Recursion
Computing the $n^{\text{th}}$ Fibonacci number

def fibo(n):
    # Base cases
    if n == 1 or n == 2:
        return 1
    # Recursive case
    else:
        return fibo(n-1) + fibo(n-2)
Efficiency of this function

- Sometimes recursive functions can be extremely inefficient.

- This is because the same sub-problem may be solved many times and our program has no recollection of having solved the same sub-problem earlier.

- This type of inefficiency shows up in the recursive implementation of the Fibonacci function.
Exponential growth: running time of fibo(n)

The plot shows the running time (in seconds) of fibo(n) for n = 20, 21,..., 30
Partial Recursion Tree for fibo(8)
Notice how the same sub-problem is solved many times. E.g., \texttt{fibo}(4) is solved 5 times, etc.

In later CS classes (e.g., 22C:19, 22C:21, or 22C:31) you will learn how to analyze recursive functions.

You will be able to show that the running time of this implementation of \texttt{fibo}(n) is exponential in \( n \), i.e., \( c^n \), where \( c \) is some constant larger than 1.
Example: Efficiently computing the power function

- The example involving Fibonacci numbers showed us that recursion can sometimes lead to inefficient programs.

- But recursion can also lead to tremendous efficiency.

**Problem:** Given a real number $a$ and a non-negative integer $n$, compute $a^n$. 
Computing $a^n$ efficiently

- It is easy to write a loop that performs $n-1$ multiplications to compute $a^n$.

- However, we can compute $a^n$ much more efficiently.

- **Example:** To compute $a^{32}$, we could compute $a^{16}$ first and then use one multiplication to square it. To compute $a^{16}$, we would compute $a^8$ and then use one multiplication to square it...
It takes 5 multiplications to compute $a^{32}$

- Each multiplication reduces the power to $\frac{1}{2}$ of what it was earlier.
- This is a common theme in many efficient algorithms: if we can shrink the size of the problem to $\frac{1}{2}$ of what it was quickly, then we have an efficient problem.
What about an odd power?

- **Example:** Compute $a^{39}$

- We could compute $a^{19}$, square it (using one multiplication) to get $a^{38}$ and then use another multiplication to compute $a^{39}$.

- Thus using 2 multiplications we can reduce the power to about $\frac{1}{2}$ of what it was earlier.
def power(a, n):
    # Base cases
    if n == 0:
        return 1
    if n == 1:
        return a

    # Recursive case: n is even
    if n % 2 == 0:
        temp = power(a, n/2)
        return temp * temp

    # Recursive case: n is odd
    if n % 2 == 1:
        temp = power(a, n/2)
        return a * temp * temp