Recursividad y Backtracking

Diego Angulo Ramirez

11 de Enero de 2024

Recursividad:

Una funcion recursiva es una funcion, que esta definida en terminos de si misma.

Es decir, en terminos de programacion, una funcion que se llama a si misma para calcular un resultado.

Por ejemplo:

$$f(n) = \begin{cases} 0 & \text{si } n = 0 \\ f(n-1) + 1 & \text{si } n > 0 \end{cases}, n \in \mathbb{N}$$

Claro, esta funcion tambien podria ser definida como:

$$f(n) = n, n \in \mathbb{N}$$

En codigo:

Codigo C++

```
int f(int n){
    if(n == 0)
        return 0;
    else
        return f(n - 1) + 1;
}
```

DATAZO!

Toda funcion recursiva necesita un caso base.
Sin caso base tendremos una recursion infinita y sin un resultado predecible.

Ejemplo

```
int f(int n) {
    if (n % 2 == 0)
        return f(n / 2);
    else
        return f(n - 1);
}
```

Cuando usar recursividad

La recursividad es util para resolver problemas que pueden ser divididos en problemas mas pequeños, que sigan siendo del mismo tipo.
Pero es importante tener mucho cuidado al momento de usarlas:

- El problema puede ser dividido en problemas mas pequeños del mismo tipo.
- El problema tiene casos base.
- Un caso base se alcanza antes de que se llegue al limite del tamaño del stack.

Cada vez que se llama a una funcion en un lenguaje de programacion, esta se aloca en el stack de llamadas(estructura de datos donde se almacena la dirección de retorno cada vez que se llama una función, además del valor de variables locales, parámetros, etc.)
Obviamente, este stack tiene cierto limite de tamaño.

Factorial:

Si definimos factorial:

$$f(n) = n * (n-1) * (n-2) * ... * 1, n \in \mathbb{N}$$

Y definido recursivamente, seria:

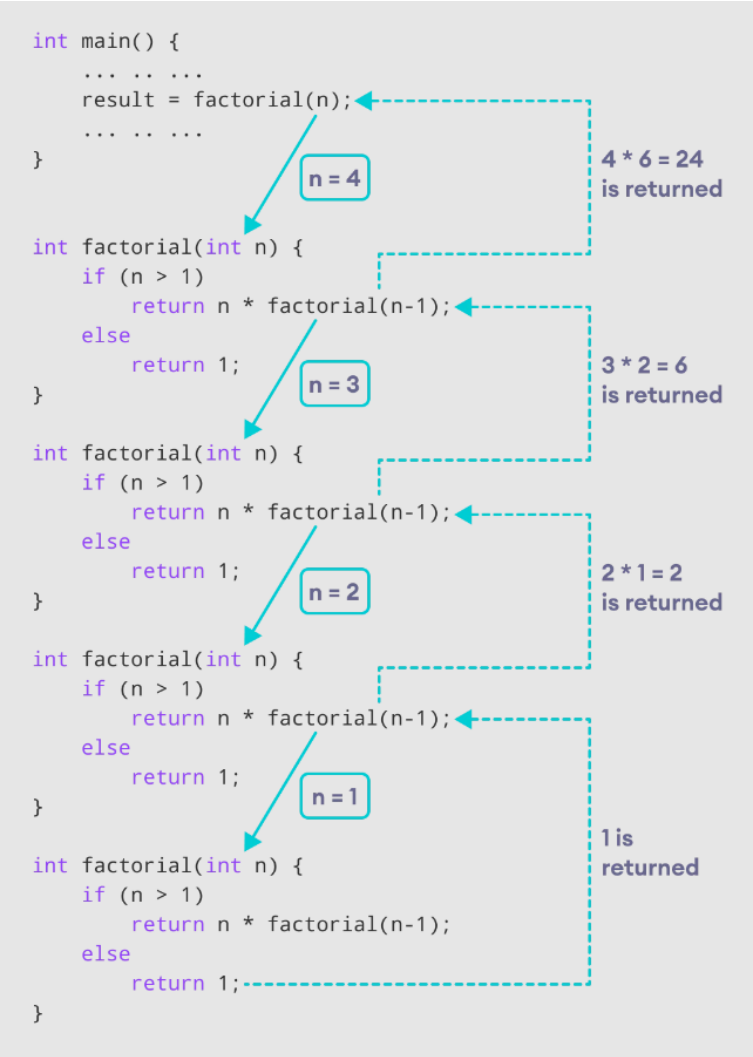
$$f(n) = \begin{cases} 1 & \text{si } n = 1 \\ f(n-1) * n & \text{si } n > 1 \end{cases}, n \in \mathbb{N}$$

