Recursividad

Algoritmos y Estructuras de Datos

Por: Violeta Ocegueda

Definición

- Una función recursiva es aquella que se invoca a sí misma.
- Permite expresar algoritmos complejos en forma compacta y elegante sin reducir la eficiencia.
- Un algoritmo recursivo es aquel que resuelve un problema resolviendo una o más instancias menores que el mismo problema.

Características

- **Caso base:** siempre debe existir casos base que se resuelven sin hacer uso de la recursión.
- **Caso recursivo:** cualquier llamada recursiva debe progresar hacia un caso base.
- **Diseño:** asumir que toda llamada recursiva interna funciona correctamente.
- **Regla de Interés compuesto:** evitar duplicar el trabajo resolviendo la misma instancia de un problema en llamadas recursivas compuestas.

Calcular el factorial de n

- El factorial de un entero positivo n es el producto de todos los números enteros positivos desde 1 hasta n.
- $n! = 1 \times 2 \times 3 \times 4 \times \ldots \times (n - 1) \times n$
- También se le define como:

$$n! = \begin{cases} 1 & si\ n = 0 \\ (n-1)! \times n & si\ n > 0 \end{cases}$$

- $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120 \leftrightarrow 5! = 5 \times 4!$
- $4! = 4 \times 3 \times 2 \times 1 \leftrightarrow 4! = 4 \times 3!$
- $3! = 3 \times 2 \times 1 \leftrightarrow 3! = 3 \times 2!$
- $2! = 2 \times 1 \leftrightarrow 2! = 2 \times 1!$
- $1! = 1$

Calcular el factorial de n

```
int factorial (int n)
{
      if ( n == 0 || n == 1 ) // caso base
            return 1;
      else // caso recursivo
            return n * factorial ( n – 1 );
}
```

Corrida de escritorio

Para n = 3

factorial(3) = 3 * factorial(2)
factorial(2) = 2 * factorial(1)
factorial(1) = 1

Y la complejidad?

Método 1

```
int factorial (int n)
{
      if ( n == 0 || n == 1 ) // caso base
          return 1;
      else // caso recursivo
          return n * factorial ( n – 1 );
}
```

1

2 + llamada recursiva

Corrida de escritorio

Para n=5

factorial(5) → 2

return 5 * factorial(4) → 2

return 4 * factorial(3) → 2

return 3 * factorial(2) → 2

return 2 * factorial(1) → 1

1

9

operaciones

Qué hace en cada llamada recurrente?

Y la complejidad? **Método 1**

```
int factorial (int n)
{
      if ( n == 0 || n == 1 ) // caso base
           return 1;
      else // caso recursivo
           return n * factorial ( n - 1 );
}
```

1

2 + llamada recursiva

Corrida de escritorio

Para n=5

factorial(5)

return 5 * factorial(4)

return 4 * factorial(3)

return 3 * factorial(2)

return 2 * factorial(1)

1

5

Cuantas llamadas recursivas hay?

Y la complejidad?

Método 1

Corrida de escritorio

```
int factorial (int n)
{
      if ( n == 0 || n == 1 ) // caso base
          return 1;
      else // caso recursivo
          return n * factorial ( n - 1 );
}
```

1

2 + llamada recursiva

Para n=5

factorial(5)

return 5 * factorial(4)

return 4 * factorial(3)

return 3 * factorial(2)

return 2 * factorial(1)

return 1

5

Tiempo = 9

LLamadas = 5

2(5) – 1 = 9

2n - 1

T(n) = 2n-1 = O(n)

Cómo se relacionan el tiempo que duró la corrida con respecto a la cantidad de veces que se llamó a la función recursiva?

Y la complejidad?

Método 2

T(n)
```
int factorial (int n)
{
      if ( n == 0 || n == 1 ) // caso base
            return 1;
      else // caso recursivo
            return n * factorial ( n – 1 );
}
```

1

2 + T(n-1)

T(n-1) + 3

Analizar esto así se volvería complicado, así que lo vamos a redondear a su notación sintótica, esto es T(1).

Relación de recurrencia

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-1) + 1 & n > 1 \end{cases}$$

Y la complejidad? Método 2

Relación de recurrencia

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-1) + 1 & n > 1 \end{cases}$$

Ahora vamos a resolver la relación por el método de sustitución:

Sustituyo T(n) en T(n-1):

T(n-1 -1) + 1 = T(n-2) + 1 $\longrightarrow$ T(n-1)

Ahora sustituyo T(n-1) en T(n)

T(n) = [T(n-2) + 1] + 1 = T(n-2) + 2

Sustituyo otra vez T(n-1) en T(n)

T(n) = [T(n-1 -2) + 1] + 2 = [T(n-3) + 1] + 2 = T(n-3) + 3

Continúo hasta k....

