Ayudantía 9

Greedy - Programación Dinámica

José Mendoza - Paula Grune - Alexander Infanta - Joaquín Viñuela

Greedy (algoritmos codiciosos)

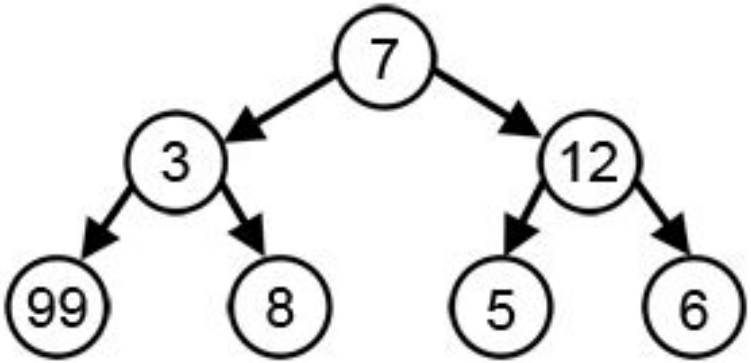
Greedy

Diferente a Backtracking

Varias soluciones pueden ser correctas dependiendo del enfoque

Usualmente usado en problemas de optimización

por ejemplo, si queremos encontrar el camino más costoso:



Greedy

- Se basa en ***tomar decisiones locales óptimas*** en cada paso con la esperanza de alcanzar una solución global óptima.

- Comienza con una solución vacía y agrega gradualmente elementos o toma decisiones que maximicen una métrica o función objetivo en cada paso.

- ***No considera ni revisa las decisiones tomadas anteriormente***, solo se enfoca en la decisión actual basada en la información disponible en ese momento.

- ***No garantiza una solución óptima global y puede quedar atrapado en mínimos locales*** o soluciones parcialmente satisfactorias. (Requiere de un problema con subestructura óptima)

Veámoslo con ejercicios:

Greedy - **PROBLEMA 1**

Estamos a final de semestre y **quieres alcanzar a realizar la mayor cantidad de actividades** antes de que el semestre cierre.

Supongamos que tienes un conjunto de actividades ***A***, cada una con un tiempo de inicio (ti) y un tiempo de finalización (tf).

El objetivo es seleccionar el conjunto máximo de actividades compatibles de manera que ninguna actividad se solape entre sí.

Greedy - **PROBLEMA 1**

- ¿Por qué este problema es greedy?
- ¿Cómo lo soluciono de forma codiciosa?

COMENTEMOS

Greedy - **PROBLEMA 1**

- ¿Por qué este problema es greedy?
  Podemos aplicar una estrategia que busque “la mejor solución disponible en el momento” con la esperanza de que el resultado sea el óptimo/factible.

- ¿Cómo lo soluciono de forma codiciosa?
  Podemos aplicar la estrategia* de iniciar la actividad que termine primero**. Será esta aplicación de una estrategia de selección, que promete una solución óptima local en cada paso, con la esperanza de encontrar una solución óptima global lo que agrega la característica codiciosa a la forma de abordar la resolución.

  (*) Una estrategia de selección -> es la “apuesta” que hacemos
  (**) Esto como completa decisión de nosotros.

Para resolver bajo **estrategia codiciosa** y ser bacanes **necesito**

No olvidar :monki-flip:

Para resolver bajo **estrategia codiciosa** y ser bacanes **necesito**

una **estrategia** ...

...tal que buscar "la mejor solución disponible en el momento" sea un camino prometedor para encontrar una solución óptima global

No olvidar :monki-flip:

Continuando con el problema...

Greedy - **PROBLEMA 1**

Objetivo: Maximizar la cantidad de actividades realizadas

Estrategia: Empezar la actividad que termina primero

Actividad 1: [Inicio: 1, Fin: 4]