Factorial

Codigo C++:

```
int f(int n){
    if(n == 1)
        return 1;
    else
        return f(n - 1) * n;
}
```

Factorial


int main() {
... .
result = factorial(n);
... .
}
n = 4
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
n = 3
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
n = 2
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
n = 1
int factorial(int n) {
if (n > 1)
return n * factorial(n-1);
else
return 1;
}
4 * 6 = 24
is returned
3 * 2 = 6
is returned
2 * 1 = 2
is returned
1 is
returned

Factorial

$$f(5) = f(4) * 5$$

Factorial

$$f(5) = f(4) * 5$$

$$f(4) = f(3) * 4$$

Factorial

$$f(5) = f(4) * 5$$

$$f(4) = f(3) * 4$$

$$f(3) = f(2) * 3$$

Factorial

$$f(5) = f(4) * 5$$

$$f(4) = f(3) * 4$$

$$f(3) = f(2) * 3$$

$$f(2) = f(1) * 2$$

Factorial

$$f(5) = f(4) * 5$$

$$f(4) = f(3) * 4, \quad f(5) = f(3) * 4 * 5$$

$$f(3) = f(2) * 3, \quad f(5) = f(2) * 3 * 4 * 5$$

$$f(2) = f(1) * 2, \quad f(5) = f(1) * 2 * 3 * 4 * 5$$

$$f(1) = 1, \quad f(5) = 1 * 2 * 3 * 4 * 5$$

$$f(5) = 5 * 4 * 3 * 2 * 1$$

Factorial

$$f(5) = 5 * 4 * 3 * 2 * 1$$

Fibonacci: Función Recursiva

- La serie de Fibonacci es una secuencia matemática definida recursivamente.
- Cada término es la suma de los dos términos anteriores.
- La función recursiva se define como sigue:

$$\text{Fib}(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ \text{Fib}(n-1) + \text{Fib}(n-2) & \text{si } n > 1 \end{cases}$$

Implementación en C++

Función Recursiva

```
int fibonacci(int n) {
  if (n == 0) return 0;
  else if (n == 1) return 1;
  else return fibonacci(n - 1) + fibonacci(n - 2);
}
```

- La función en C++ refleja la definición recursiva.
- Puede ser ineficiente para valores grandes de n debido a la repetición de cálculos.