T(n) = T(n-k) + k

Para n>1:

T(n) = T(n-1) + 1

T(n-1) = T(n-2) + 1

T(n-2) = T(n-3) + 1

Cuándo dejo de sustituir?

Cuando encuentre el menor valor posible, para este caso es cuando:

n – k = 0

Y la complejidad? **Método 2**

Relación de recurrencia

$$T(n) = \begin{cases} 1 & n \leq 1 \\ T(n-1) + 1 & n > 1 \end{cases}$$

T(n) = T(n-k) + k

Cuándo dejo de sustituir?

Cuando encuentre el menor valor posible, para este caso es cuando:

n – k = 0

↓

n = k

Entonces:

T(n) = T(n-k) + k
T(n) = T(n-n) + n
T(n) = T(0) + n
T(n) = 1 + n

Por lo tanto:

T(n) = O(n)

Calcular el n-ésimo término de la sucesión Fibonacci

- Comienza con los números 0 y 1, y a partir de estos, cada término es la suma de los dos anteriores.
  - $0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987, 1597, \ldots$
- La sucesión se puede definir por las siguientes ecuaciones:
  - $f_0 = 0$
  - $f_1 = 1$
  - $f_n = f_{n-1} + f_{n-2}$
- De tal manera que:
  - $f_2 = f_1 + f_0 = 1 + 0 = 1$
  - $f_3 = f_2 + f_1 = 1 + 1 = 2$
  - $f_4 = f_3 + f_2 = 2 + 1 = 3$
  - $f_5 = f_4 + f_3 = 3 + 2 = 5$
  - $f_6 = f_5 + f_4 = 5 + 3 = 8$

Calcular el n-ésimo término de la sucesión Fibonacci

```
int fibonacci (int n)
{
      if ( n == 0 || n==1 ) // caso base
            return n;
      else // caso recursivo
            return fibonacci ( n – 1 ) + fibonacci ( n – 2 );
}
```

Corrida de escritorio

Para n = 6

fibonacci(6) = fibonacci(5) + fibonacci(4)
fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(4) = fibonacci(3) + fibonacci(2)
fibonacci(3) = fibonacci(2) + fibonacci(1)
fibonacci(2) = fibonacci(1) + fibonacci(0)
fibonacci(1) = 1
fibonacci(0) = 0

Calculando la complejidad de Fibonacci

```
int fibonacci (int n)
{
      if ( n == 0 || n==1 ) // caso base
            return n;
      else // caso recursivo
            return fibonacci ( n - 1 ) + fibonacci ( n - 2 );
}
```