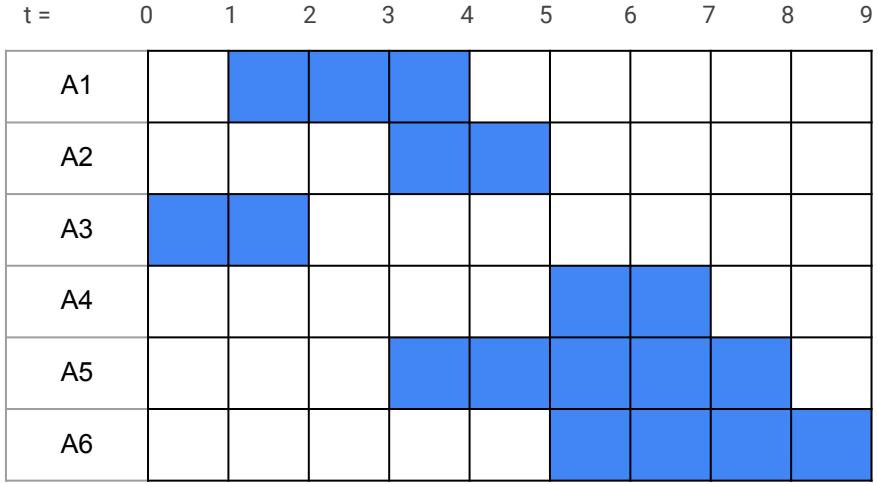
Actividad 2: [Inicio: 3, Fin: 5]

Actividad 3: [Inicio: 0, Fin: 2]

Actividad 4: [Inicio: 5, Fin: 7]

Actividad 5: [Inicio: 3, Fin: 8]

Actividad 6: [Inicio: 5, Fin: 9]



Greedy - **PROBLEMA 1**

Actividad 1: [Inicio: 1, Fin: 4]

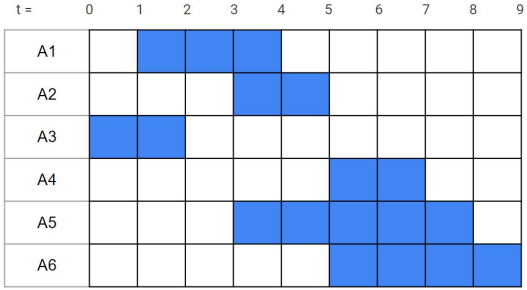
Actividad 2: [Inicio: 3, Fin: 5]

Actividad 3: [Inicio: 0, Fin: 2]

Actividad 4: [Inicio: 5, Fin: 7]

Actividad 5: [Inicio: 3, Fin: 8]

Actividad 6: [Inicio: 5, Fin: 9]



Paso 1: Seleccionar la actividad 3, ya que es la que **finaliza primero**.

Paso 2: Descartar la actividad 1, ya que se solapa con la actividad 3.

Paso 3: Seleccionar la actividad 2, ya que es la que **finaliza primero después de la actividad 3**.

Paso 4: Descartar la actividad 5, ya que se solapa con la actividad 2.

Paso 5: Seleccionar la actividad 4, ya que es la que **finaliza primero después de la actividad 2.**

Paso 6: Descartar la actividad 6, ya que se solapa con la actividad 4.

Greedy - **PROBLEMA 1**

```
ActivitySelection(A):
    Sort activities in non-decreasing order of finish time
    selectedActivities = [A[0]]
    lastSelectedActivity = A[0]

    for i = 1 to length(A) - 1:
        if A[i].start >= lastSelectedActivity.finish:
            selectedActivities.append(A[i])
            lastSelectedActivity = A[i]

    return selectedActivities
```

I2 2023-2 - **PROBLEMA 2**

Para satisfacer la demanda de peluches de Fiu, la mascota de los Juegos Panamericanos y Parapanamericanos, se deben construir tiendas cerca de los recintos deportivos. Consideremos el problema de **minimizar la cantidad de tiendas**, para lo cual, representamos los recintos como puntos **R = {r_1, ..., r_n}** en una misma recta, diciendo que un recinto está cubierto si en la recta hay una tienda a lo más a **L** kilómetros de el. Por ejemplo, si L = 1 y R = {0, 2} la solución óptima es ubicar una tienda entre ambos recintos

I2 2023-2 - **PROBLEMA 2**

A) Se propone la siguiente estrategia para escoger dónde colocar tiendas: ***"ubicar la siguiente tienda en la posición que maximiza el número de recintos nuevos cubiertos y que no habían sido cubiertos por tiendas añadidas antes"***. Demuestre que esta estrategia no es óptima en todos los casos.

