octubre 2025

# PROGRAMACIÓN DINÁMICA

Joaquín - Steven - Elías - Alexander - Paula

# MOTIVACIÓN

Programación Dinámica *es como la experiencia*:

aprendes de los pequeños pasos que ya diste para **tomar la mejor decisión final, sin repetir errores.**

# PROGRAMACIÓN DINÁMICA

## ¿Qué es?

La programación dinámica es un enfoque de resolución de problemas que involucra dividir un problema en subproblemas más pequeños y resolverlos de manera independiente.

## Divide y Vencerás

La programación dinámica busca encontrar la solución óptima global combinando las soluciones óptimas de los subproblemas.

## La Estrategia

Se caracteriza por almacenar y reutilizar los resultados de los subproblemas para evitar re-calcularlos, lo que mejora la eficiencia del algoritmo.

# ¿CUÁNDO USARLA?

Especialmente útil para **problemas con superposición de subproblemas**, donde los mismos subproblemas se resuelven repetidamente.

# DP
# → EJERCICIO 1

# PROBLEMA: CALCULAR EL N-ÉSIMO NÚMERO DE FIBONACCI.

- La subestructura óptima en este problema radica en que el n-ésimo número de Fibonacci se puede calcular utilizando los resultados de los dos números de Fibonacci anteriores (n-1 y n-2).

- Estos a su vez se calculan utilizando los resultados de los números de Fibonacci aún más anteriores (n = 3, n = 2, n = 2 y n = 1).

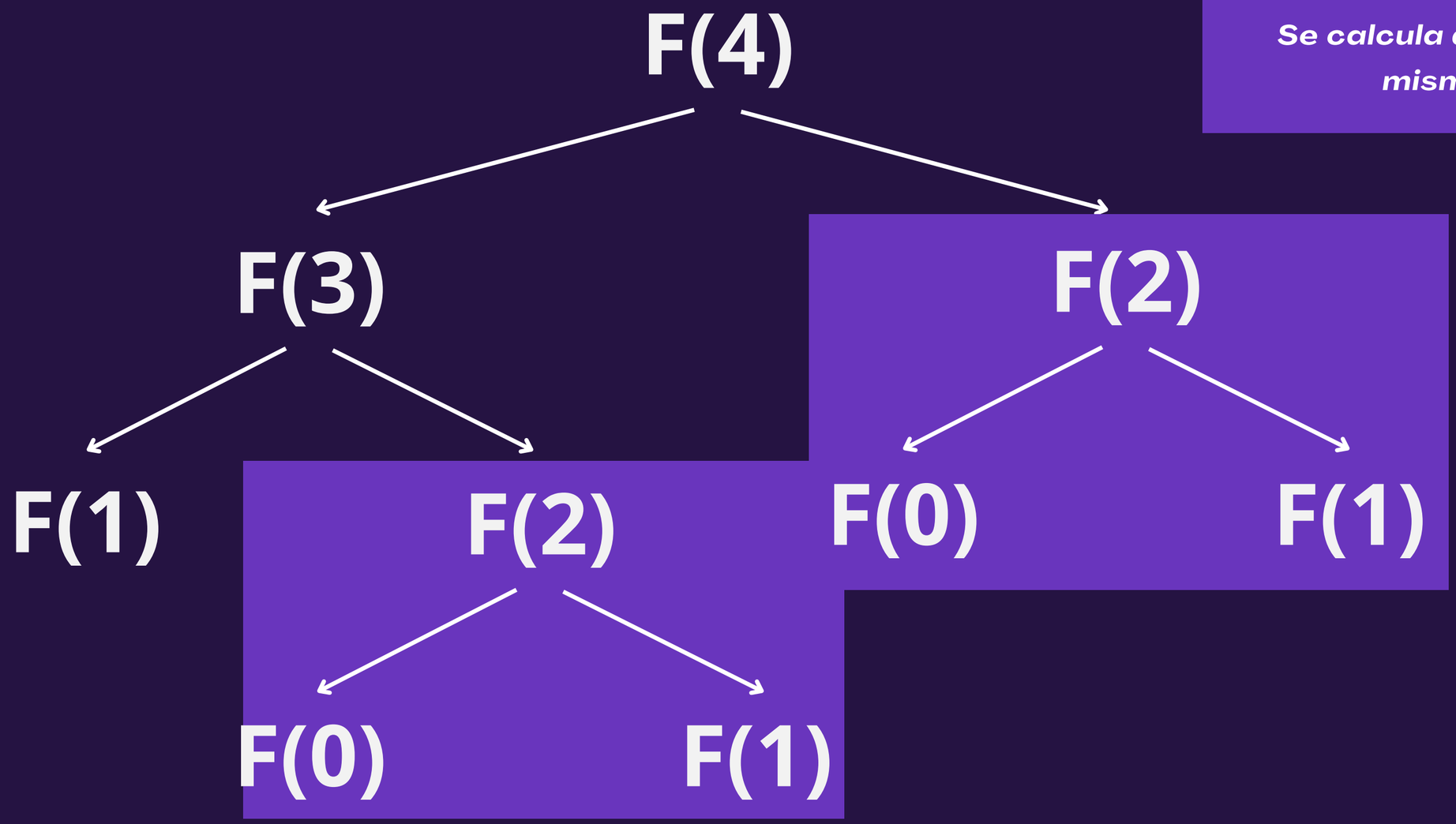
- La solución óptima para calcular el quinto número de Fibonacci es combinar las soluciones óptimas para calcular el cuarto y tercer número de Fibonacci.