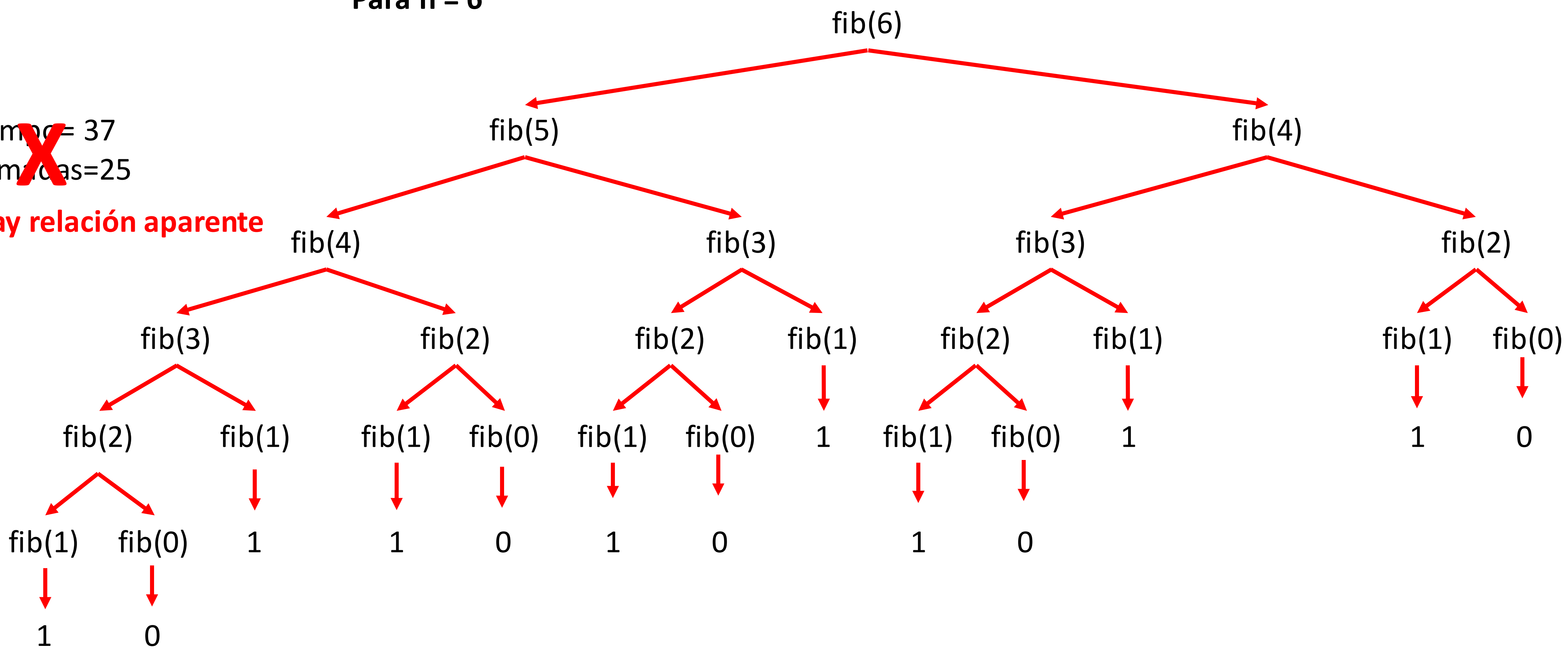
1

2 + 2 llamadas recursivas

Calculando la complejidad de Fibonacci

Corrida de escritorio

Para n = 6



Tiempo = 37

Llamadas=25

X

No hay relación aparente

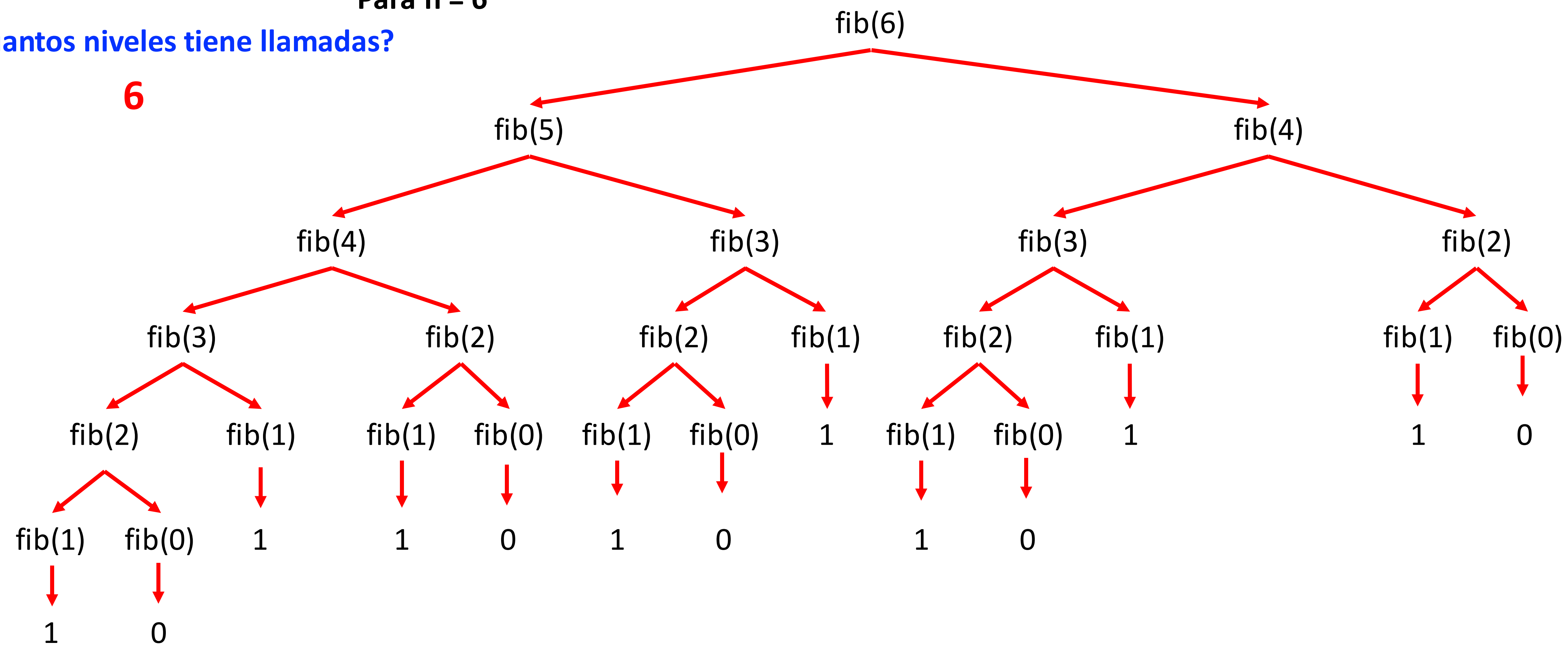
Calculando la complejidad de Fibonacci

Corrida de escritorio

Para n = 6

En cuantos niveles tiene llamadas?