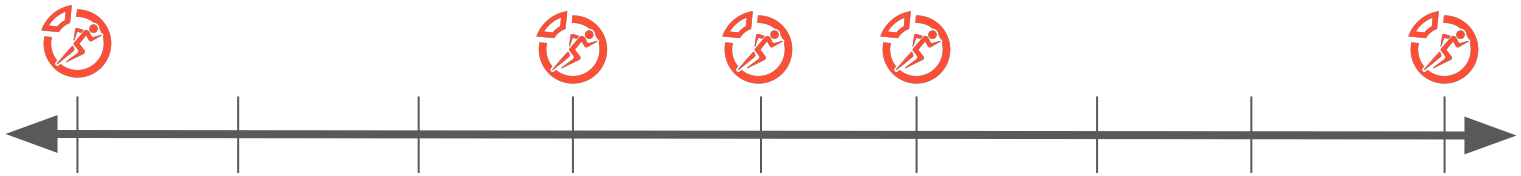
I2 2023-2 - **PROBLEMA 2**

A) Se propone la siguiente estrategia para escoger dónde colocar tiendas: "*ubicar la siguiente tienda en la posición que maximiza el número de recintos nuevos cubiertos y que no habían sido cubiertos por tiendas añadidas antes*". Demuestre que esta estrategia no es óptima en todos los casos.

→ Forzaremos a que la estrategia se equivoque

I2 2023-2 - **PROBLEMA 2**

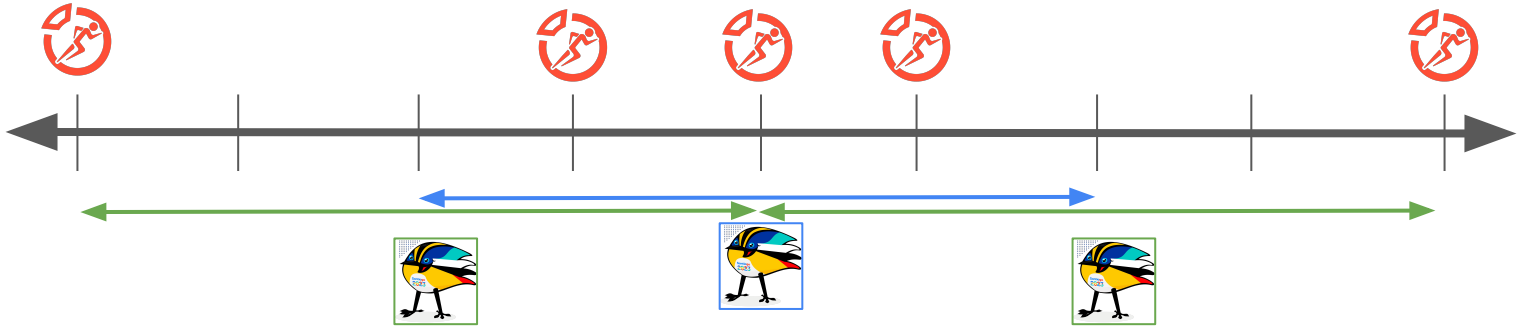
A) Se propone la siguiente estrategia para escoger dónde colocar tiendas: ***"ubicar la siguiente tienda en la posición que maximiza el número de recintos nuevos cubiertos y que no habían sido cubiertos por tiendas añadidas antes***". Demuestre que esta estrategia no es óptima en todos los casos.



→ Elegimos **L = 2** y "sobrecargamos" el centro para que la estrategia ponga tiendas de más

I2 2023-2 - **PROBLEMA 2**

A) Se propone la siguiente estrategia para escoger dónde colocar tiendas: "*ubicar la siguiente tienda en la posición que maximiza el número de recintos nuevos cubiertos y que no habían sido cubiertos por tiendas añadidas antes*". Demuestre que esta estrategia no es óptima en todos los casos.



→ Habrá una tienda extra al centro

I2 2023-2 - **PROBLEMA 2**

B) Considere la siguiente estrategia codiciosa: ***“Con los recintos en orden creciente, si el primer recinto no cubierto es r, entonces ubicar la siguiente tienda a distancia L hacia adelante de r***". Proponga el pseudocódigo de un algoritmo codicioso que use esta estrategia para retornar la lista más corta con las ubicaciones de las tiendas que cubren el conjunto de recintos R con distancia L. Asuma que R es un arreglo no necesariamente ordenado.

