store the answer,

in this view: compute node (= "state" = "input") once.



# Dynamic Programming (DP)

"top-down"

memoization: Keep a record of each answer you compute.

Before recursion, check the memo.

"bottom-up" DP: compute answers low  $\rightarrow$  high.

```
def fib(n):  
    fibs = [0, 1]  
    for i in range(2, n+1):  
        fibs.append(fibs[i-1] + fibs[i-2])  
    return fibs[n]
```

Time:  $O(n)$

Space:  $O(n)$

"sliding window" DP:

```
def fib(n):  
    a, b = 0, 1  
    for i in range(2, n+1):  
        a, b = b, a+b  
    return b
```

Time:  $O(n)$

space:  $O(1)$

# Matrix Exponentiation Method.

Can view each step of the sliding window method as a matrix multiply.

$$\begin{bmatrix} a \\ b \end{bmatrix} = \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix} \cdot \begin{bmatrix} a \\ b \end{bmatrix}$$

The final  $\begin{bmatrix} a \\ b \end{bmatrix}$  is thus  $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n-1} \begin{bmatrix} 0 \\ 1 \end{bmatrix}$

$$\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^2 = \begin{bmatrix} 1 & 1 \\ 1 & 2 \end{bmatrix}, \quad \begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^3 = \begin{bmatrix} 1 & 2 \\ 2 & 3 \end{bmatrix}, \quad \begin{bmatrix} 2 & 3 \\ 3 & 5 \end{bmatrix}, \dots$$

$F_n =$  bottom right of  $\begin{bmatrix} 0 & 1 \\ 1 & 1 \end{bmatrix}^{n-1}$ .

How quickly can we compute  $A^n$  for a  $2 \times 2$  matrix  $A$ ?

Repeated Squaring:

First compute  $A, A^2, A^4 = (A^2)^2, A^8 = (A^4)^2, \dots$

up to  $A^{2^k}$  for  $k = \lfloor \log_2 n \rfloor$

Express  $n$  in base 2:

$$n = \sum_{i=0}^k a_i \cdot 2^i$$

$$\text{Then } A^n = \left( \sum_{i=0}^k a_i \cdot 2^i \right) = \prod_{i=0}^k A^{a_i \cdot 2^i} = \prod_{i: a_i=1} A^{2^i}$$

Example:

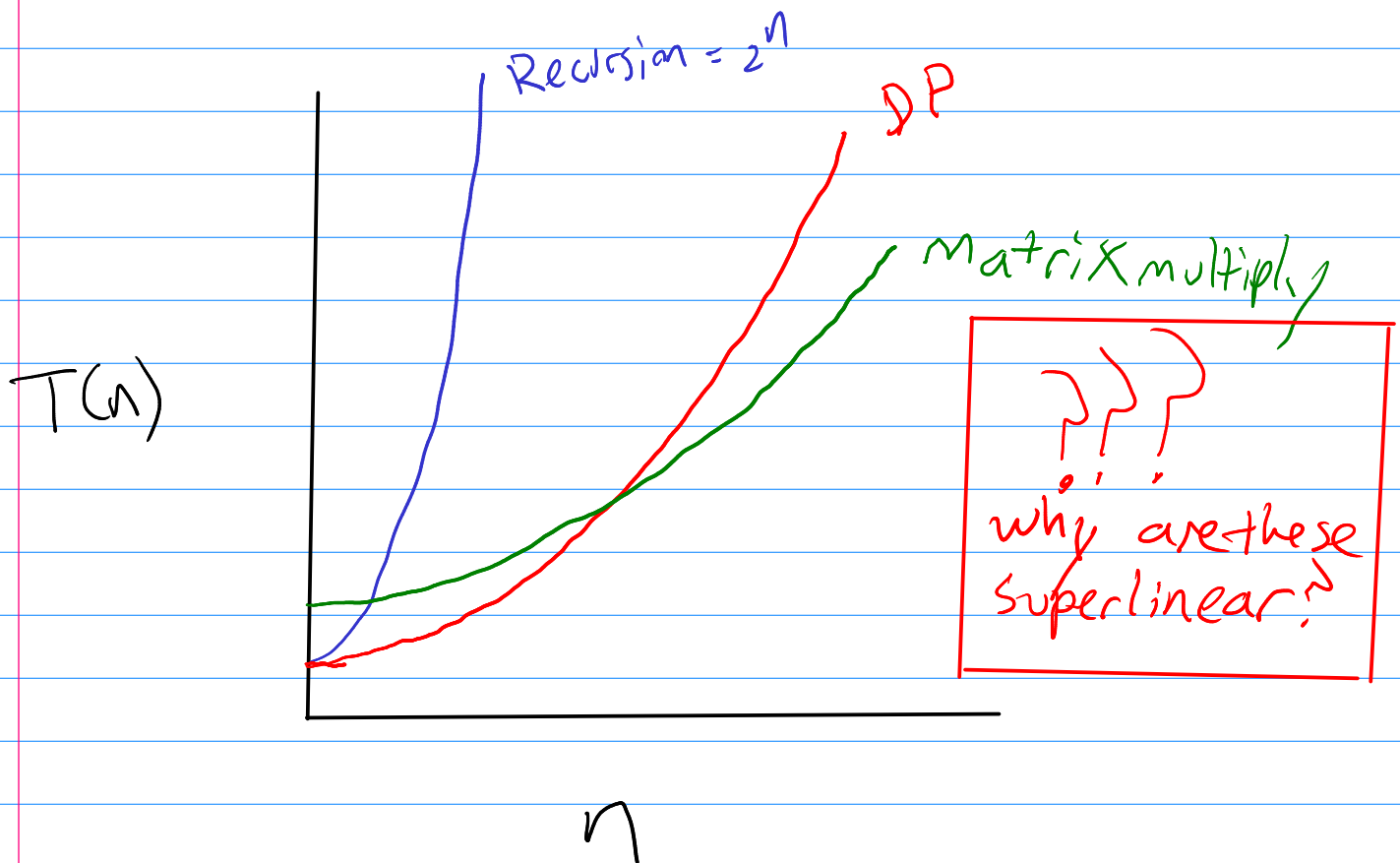
Suppose  $n = \underbrace{101101}_2 = 45$   
K bits

