The fibonacci function

def fibonacci(n):
    # Base cases
    if n == 1 or n == 2:
        return 1
    else:
        # Recursive case
        return fibonacci(n-1) + fibonacci(n-2)
Partial Recursion Tree for fibonacci(8)
A puzzle

- Implement the fibonacci function non-recursively.

- Compare the running times of the recursive and non-recursive functions.

- What do you find? How might you explain the results?
def fibonacci(n):
    if n == 1 or n == 2:
        return 1
    current = 1
    previous = 1
    count = 2
    while count < n:
        temp = current
        current = previous + current
        previous = temp
        count = count + 1
    return current
Inefficiency of the Recursive Function

- Sometimes recursive functions can be extremely inefficient. The fibonacci function is an example.

- The (partial) recursion tree for fibonacci(8) hints at why this might be: the same problem is being solved many times.

- The function has no recollection of having solved the problem earlier!
Efficiency of fibonacci(n): Exponential Growth

The plot shows the running time (in seconds) of fibo(n) for n = 20, 21,…, 30.
Example: Efficiently computing the *power* function

- The *fibonacci* example showed that sometimes recursion can be extremely inefficient compared.

- However, sometimes a recursive-approach can lead to tremendous gains in efficiency.

- We will see several important examples of this.

- **Problem:** Given a real number $a$ and a nonnegative 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 $a^{32}$

- For each recursive call, we perform just one multiplication, but manage to reduce the problem size to $1/2$ its previous size.

- This is a common theme in many efficient algorithms: if we can do just a little work and manage to shrink the problem size down to $1/2$ its original size, then we have an efficient solution.
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 less than $\frac{1}{2}$ of what it was earlier.
def power(a, n):
    # Base Cases
    if n == 0:
        return 1
    if n == 1:
        return a
    # Recursive Case: even n
    if n % 2 == 0:
        temp = power(a, n/2)
        return temp*temp
    if n % 2 == 1:
        temp = power(a, n/2)
        return temp*temp*a