I2 2023-2 - **PROBLEMA 2**

```
Greedy(R, L):
1     T ← lista vacía
2     R ← InsertionSort(R)
3     Insertar al final de T el dato R[0] + L
4     i ← 1
5     while i < n :
6         if T.last + L < R[i] :
7             Insertar al final de T el dato R[i] + L
8         i + 1
9     return T
```

n -> largo R

Problema corto

I2 2023-2 - **PROBLEMA 2**

C) Si R contiene *n* recintos y se entrega como arreglo no necesariamente ordenado, determine si existe distinción entre mejor y peor caso para su algoritmo de (b) y determine la complejidad de los casos identificados

I2 2023-2 - **PROBLEMA 2**

C) Si R contiene *n* recintos y se entrega como arreglo no necesariamente ordenado, determine si existe distinción entre mejor y peor caso para su algoritmo de (b) y determine la complejidad de los casos identificados

El algoritmo propuesto solo depende de los mejores/peores casos del algoritmo de ordenación empleado. Para el caso específico de InsertionSort, el mejor caso ocurre cuando R está ordenado y el peor, cuando hay desorden en una cantidad lineal de elementos.

I2 2023-2 - **PROBLEMA 2**

C) Si R contiene *n* recintos y se entrega como arreglo no necesariamente ordenado, determine si existe distinción entre mejor y peor caso para su algoritmo de (b) y determine la complejidad de los casos identificados

El algoritmo propuesto solo depende de los mejores/peores casos del algoritmo de ordenación empleado. Para el caso específico de InsertionSort, el mejor caso ocurre cuando R está ordenado y el peor, cuando hay desorden en una cantidad lineal de elementos.

La complejidad del loop es O(n) siempre. Luego, como InsertionSort es O(n) en el mejor caso, el mejor caso es lineal ($O(n) + O(n) = O(n)$). El peor caso es cuadrático ($O(n^2) + O(n) = O(n^2)$).

PROGRAMACIÓN **DINÁMICA**

¿Cuál es la idea principal en PD?

“Comerse” (resolver) al problema por partes

Programación Dinámica (DP)

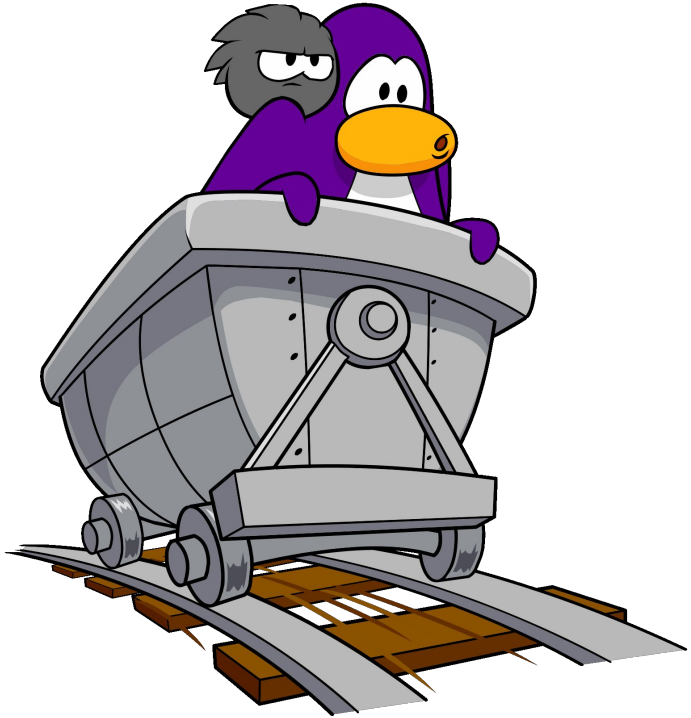
- La programación dinámica es un enfoque de resolución de problemas que involucra ***dividir un problema en subproblemas más pequeños*** y resolverlos de manera independiente.