# PD-RECURSIVO:

```
int fib(int n)
{
    if (n <= 1)
        return n;
    return fib(n - 1) + fib(n - 2);
}
```



# PD-RECURSIVO: SU ÁRBOL ASOCIADO




***Se calcula dos veces lo mismo!!!!!!***



# PD-RECURSIVO CON MEMORIA:

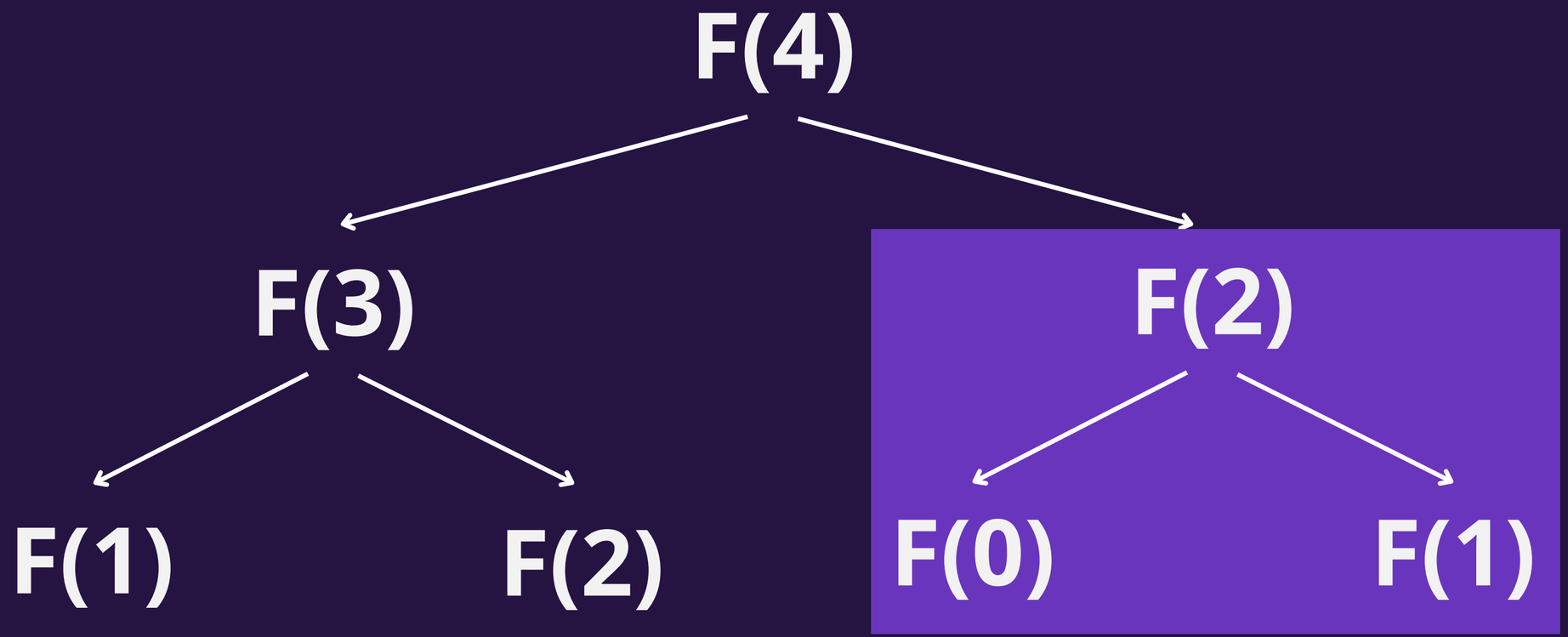
```
int fib(int n, int* mem)
{
    if (n <= 1)
        return n;

    if (mem[n] != 0)
        return mem[n];

    mem[n] = fib(n - 1, mem) + fib(n - 2, mem);
    return mem[n];
}
```



