PROGRAMACIÓN DINÁMICA

Amelia González - Paula Grune - Alexander Infanta – Joaquin Peralta – Elias ayaach

¿Cuál es la idea principal en PD?

"Comerse" (resolver) al problema por partes

Programación Dinámica (DP)

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

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

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

¿Cuándo es útil usar PD?

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

Veámoslo con ejercicios:

PD - **PROBLEMA 1**

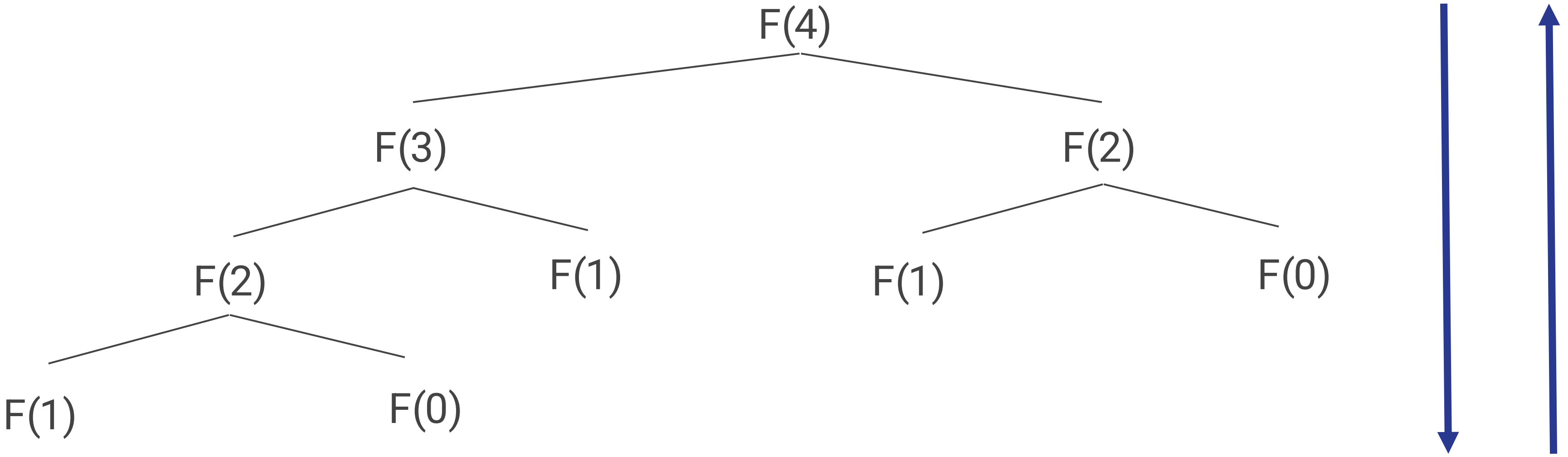
Problema: Calcular el n-ésimo número de Fibonacci.

1. **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).**
2. **Por ejemplo, para calcular el quinto número de Fibonacci (n = 5), necesitamos conocer los resultados de los números de Fibonacci anteriores (n = 4 y n = 3).**
3. **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).**
4. **La solución óptima para calcular el quinto número de Fibonacci es combinar las soluciones óptimas para calcular el cuarto y tercer número de Fibonacci.**