- 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**

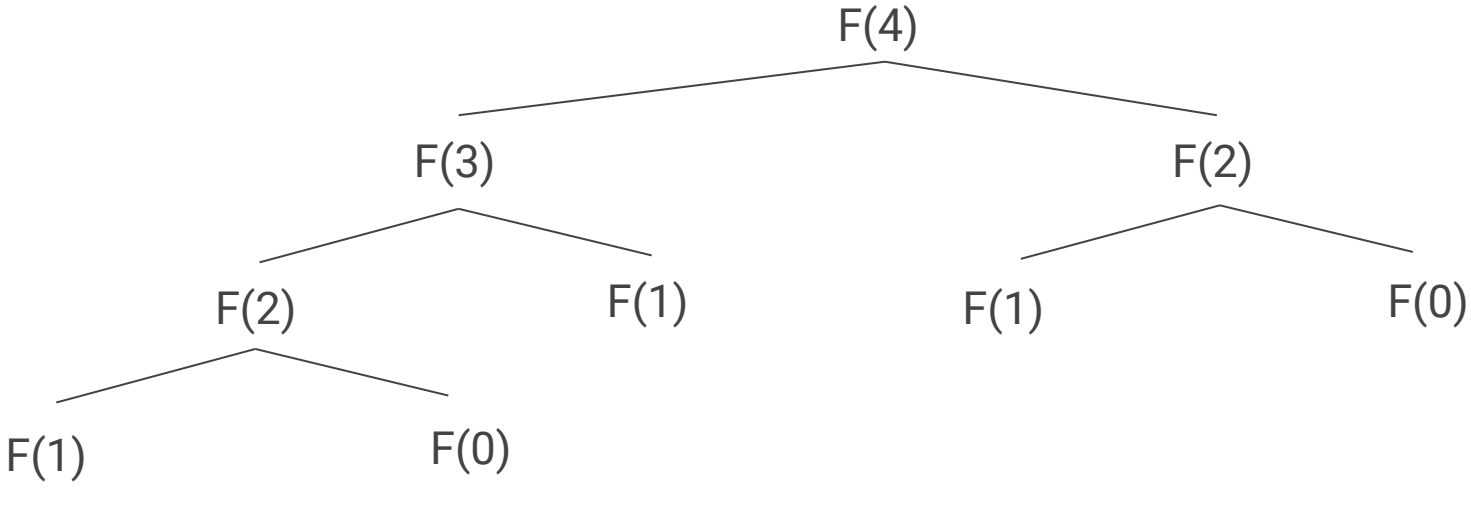
Problema: Calcular el n-ésimo número de Fibonacci.

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.**

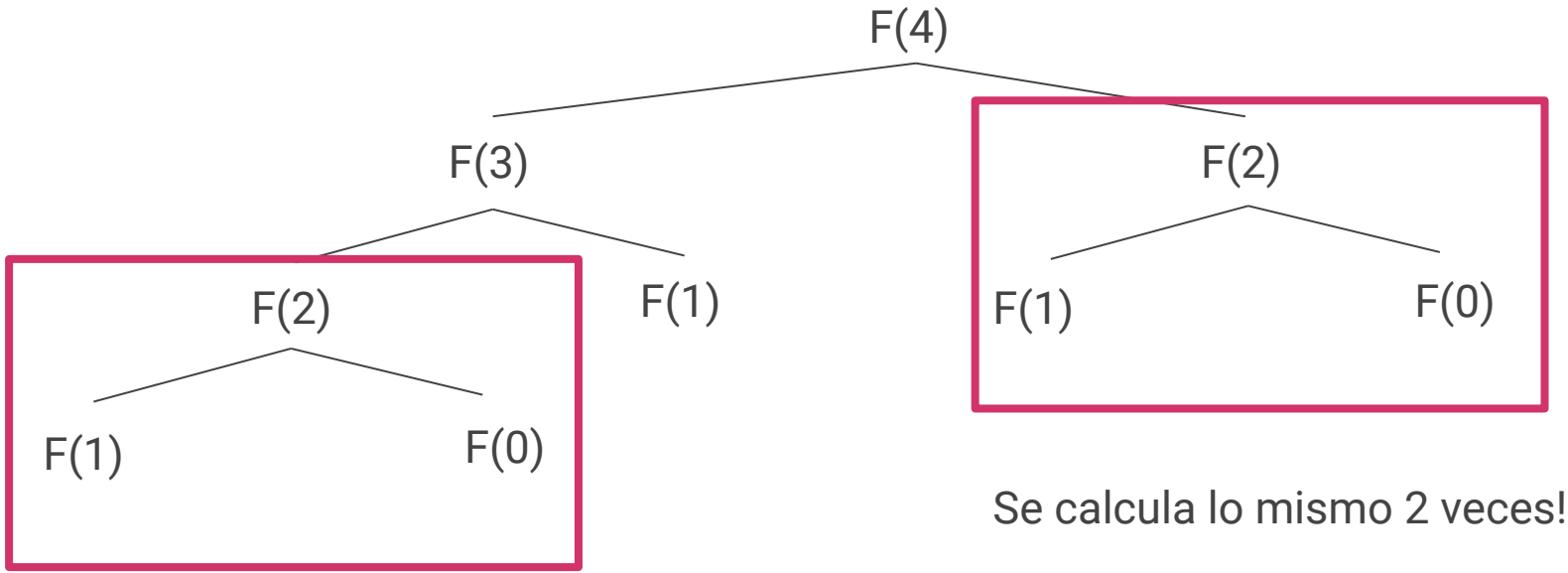
PD Recursivo - **PROBLEMA 1**

```
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!

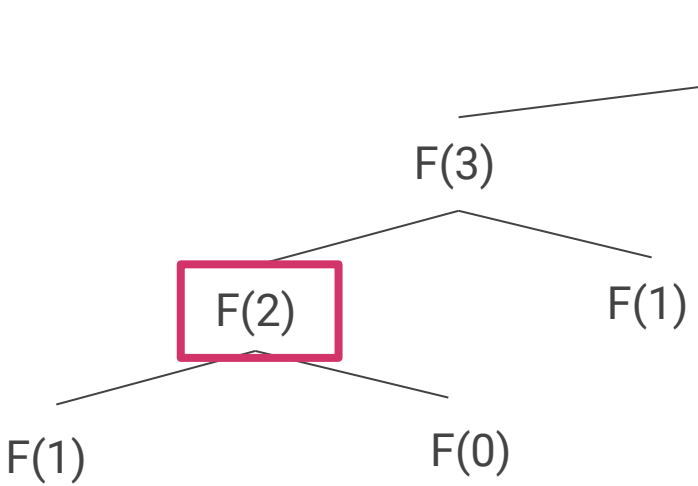
PD Recursivo con memoria - **PROBLEMA 1**