# PD-RECURSIVO CON MEMORIA: SU ÁRBOL




***Se computa solo una vez f(2), f(1) y f(0)***

***mem***

| f(0) | f(1) | f(2) | f(3) | f(4) |
|---|---|---|---|---|
| 0 | 1 | 1 | 2 | 3 |

# PD-ITERATIVO

```
int fib(int n)
{
    int f[n + 2];
    f[0] = 0;
    f[1] = 1;
    for (int i = 2; i <= n; i++)
        f[i] = f[i - 1] + f[i - 2];
    return f[n];
}
```

# DP
# → EJERCICIO 2

# FIESTA MINION

Los Minions han decidido organizar una gran serie de fiestas por todo el vecindario de Gru. Cada casa tiene un potencial diferente para recaudar bananas (la moneda oficial minion), pero también son un poco caóticos...

Hay N casas, y en cada casa i se puede obtener una ganancia de bi bananas ($0 \leq i < N$). Sin embargo, los Minions tienen una regla muy clara:

- **Si organizan una fiesta en la casa i, las dos casas anteriores (i - 1 y i - 2) y las dos siguientes (i + 1 y i + 2) no podrán hacer fiesta**, ya que los vecinos se quejan del ruido, las risas y las explosiones de confeti amarillo.**

Tu misión es ayudar a los Minions a planificar las fiestas para maximizar la cantidad total de bananas recaudadas, dado el arreglo B que indica cuántas bananas se ganarían en cada casa si se realiza una fiesta allí.



# FIESTA MINION

1. Plantea el pseudocódigo de programación dinámica recursivo que resuelve el problema.
2. Plantea el pseudocódigo de programación dinámica iterativo que resuelve el problema.



# FIESTA MINION

1. Plantea el pseudocódigo de programación dinámica recursivo que resuelve el problema.

```
int candace_fiesta(int i, int* g, int* p){
    if (i < 0){
        return 0;
    }

    if (p[i] == 0){
        p[i] = max(g[i] + candace_fiesta(i - 3, g, p), candace_fiesta(i - 1, g, p));
    }

    return p[i];
}
```

g = lista de ganancias
p = lista de max ganancia calculada



# FIESTA MINION

2. Plantea el pseudocódigo de programación dinámica iterativo que resuelve el problema.

```
int candace_fiesta_iterativo(int i_ultima_casa, int* g, int* p){
    p[0] = g[0];

    for (int i = 1; i <= i_ultima_casa; i++){
        if (i < 3){
            p[i] = max(g[i], p[i - 1]);
        }
        else{
            p[i] = max(g[i] + p[i - 3], p[i - 1]);
        }
    }

    return p[i_ultima_casa];
}
```

g = lista de ganancias

p = lista de max ganancia calculada

Ejemplo:

p= [p0, p1, p2, p3]

El resultado se encuentra en p[3]





# Veamos un ejemplo

# FIESTA MINION

La ganancia máxima posible es 5

Eligiendo los iglús 1 y 4

¿Por qué?

