UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

FACULTAD DE ESTUDIOS SUPERIORES ARAGON

TAREA: INVESTIGACION

P R E S E N T A

Alexis Hernández Zamudio

APROFESOR

Jesús Hernández Cabrera

Gpo:1558

Ciudad Nezahualcóyotl, EDOMEX. 5 DE noviembre del 2025

Serie de Fibonacci

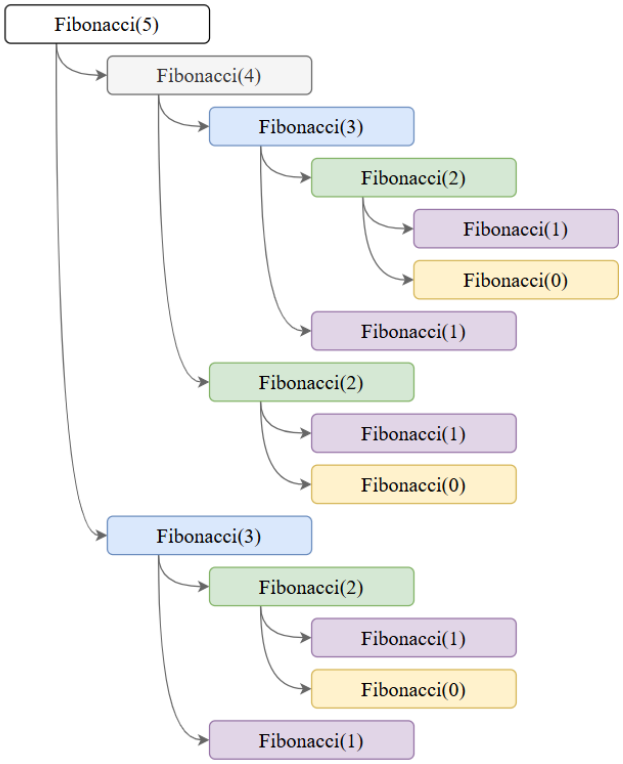
La serie de Fibonacci puede definirse de la siguiente manera:

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

Una posible implementación del cálculo de la serie de Fibonacci utilizando recursión es la siguiente:

```
1 func Fibonacci(n int) int {
2     if n <= 1 {
3         return n
4     }
5     return Fibonacci(n-1) + Fibonacci(n-2)
6 }
```

En la línea 5 se observa que la función `Fibonacci` se llama a sí misma dos veces, lo que provoca que se realicen cálculos redundantes. Por ejemplo, al calcular `Fibonacci(5)`, se realizan las siguientes llamadas:



Se puede observar que `Fibonacci(5)` y `Fibonacci(4)` se calcularon 1 vez, `Fibonacci(3)` se calculó 2 veces, `Fibonacci(2)` y `Fibonacci(0)` se calcularon 3 veces cada uno, y `Fibonacci(1)` se calculó 5 veces. Esto significa que el tiempo de ejecución del algoritmo es exponencial, lo que lo hace ineficiente para valores grandes de `n`.

Problema de la mochila (*Knapsack Problem*)

El problema de la mochila es un clásico en programación dinámica y se puede enunciar como:

Dado un conjunto de objetos, cada uno con un peso y un valor, el objetivo es determinar que objetos se deben incluir en una mochila de capacidad limitada para maximizar el valor total.

Supongamos que tenemos una mochila de capacidad 5 y los siguientes objetos:

Objeto	Peso	Valor
1	3	2
2	4	3
3	1	4
4	2	2
5	5	5

La solución se puede encontrar utilizando nuevamente tabulación, donde se construye una tabla que almacena el valor máximo que se puede obtener para cada capacidad de la mochila y cada objeto.