```
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!

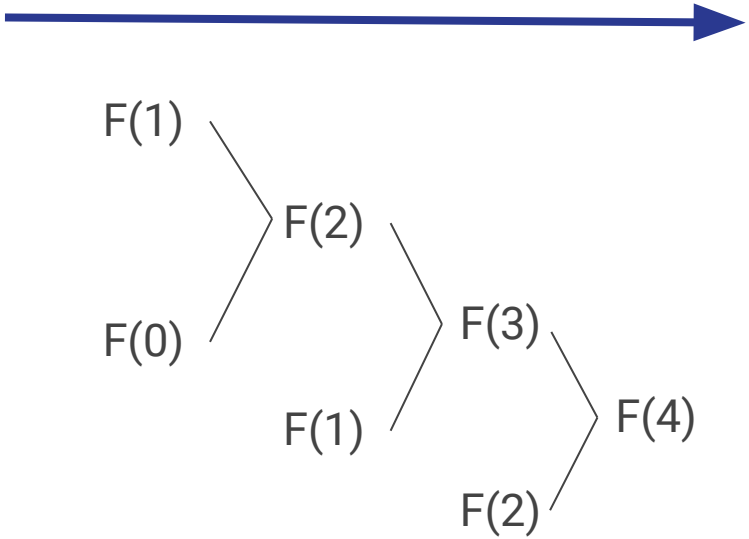
mem =

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**

Fiestas de Puffles

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

Se tendrá N iglús, donde en cada iglú i se podrá ganar gi ∈ G oro (0 ≤ i < N). Cuando se hace una fiesta el iglú i, los dos iglús anteriores (i − 2 e i − 1) y los dos siguientes (i + 1 e i + 2) no permitirán una fiesta adicional. Se busca obtener la mayor cantidad de oro que se podrá recaudar, dado la lista de oro G que indica cuánto se podrá ganar en cada iglú 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 puffle_fiesta(int i, int* g, int* p){
    if (i < 0){
        return 0;
    }

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

    return p[i];
}
```

Ejemplo con p = [p_0, p_1, p_2, p_3]

El resultado se encuentra en p[3]

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 puffle_fiesta_iterativo(int i_ultimo_iglu, int* g, int* p){
    p[0] = g[0];

    for (int i = 1; i <= i_ultimo_iglu; 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_ultimo_iglu];
};
```

PD - **PROBLEMA 2**

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]

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

```
int puffle_fiesta_iterativo(int i_ultimo_iglu, int* g,
    p[0] = g[0];

    for (int i = 1; i <= i_ultimo_iglu; 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_ultimo_iglu];
};
```

TIPS - Greedy , PD

- Greedy: Comúnmente, detectar naturaleza del problema, o proveer contraejemplos
  - Dado problema, demostrar que posee estructura subóptima
  - Dada una estrategia/algoritmo, demostrar que no funciona (Contraejemplo), ó que funciona (Inducción)

- 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**

- Greedy
  - I2-2020-2: Determinar si un grafo es K-coloreable
  - Ex-2018-2: Demostrar correctitud del algoritmo greedy
  - I3-2013-1: Definir subestructura óptima y proponer algoritmo

- 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