Fibonacci Numbers

\[ F_0 = 0, \quad F_1 = 1 \]
\[ F_n = F_{n-1} + F_{n-2}. \]

How quickly can we compute the \( n \)th Fibonacci number?

Naive recursion:

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

Computation graph:

More compact view:

"state" \( n \)

\[ f(n) \quad 0 \quad 1 \quad 2 \quad 3 \quad \cdots \quad n-3 \quad n-2 \quad n-1 \quad n \]
Returns to $f(i)$ again and again
\[
T(n) = T(n-1) + T(n-2) + \text{Something } \geq 1
\]

\[
\Rightarrow T(n) \geq \sum_{k=0}^{\lfloor n/2 \rfloor} F_k > 2^{n/2} \approx 1.6^n
\]

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 → 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.

\[
\begin{bmatrix}
a \\
b
\end{bmatrix} = \begin{bmatrix}
0 & 1 \\
1 & 1
\end{bmatrix} \begin{bmatrix}
a \\
b
\end{bmatrix}
\]

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

\[
\begin{bmatrix}
0 & 1 \\
1 & 1
\end{bmatrix}^2 = \begin{bmatrix}
1 & 1 \\
1 & 2
\end{bmatrix}, \quad \begin{bmatrix}
0 & 1 \\
1 & 1
\end{bmatrix}^3 = \begin{bmatrix}
1 & 2 \\
2 & 3
\end{bmatrix}, \quad \begin{bmatrix}
2 & 3 \\
3 & 5
\end{bmatrix}.
\]

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

How quickly can we compute \( A^n \) for a 2x2 matrix \( A \)?

Repeated Squaring:

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

up to \( A^{2^k} \) for \( k = \lfloor \log_2 n \rfloor \).

Express \( n \) in base 2:

\( n = \sum_{i=0}^{K} a_i \cdot 2^i \)

Then \( A^n = \prod_{i=0}^{K} A^{a_i \cdot 2^i} = \prod_{i, a_i = 1} A^{a_i \cdot 2^i} \).
Example:

Suppose \( N = \underbrace{1011010}_{12} = 45 \) \((K = 5)\) 

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

Total time = \( O(\log n) \) 2x2 matrix multiplications \( = O(\log n) \).

---

Easy to implement these algorithms benchmark time.

\[ T(n) \]

[Graph showing comparison of recursion, matrix multiply, and dynamic programming (DP) with annotations: "why are these superlinear?" and "Recursion = 2^n".]
All the preceding time/space bounds are wrong.

Because the numbers get big,

\[ F_n \approx 1.6^n \quad \text{takes } \Theta(n) \text{ bits to store.} \]

\[
\text{DP:} \quad \prod_{i=1}^{n} \frac{1}{z_i} = \Theta(n^2) \text{ time}
\]

Matrix Multiplication:

The \(2^{k+1}\times 2^{k+1}\) step involves multiplying \(\Theta(2^k)\)-bit numbers.

Then Fibonacci has

\[
T(n) = T\left(\frac{n}{2}\right) + (\text{multiplication time for } n \text{ bits})
\]

\[
= \Theta(\text{multiplication time for } n \text{ bits}) \quad (\text{since } n > 2^{n/2})
\]

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

\([64\text{-bit word size, } n < 2^{64}]\)