6

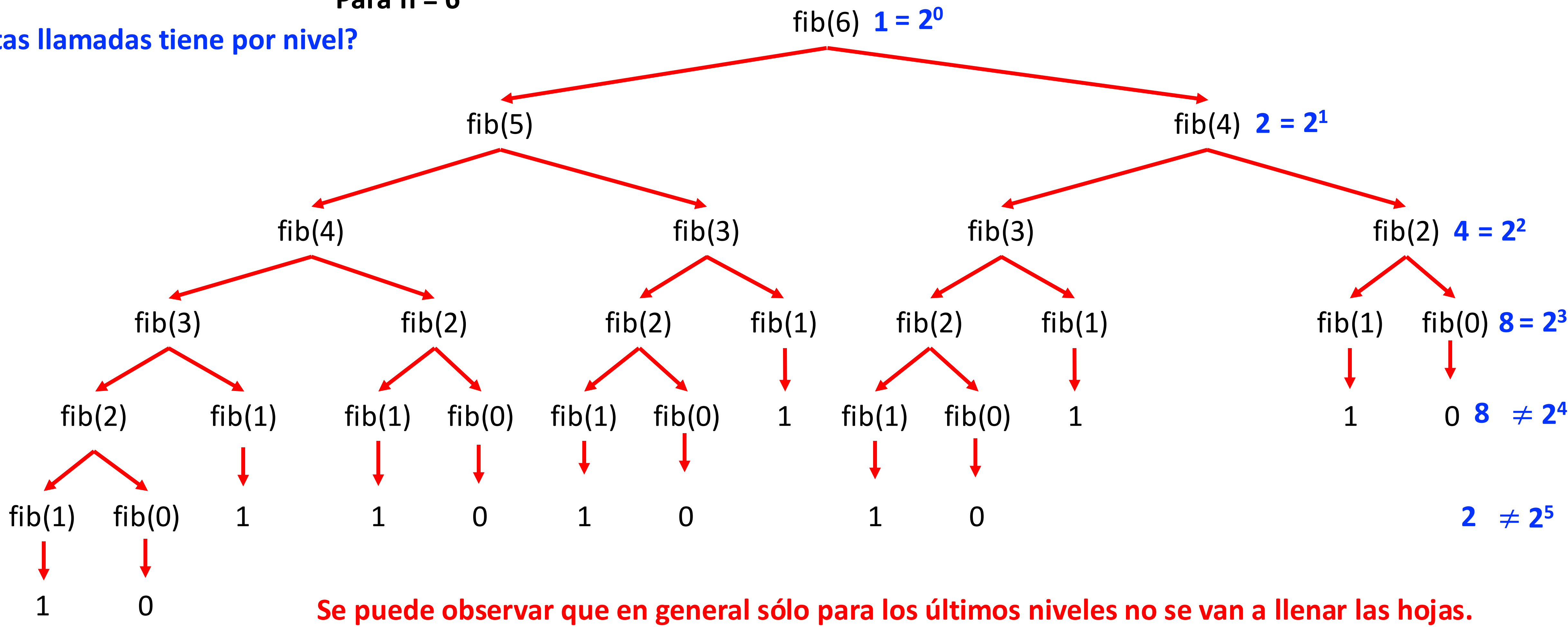


Calculando la complejidad de Fibonacci

Corrida de escritorio

Para n = 6

Cuántas llamadas tiene por nivel?



Se puede observar que en general sólo para los últimos niveles no se van a llenar las hojas.

Calculando la complejidad de Fibonacci

$$T(n) = 2^0 + 2^1 + 2^2 + \cdots + 2^{n-1}$$

Nos quedamos con el término de mayor grado

$$T(n) = O(2^n)$$

Ejercicios

- Escriba una función recursiva que reciba como parámetro una lista de números enteros y un número entero a eliminar. La función debe permitir eliminar todos los elementos iguales al número entero seleccionado.
- Escriba una función recursiva que reciba como parámetro una cadena y la devuelva invertida.
- Escriba una función recursiva que reciba como parámetro una lista de números enteros y devuelva la lista sin elementos repetidos.
- Escriba una función recursiva que reciba como parámetro una cadena y compruebe si la cadena es o no un palíndromo.