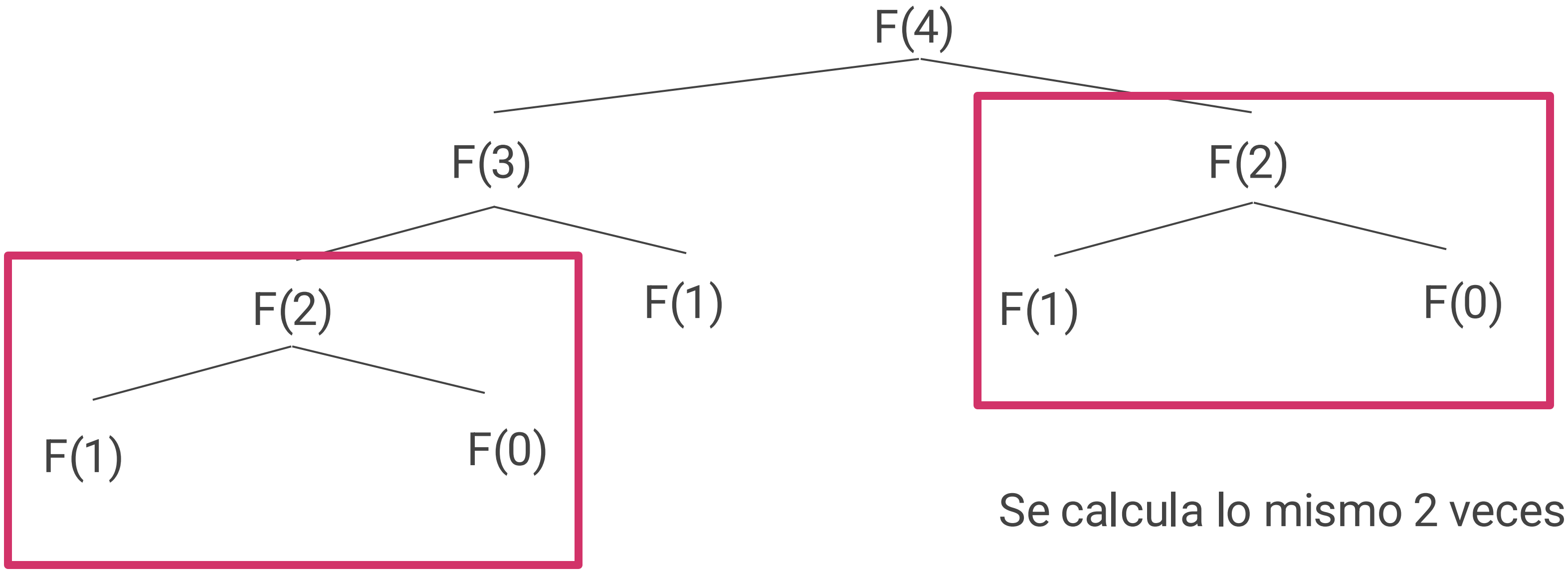
PD Recursivo - **PROBLEMA 1**

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

PD Recursivo - **PROBLEMA 1**



PD Recursivo - **PROBLEMA 1**



Se calcula lo mismo 2 veces!

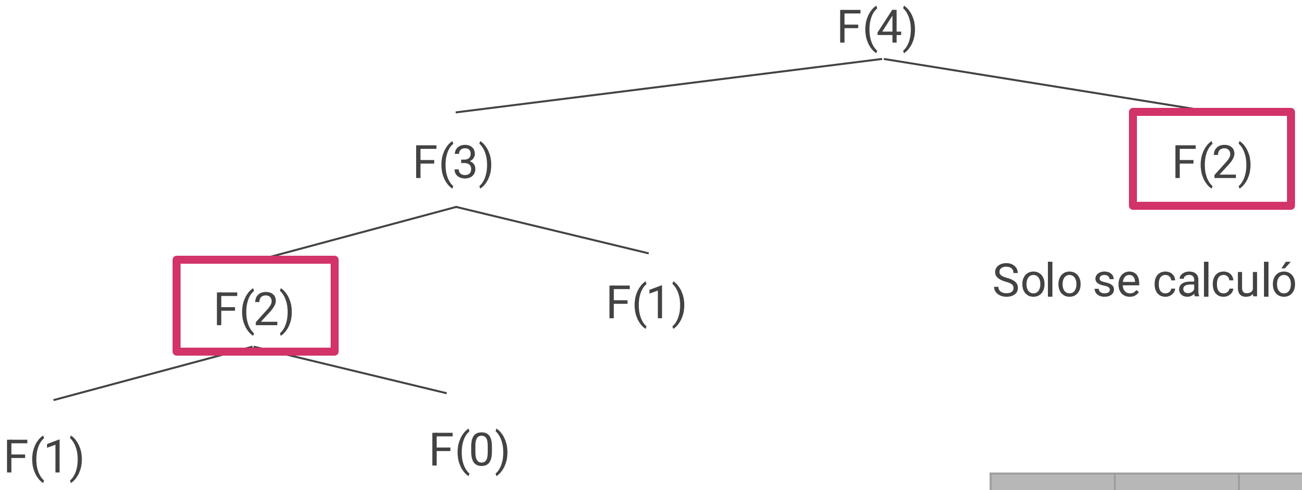
PD Recursivo con memoria - **PROBLEMA 1**

```
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 - **PROBLEMA 1**



Solo se calculó 1 vez!

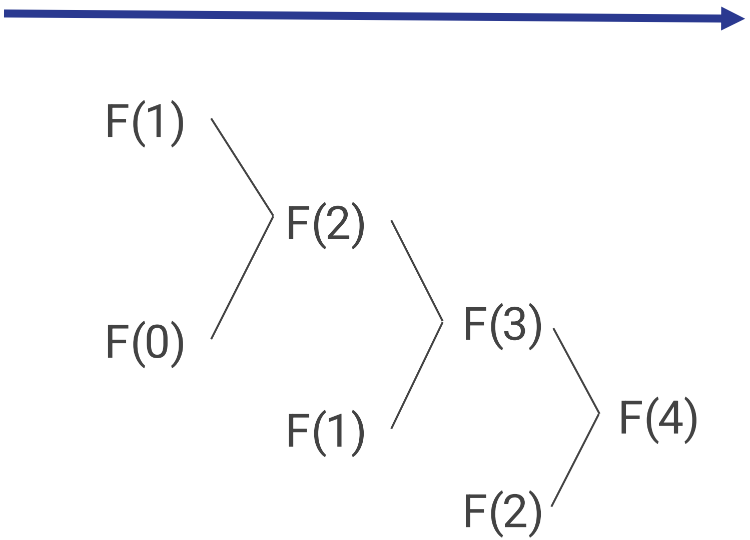
mem =

F(0)	F(1)	F(2)	F(3)	F(4)
0	1	1	2	3

PD Iterativo - **PROBLEMA 1**