Con 6 pueblos tal que g = [1, 2, 4, 0, 3, 0]

| i | g[i] | p |
|---|---|---|
| 0 | 1 | [1, 0, 0, 0, 0, 0] |
| 1 | 2 | [1, 2, 0, 0, 0, 0] |
| 2 | 4 | [1, 2, 4, 0, 0, 0] |
| 3 | 0 | [1, 2, 4, 4, 0, 0] |
| 4 | 3 | [1, 2, 4, 4, 5, 0] |
| 5 | 0 | [1, 2, 4, 4, 5, 5] |

```
int candace_fiesta_iterativo(int i_ultima_casa, int* g, int* p){
    p[0] = g[0];

    for (int i = 1; i <= i_ultima_casa; i++){
        if (i < 3){
            p[i] = max(g[i], p[i - 1]);
        }
        else{
            p[i] = max(g[i] + p[i - 3], p[i - 1]);
        }
    }

    return p[i_ultima_casa];
}
```

# DP
# → EJERCICIO 3

# I2-2024-2: DIV-SUM

Dado un conjunto de enteros positivos $A = \{a_1, \ldots, a_n\}$, el problema DivSum consiste en responder si es posible construir dos conjuntos $B_1$ y $B_2$ disjuntos, no vacíos, tales que $B_1 \cup B_2 = A$, y que tengan la misma suma de sus elementos.

DivSum puede resolverse como un problema de decisión, definiendo:
$p(k, S) = 1$ si y solo si es posible construir un conjunto con elementos de $a_k, \ldots, a_n$ cuya suma sea exactamente S.

Su relación recursiva es:

- $p(k, S) = p(k + 1, S)$ OR $p(k + 1, S - a_k)$

Con OR siendo el operador de disyunción que evalúa primero el lado izquierdo. Si dicho lado es 1, devuelve 1 sin evaluar el lado derecho. Esto significa que es posible que ciertos llamados de $p(k, S)$ no se realicen.

# I2-2024-2: DIV-SUM

(a) Identifique los casos base para p(k, S). Defina el valor adecuado de p(k, S) en tales casos.

(b) Resuelva el problema DivSum para A = {1, 4, 3} mediante la definición de p(k, S), usando un llamado inicial adecuado.
Muestre el árbol recursivo de llamados que se genera, indicando qué elementos se incluyen o descartan en cada paso y mostrando cuándo la recursión llega a un caso base.

(c) Proponga el pseudocódigo de un algoritmo recursivo que, para cualquier conjunto A de enteros positivos, responda el problema DivSum para A.

# I2-2024-2: DIV-SUM

(a) Identifique los casos base para p(k, S). Defina el valor adecuado de p(k, S) en tales casos.

**Solución:**

Como nos interesa que la recursión no persista, debemos considerar:

1. k > n: representa el caso en que no quedan elementos.
2. S < 0: representa una suma negativa.
3. S = 0: representa la condición de éxito (agotamos la suma buscada).

Con esto, los casos se resumen en:

