Paso a Paso: Funcionamiento de la recursividad con Fibonacci en Python

Vamos a analizar paso a paso lo que ocurre cuando llamamos a la función `fibonacci(5)` usando recursividad.

Definición de la función:

```
def fibonacci(n):
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        return fibonacci(n - 1) + fibonacci(n - 2)
```

Llamada inicial:

```
fibonacci(5)
```

Proceso de ejecución:

Para calcular `fibonacci(5)` la función hace:

- `fibonacci(5) = fibonacci(4) + fibonacci(3)`

Ahora veamos cómo se expande cada llamada:

1. `fibonacci(4) = fibonacci(3) + fibonacci(2)`
2. `fibonacci(3) = fibonacci(2) + fibonacci(1)`
3. `fibonacci(2) = fibonacci(1) + fibonacci(0)`

Ahora resolvemos los casos base:

- `fibonacci(1) = 1`
- `fibonacci(0) = 0`

Recomponiendo:

- `fibonacci(2) = 1 + 0 = 1`
- `fibonacci(3) = 1 + 1 = 2`
- `fibonacci(4) = 2 + 1 = 3`
- `fibonacci(3)` (desde la rama original) = `2`
- `fibonacci(5) = 3 + 2 = 5`

Representación visual del árbol de llamadas:

```
fibonacci(5)
├── fibonacci(4)
│   ├── fibonacci(3)
│   │   ├── fibonacci(2)
│   │   │   ├── fibonacci(1) → 1
│   │   │   └── fibonacci(0) → 0
│   │   └── fibonacci(1) → 1
│   └── fibonacci(2)
```

```
│        ├── fibonacci(1) → 1
│        └── fibonacci(0) → 0
└── fibonacci(3)
    ├── fibonacci(2)
    │   ├── fibonacci(1) → 1
    │   └── fibonacci(0) → 0
    └── fibonacci(1) → 1
```

Observaciones importantes:

- La función `fibonacci` hace muchas llamadas repetidas (como `fibonacci(2)` y `fibonacci(1)`).
- Esto provoca un crecimiento exponencial en la cantidad de llamadas, lo cual es ineficiente.
- Puede mejorarse usando **memoización** o una versión iterativa.

Conceptos clave:

- Cada llamada se divide en dos nuevas llamadas (excepto los casos base).
- El árbol crece rápidamente con `n`.
- Aunque el código es simple, no es el más eficiente para grandes valores de `n`.

Este ejemplo es muy útil para entender la recursividad y también para aprender a optimizar algoritmos recursivos.