```
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];
}
```

PD Iterativo - **PROBLEMA 1**



F =

F(0)	F(1)	F(2)	F(3)	F(4)
0	1	1	2	3

PD - **PROBLEMA 2**

Candace Fiesta

Las Fiestas de Candace son muy lucrativas pero a su vez muy ruidosas, por lo que tu misión será determinar en qué casas hacer una fiesta para ganar lo máximo posible.

Se tendrá N casas, donde en cada casa i se podrá ganar $g_i \in G$ dinero $(0 \leq i < N)$. Cuando se hace una fiesta la casa i, las dos casas anteriores $(i - 2$ e $i - 1)$ y las dos siguientes $(i + 1$ e $i + 2)$ no permitirán una fiesta adicional. Se busca obtener la mayor cantidad de dinero que se podrá recaudar, dado la lista de dinero G que indica cuánto se podrá ganar en cada casa en particular.

PD - **PROBLEMA 2**

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

PD - **PROBLEMA 2**

g = lista de ganancias

p = lista de max ganancia calculada

1. Código recursivo

```
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];
}
```

PD - **PROBLEMA 2**

2. Código iterativo

g = lista de ganancias

p = lista de max ganancia calculada

Ejemplo:

p= [p_0, p_1, p_2, p_3]

El resultado se encuentra en p[3]

```
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];
}
```

PD - **PROBLEMA 2**

Fiesta de Candace

PD - **PROBLEMA 2**

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];
}
```

La ganancia máxima posible es 5
Eligiendo los iglús 1 y 4
¿Por qué?

PD - **PROBLEMA 3**

Considera la ruta Santiago – Puerto Montt. A lo largo de esta ruta, la autoridad vial ha dispuesto *n* lugares en los cuales está permitido poner letreros con propaganda. Estos lugares —x_1, x_2, …, x_n— están especificados por sus distancias desde Santiago en kilómetros, siendo x_1 el más cercano. Por otra parte, tu negocio contrató a una empresa de marketing que calculó que si pones un letrero en el lugar x_i, entonces vas a recibir una ganancia de r_i (que representa las ventas que va a hacer tu negocio gracias a esa propaganda). Hay, sin embargo, una única restricción legal que especifica que un mismo negocio no puede poner letreros 5 kms o menos de separación entre ellos.

