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?

Naïve recursion:

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

Computation graph:

More compact view:

"State" n: \( 0, 1, 2, 3, \ldots, n-3, n-2, n-1, n \)

\( f(n) \): 0, 1, 1, 2, 3, 5, \ldots, \( F_{n-3}, F_{n-2}, F_{n-1}, F_n \)
Returns to $f(1)$ again and again

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

\[ \implies T(n) \geq F_n > 2^{n/2} \quad (\approx 1.6^n) \]

How can we avoid this repetition?

Only compute each $F_i$ once.

\[ f(3) \text{ never changes. Can store the answer,} \]

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 to high.

```python
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:

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

Time: $O(1)$
Space: $O(1)$
Matrix Exponentiation Method.

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

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

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

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

\( F_n = \) 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 \( A, A^2, A^4, A^8, (A^4)^2 \).

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

Suppose \( N = \sqrt{101101_2} = 45 \) K bits

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.
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: \( \prod_{i=1}^n n = \Theta(n^2) \) plus time

Matrix multiplication:

the \( 2^{k+1} \)

\[ A^1 = (A^k)^2 \]

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

Then Fibonacci has

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

\[ = \Theta(\text{multiplication time for } n \text{ bits}) \]

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

\[ [\text{64-bit word size, } n \leq 2^{64}] \]