Visualización

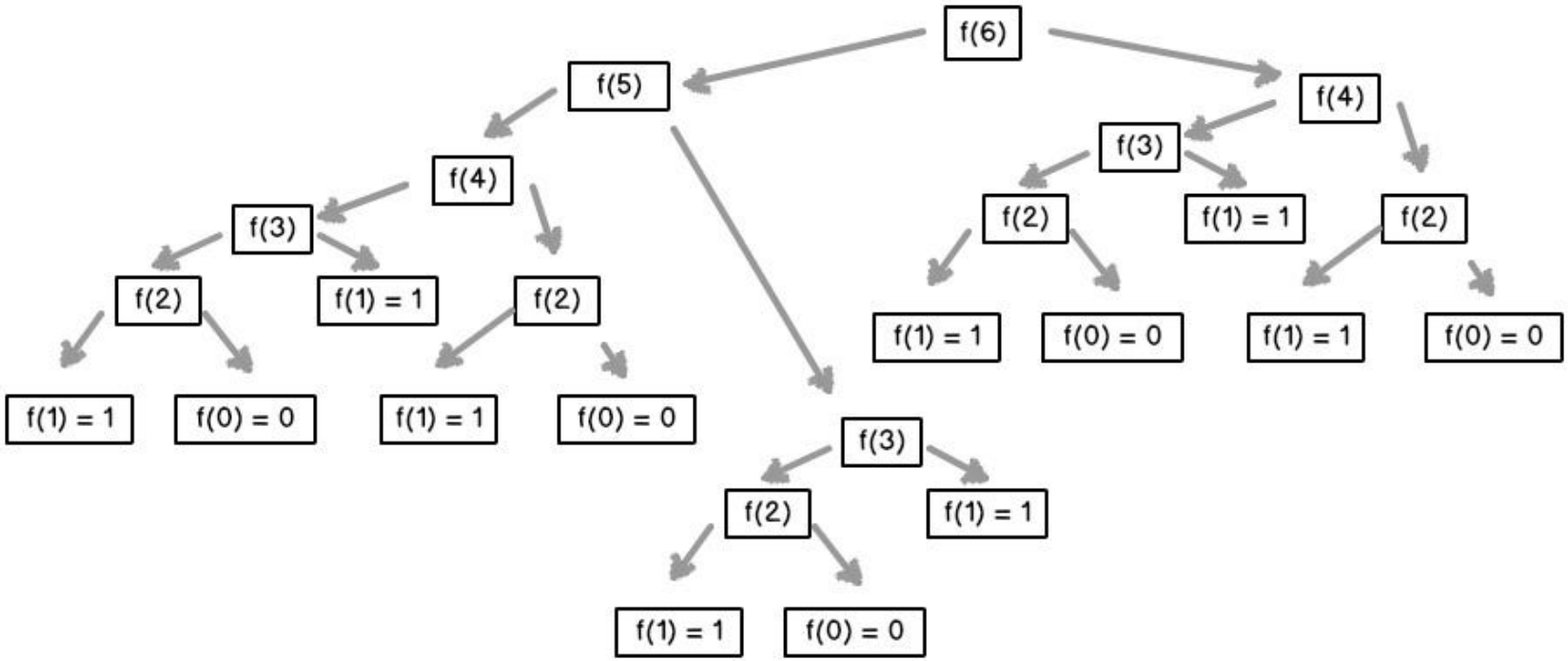


Figure: Llamadas recursivas de la funcion

La recursividad, es buena?

NO CALCULES, LO QUE YA CALCULASTE

Memoizacion

Guarda los resultados que ya conseguiste.
Si ya sabes la respuesta de factorial, para un valor de 5, guardalo!
Como? Memoizacion.

Factorial con memoizacion

```
vector<long long> memo(25, -1);

long long f(int n) {
    if (n == 1)
        return 1;
    if (memo[n] != -1)
        return memo[n];

    return memo[n] = f(n - 1) * n;
}
```

Memoizacion

Fibonnaci con memoizacion

```
vector<long long> memo(10000, -1);

int fibonacci(int n) {
  if (n == 0)
    return 0;
  if (n == 1)
    return 1;
  if (memo[n] != -1)
    return memo[n];
  return memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
}
```