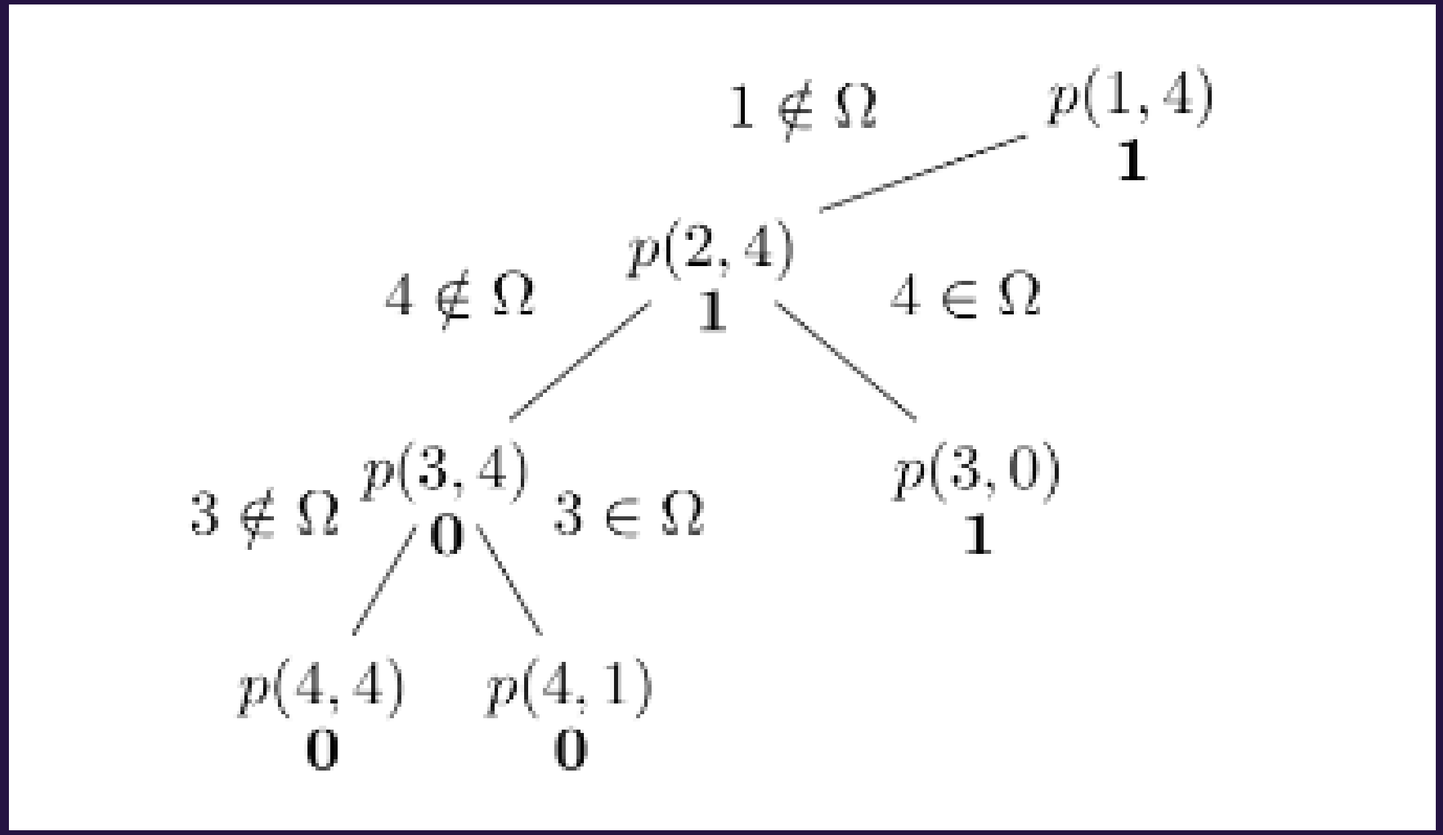
```
p(k, S) = 0   si k > n o S < 0
p(k, S) = 1   si S = 0
```

# I2-2024-2: DIV-SUM

(b) Resuelva el problema DivSum para A = {1, 4, 3} mediante la definición de p(k, S), usando un llamado inicial adecuado.
Muestre el árbol recursivo de llamados que se genera, indicando qué elementos se incluyen o descartan en cada paso y mostrando cuándo la recursión llega a un caso base.

**Solución**:
El llamado que responde al problema DivSum para A = {1, 4, 3} es p(1, 4), donde el segundo argumento es la mitad de la suma de todos los elementos de A. Si p(1, 4) = 1, significa que es posible construir conjuntos con igual suma.
Sea Ω uno de los subconjuntos. A continuación se muestran los llamados recursivos que indican cómo construir Ω:

# I2-2024-2: DIV-SUM



# I2-2024-2: DIV-SUM

(c) Proponga el pseudocódigo de un algoritmo recursivo que, para cualquier conjunto A de enteros positivos, responda el problema DivSum para A.

Se asume $M$ un arreglo bidimensional con celdas vacías. Primero proponemos el algoritmo de programación dinámica que calcula $p(k, S)$.

```
Dynamic(k, S):
1    if k > n ∨ S < 0 :
2        return 0
3    if S = 0 :
4        return 1
5    else:
6        if M[k][S] ≠ ∅ :
7            return M[k][S]
8        else:
9            M[k][S] ← Dynamic(k + 1, S) OR Dynamic(k + 1, S − a_k)
10           return M[k][S]
```

Luego, el algoritmo que resuelve el problema `DivSum` es

```
DivSum(A):
1    S ← SUM(A)
2    if S es impar :
3        return 0
4    return Dynamic(1, S/2)
```

octubre 2025

# PROGRAMACIÓN DINÁMICA

Resumen | Pendiente | Pendiente

Joaquín - Steven - Elías - Alexander - Paula