A continuación, se presenta la ecuación de recurrencia que se utiliza para llenar la tabla:

$$V(i,w) = \begin{cases} 0 & \text{si } i = 0 \text{ o } w = 0 \\ V(i-1,w) & \text{si } p_i > w \\ \max(V(i-1,w), v_i + V(i-1,w-p_i)) & \text{si } p_i \leq w \end{cases}$$

Donde:

$V(i,w)$ es el valor máximo que se puede obtener con los primeros objetos y una capacidad de mochila .

p_i es el peso del objeto .

v_i es el valor del objeto .

w es la capacidad de la mochila.

i es el índice del objeto actual.

$V(i-1, w)$ es el valor máximo, para una mochila de capacidad , sin incluir el objeto .

$V(i-1, w-p_i)$ es el valor máximo al incluir el objeto y restar su peso de la capacidad de la mochila.

La celda $V(i, w)$ representa el valor máximo que se puede obtener con los primeros objetos y una capacidad de mochila .

La complejidad temporal y espacial del algoritmo es:

Complejidad temporal

$O(n \times w)$, donde es el número de objetos y es la capacidad de la mochila. Esto se debe a que se recorre la tabla de tamaño por .

Complejidad espacial

$O(n \times w)$, ya que se utiliza una tabla de tamaño por para almacenar los resultados de los subproblemas.

Tabulación (*Bottom/Up*) vs. Memorización (*Top/Down*)

La **tabulación** y la **memorización** son dos enfoques diferentes para implementar la programación dinámica, y cada uno tiene sus propias ventajas y desventajas.

Como vimos la **tabulación**, es un enfoque más iterativo que utiliza una tabla para almacenar los resultados de todos los subproblemas desde el principio. Esto significa que se resuelven todos los subproblemas de manera sistemática y se almacenan en la tabla.

A diferencia de la **tabulación**, la **memorización** utiliza recursión para resolver el problema y almacena los resultados de los subproblemas en alguna estructura de datos conveniente, por ejemplo un diccionario o una tabla, a medida que se van calculando. Esto significa que se resuelven los subproblemas solo cuando son necesarios, lo que puede ser más eficiente en algunos casos.

La **memorización** es útil cuando el problema tiene una estructura de árbol, donde algunos subproblemas se resuelven varias veces. Por ejemplo en el caso de la serie de Fibonacci, donde se realizan múltiples llamadas recursivas a los mismos subproblemas.

En el siguiente fragmento de código se muestra la implementación de la serie de Fibonacci utilizando memorización:

```go
func Fibonacci(n int, memo map[int]int) int {
    if n <= 1 {
        return n
    }
    if val, ok := memo[n]; ok {
        return val
    }
    memo[n] = Fibonacci(n-1, memo) + Fibonacci(n-2, memo)
    return memo[n]
}
```

En este caso, se utiliza un mapa `memo` para almacenar los resultados de los subproblemas a medida que se van calculando. Si el resultado ya está en el mapa, se devuelve directamente sin volver a calcularlo.

La complejidad temporal y espacial del algoritmo es:

Complejidad temporal

$O(n)$, ya que cada subproblema se resuelve una sola vez y se almacena en el mapa.

Complejidad espacial

$O(n)$, ya que se utiliza un mapa de tamaño para almacenar los resultados de los subproblemas.

EXPLICACION

¿QUÉ ES?
Elegir qué objetos meter en una mochila con espacio limitado para llevar el máximo valor posible.
EL PROBLEMA
Mochila con peso máximo (ej: 5 kg)
Objetos con diferente peso y valor
No cabe todo - hay que elegir

LA SOLUCIÓN
No es llevar lo más valioso, sino la mejor combinación que quepa.

CÓMO FUNCIONA
Probar combinaciones inteligentemente
Empezar con objetos simples
Recordar las mejores opciones
Usar eso para decidir con nuevos objetos

EJEMPLO PRÁCTICO
Mochila de 4 kg:
Objeto A: 3 kg, valor 4
Objeto B: 2 kg, valor 3
Objeto C: 1 kg, valor 1
Mejor solución: B + C = 3 kg, valor 4

APLICACIONES
Empacar maletas
Hacer compras con presupuesto
Elegir inversiones
Planificar tiempo

CONCLUSIÓN
Es encontrar el máximo valor dentro de tus límites.