Memoizacion

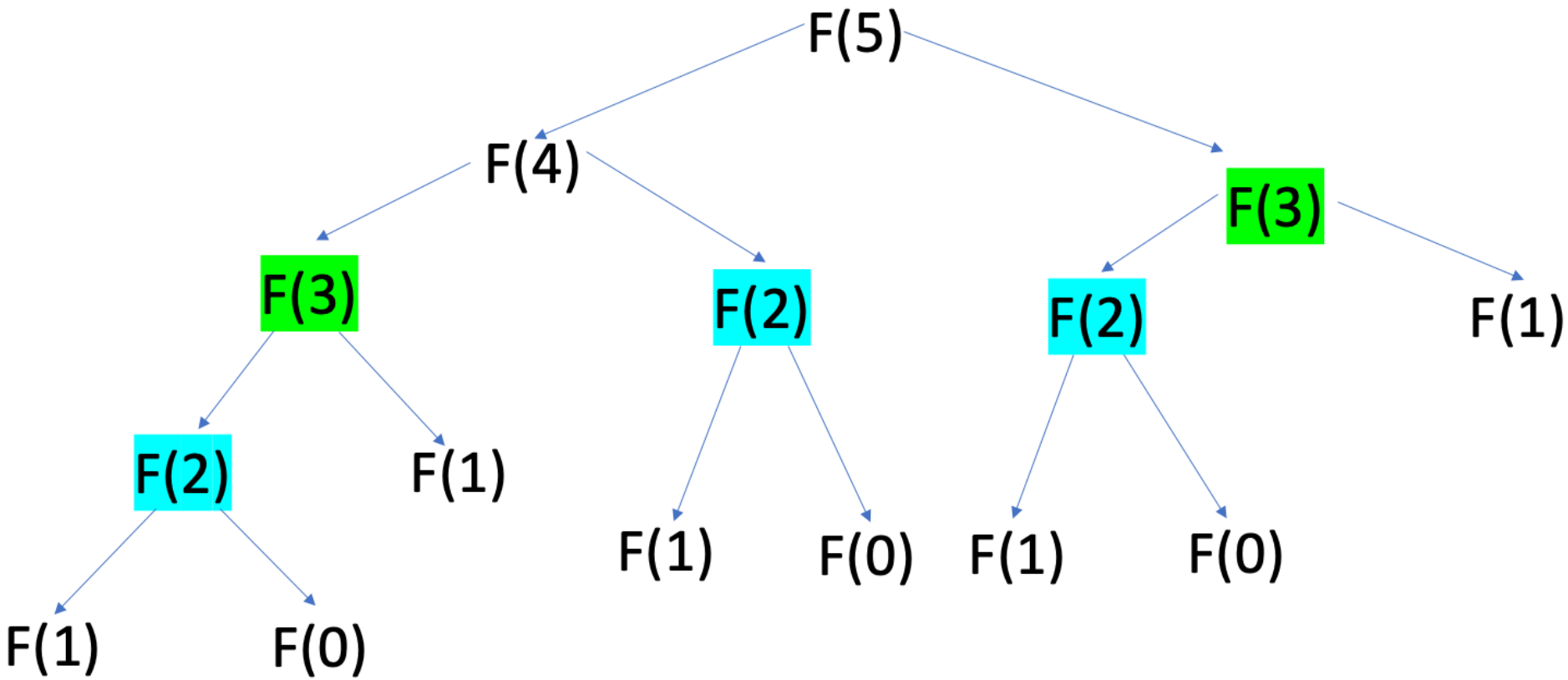
Fibonnaci con memoizacion

```
vector<long long> memo(10000, -1);
memo[0] = 0;
memo[1] = 1;

int fibonacci(int n) {
  if (memo[n] != -1)
    return memo[n];
  return memo[n] = fibonacci(n - 1) + fibonacci(n - 2);
}
```