Then  $A^{45} = A^{32} \cdot A^8 \cdot A^4 \cdot A$

Total time =  $O(\log n)$   $2 \times 2$  matrix  
multiplications  
 $= O(\log n)$ .

---

Easy to implement these algorithms,  
benchmark time.



All the preceding time/space bounds are wrong

Because the numbers get big.

$F_n \approx 1.6^n$  takes  $\Theta(n)$  bits to store.

DP:  $\sum_{i=1}^n n = \boxed{\Theta(n^2) \text{ time}}$

Matrix Multiplication:

the

$$A^{2^{k+1}} = (A^{2^k})^2$$

step involves multiplying  $\Theta(2^k)$ -bit numbers.

Then Fibonacci has

$$\begin{aligned} T(n) &= T\left(\frac{n}{2}\right) + (\text{multiplication time for } n \text{ bits}) \\ &= \Theta(\text{multiplication time for } n \text{ bits}) \end{aligned}$$

(!+ this is  $\gg n^{1.01}$ )

Elementary:  $\Theta(n^2)$

Karatsuba:  $\Theta(n^{\log_2 3})$

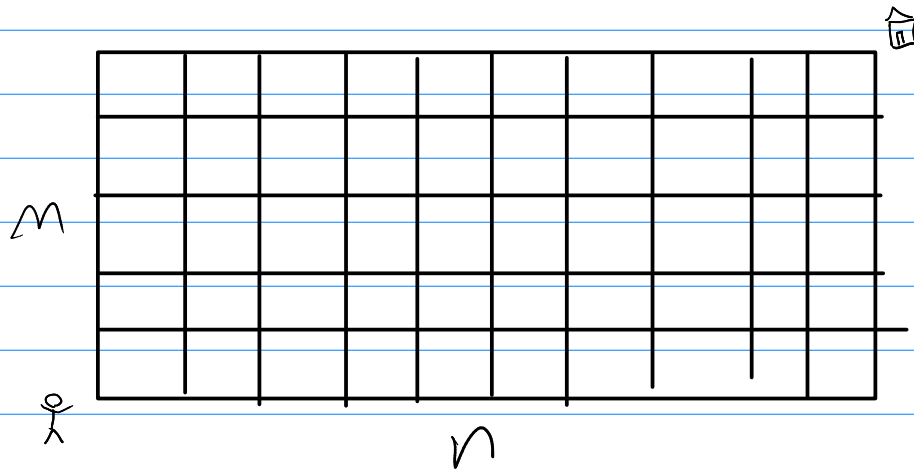
For the rest of this course, avoid this issue  
- only deal with numbers bounded by  $n^{O(1)}$

- Assume computer can manipulate such numbers in  $O(1)$  time

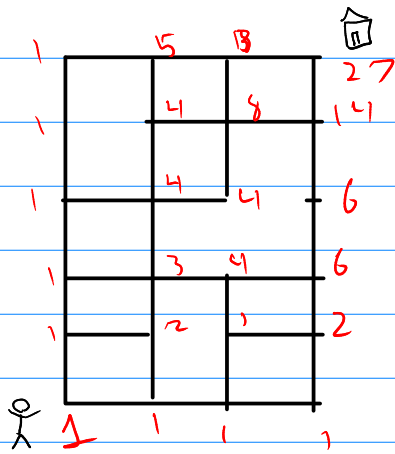
[64-bit word size,  $n < 2^{64}$ ]

Questions:

How many paths, moving only up and right?



Now?



Catalan numbers: # valid parentheses of length  $n$

Delannoy: # paths  $(0,0)$  to  $(n,m)$   
up, right, or up&right ( $\mathbb{Z}$ )

Schröder number: Delannoy( $n,n$ ) & validly parenthesized

In general: #  $S \rightarrow T$  paths on a DAG