Por lo tanto, la pregunta es, ¿En cuáles lugares —un subconjunto de x_1, x_2, …, x_n— te conviene poner los letreros con propaganda de tu negocio, cumpliendo con la restricción anterior, de manera de maximizar el total de las ganancias que recibirás?

Por ejemplo, si $n = 4$, $(x_1, x_2, x_3, x_4) = (6, 7, 12, 14)$ y $(r_1, r_2, r_3, r_4) = (5, 6, 5, 1)$, entonces la solución óptima sería poner los letreros en x_1 y x_3 para una ganancia total de 10.
Plantea un algoritmo de programación dinámica para resolver este problema.

PD - **PROBLEMA 3**

1. Considera las dos alternativas de que el lugar x_n esté o no en la solución óptima; ¿Cuál es el problema que queda por resolver en cada caso?

2. ¿Cómo se generaliza este razonamiento si consideramos el problema definido sólo por los primeros *k* lugares: $x_1, x_2, \ldots, x_k$?

3. Si *opt(k)* representa la ganancia de un subconjunto óptimo de lugares entre $x_1, x_2, \ldots, x_k$, ¿cuál es la recurrencia correspondiente? Es decir, si **opt(k) = max(..., ...)**, ¿qué va en los …?

4. Así, finalmente, a partir de su respuesta anterior, plantea el algoritmo pedido.

PD - **PROBLEMA 3**

a) Considera las dos alternativas de que el lugar x_n esté o no en la solución óptima; ¿cuál es el problema que queda por resolver en cada caso?

Llamemos O a la solución óptima. Para cada lugar x_k consideremos el lugar x_j, $j < k$ (es decir, x_j está más cerca de Santiago que x_k), tal que x_j es el lugar más cercano a x_k que está a una distancia > 5 km de x_k; llamemos $b(k)$ a este lugar.

Así, si x_n está en O, entonces el lugar anterior a x_n que también podría estar en O es $b(n)$; es decir, O sería x_n más (los lugares correspondientes a) **la solución óptima al problema definido por los lugares $x_1, \ldots, b(n)$**.

En cambio, si x_n no está en O, entonces O es igual a **la solución óptima para los lugares $x_1, \ldots, x_{n-1}$**.

PD - **PROBLEMA 3**

b) ¿Cómo se generaliza este razonamiento si consideramos el problema definido sólo por los primeros k lugares: x_1, x_2, …, x_k ?

Sea O_k la solución óptima para los lugares x_1, …, x_k. (Es decir, en a) buscamos O_n.). Generalizando el razonamiento de a), si O_k incluye al lugar x_k, entonces es igual a x_k más $O_{b(k)}$; y si O_k no incluye al lugar x_k, entonces es igual a $O_{k\text{-}1}$.

c) Si $opt(k)$ representa la ganancia de un subconjunto óptimo de lugares entre x_1, x_2, …, x_k , ¿cuál es la recurrencia correspondiente? es decir, si $opt(k) = \max\{ \dots , \dots \}$, ¿qué va en los …?

$$opt(k) = \max\{\, opt(k\text{-}1),\ r_k + opt(b(k))\, \}$$

PD - **PROBLEMA 3**

d) Así, finalmente, a partir de la respuesta a c), plantea el algoritmo pedido.

Esta es la versión recursiva más directa del algoritmo, a partir de la recurrencia anterior:

```
opt(j):
        if j = 0:
                return 0
        else:
                return max{ vj + opt(b(j)) , opt(j-1) }
```

Por supuesto, también son válidas la versión iterativa, y la variante de la versión recursiva en que los valores $opt(k)$ se van almacenando en una tabla a medida que se van calculando y se sacan de la tabla cada vez que vuelven a aparecer en la recursión.

TIPS - PD

PD: Comúnmente, plantear ecuación de recurrencia / algoritmo

- Dado problema, plantear la ecuación de recurrencia
- Dada la ecuación de recurrencia plantear el algoritmo que resuelve el problema

Recomendaciones de ejercicios para **gente que resuelve**

- PD
    - EX-2020-1: Describir ecuación de recurrencia y algoritmo
    - C6-2019-1: Definir recurrencia y diagramar árbol de ejecución
    - Ex-2015-2: Aplicar PD para generar algoritmo en problema mochila