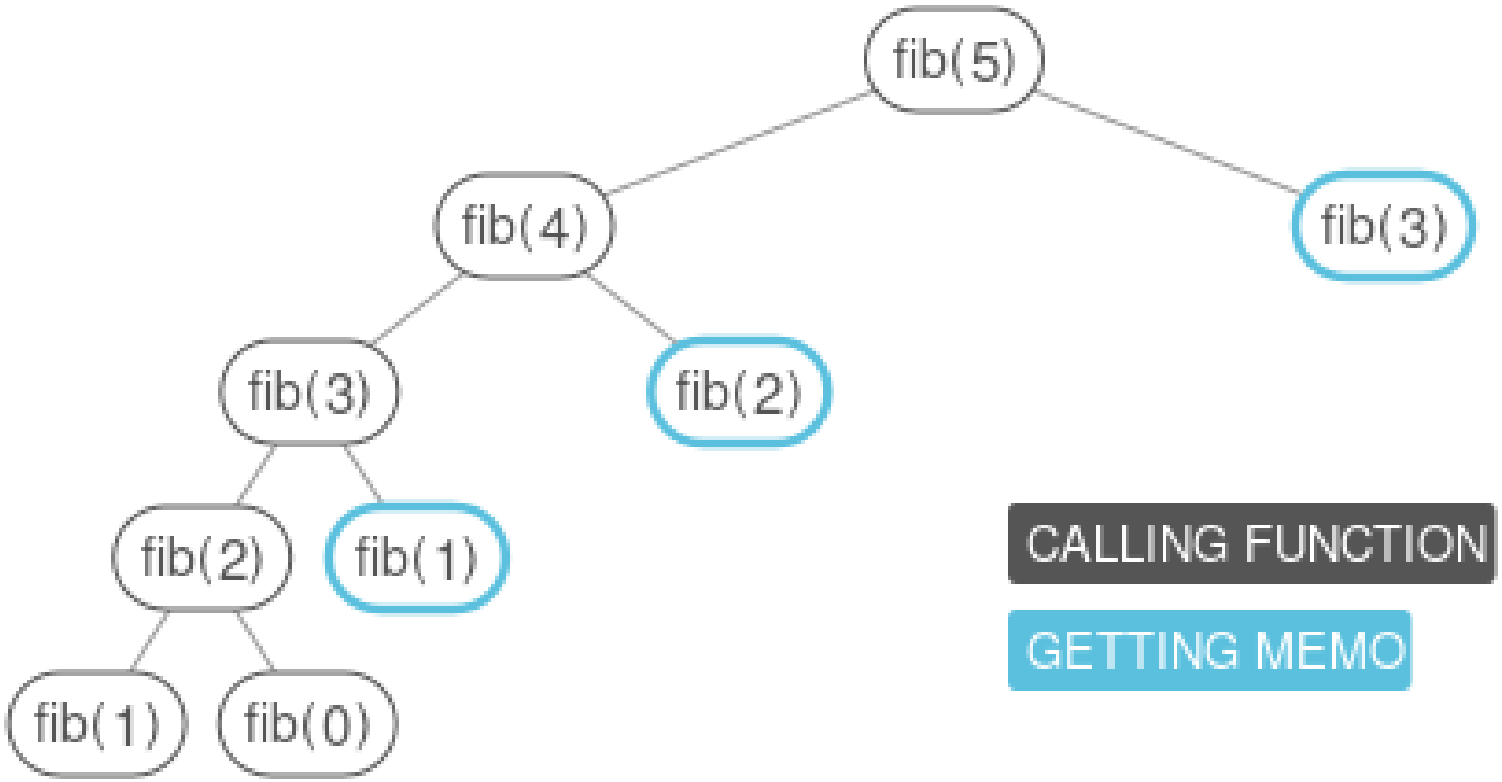
Memoizacion

Resultados repetidos:



Memoizacion

Memoizemos!



¿Qué es el Backtracking?

- El backtracking es un enfoque algorítmico para resolver problemas de decisión, búsqueda y optimización.
- Se utiliza cuando es necesario probar todas las soluciones posibles para encontrar la mejor.
- A menudo se aplica a problemas combinatorios y de optimización donde se busca una solución en un espacio de búsqueda.

Cómo Funciona el Backtracking

Pasos Básicos

1. Selecciona una opción y avanza a la siguiente etapa.
2. Si al avanzar encuentras que tu elección anterior no puede ser completada en una solución válida, retrocede (backtrack) a la elección anterior.
3. Repite este proceso hasta encontrar una solución completa o agotar todas las opciones posibles.

Ejemplo Común: Problema de las N-Reinas

Colocar N reinas en un tablero de ajedrez de manera que ninguna reina ataque a otra.