

# Programación Dinamica

Ernesto Rodriguez - Juan Roberto Alvaro Saravia

Universidad Francisco Marroquin

*ernestorodriguez@ufm.edu - juanalvarado@ufm.edu*

- Consiste en dividir el problema en instancias más simples y resolverlas recursivamente
- Similar a divide and conquer
- Sin embargo, aplica cuando el mismo sub-problema debe ser resuelto varias veces
- **Idea:** Almacenar soluciones que ya hayan sido calculadas para evitar tener que calcularlas nuevamente.
- Se utiliza a menudo en problemas de optimización

# Ejemplo: Secuencia de Fibonacci

# Ejemplo: Secuencia de Fibonacci

---

## Algorithm 1 Fibonacci

---

```
1: procedure FIBONACCI( $n$ )
2:   if  $n \equiv 0$  then
3:     return 0
4:   if  $n \equiv 1$  then
5:     return 1
6:   return Fibonacci( $n - 1$ ) + Fibonacci( $n - 2$ )
```

---

- ¿Cual es la complejidad respecto a  $n$  de este algoritmo?
- ¿Por que es tan lento?
- ¿Que trabajo estamos repitiendo?
- ¿Podemos optimizar?

# Mejorando la función de Fibonacci

- Consideremos la aplicación recursiva:  
 $\text{Fibonacci}(n - 1) + \text{Fibonacci}(n - 2)$ 
  - Ambos casos deben llamar  $\text{Fibonacci}(n - 3)$ ,  $\text{Fibonacci}(n - 4)$ , ect
  - Cada llamada recursiva crea un arbol de ejecución que repite el trabajo que ya fue hecho
- Estamos repitiendo cantidades excesivas de trabajo, en efecto, tiene un crecimiento exponencial el arbol de ejecución
- **Idea:** Guardemos el trabajo que ya haya sido llevado a cabo, asi evitamos repetir nuestros pasos

# Fibonacci mejorado

---

## Algorithm 2 FibonacciLineal

---

```
1: procedure FIBONACCI LINEAL( $n$ )
2:   let  $cache \leftarrow \text{int}[n + 1]$ 
3:    $cache[0] \leftarrow 0$ 
4:    $cache[1] \leftarrow 1$ 
5:   for let  $i = 2$  upto  $n$  do
6:      $cache[i] \leftarrow cache[i - 1] + cache[i - 2]$ 
7:   return  $cache[n]$ 
```

---

- ¿Cual es la complejidad respecto a  $n$ ?

Por lo general, se utiliza el siguiente proceso:

- Caracterizar la estructura de una solución optima
- Definir recursivamente el valor de cada pedazo de la solución
- Calcular recursivamente cada valor, por lo general de abajo hacia arriba
- Recuperar la solución final de la estructura que fue construida



# Corte de Barras

Problema:

- Una empresa compra barras de acero y los corta en secciones más pequeñas
- Cada corte es gratuito
- La empresa quiere optimizar ganancias: Cortar las barras de acero de tal forma que las barras resultantes se puedan vender al mejor precio
- Las barras se cortan en intervalos enteros

A continuación se muestra la tabla de precios de cada segmento de acero:

length $i$	1	2	3	4	5	6	7	8	9	10
price $p_i$	1	5	8	9	10	17	17	20	24	30

# Caracterización del problema

- ¿Cuántas posibles combinaciones puede cortarse una barra de longitud  $l$ ?
- ¿Es factible explorar todas las combinaciones para encontrar una solución óptima?
- ¿Es necesario considerar todas las posibles combinaciones para encontrar una solución óptima?

# Planteamiento del Problema

- Se utilizarán sumas para denotar cortes:  $l = i_0 + i_1 + \dots + i_i$ , por ejemplo,  $7 = 2 + 2 + 4$  corresponde a una barra de longitud 7 que fue cortado en segmentos de 2, 2 y 4
- El objetivo es buscar la combinación de cortes:  
$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \dots, r_{n-1} + r_1)$$
  - Esto corresponde a hacer un corte inicial  $r_i$  y a eso agregar recursivamente los cortes restantes  $r_n$
  - La idea es optimizar cada uno de los pasos en el proceso de cortado

# Primer intento

---

## Algorithm 3 Cortar

---

```
1: procedure CORTAR( $p, l$ )
2:   if  $n \equiv 0$  then
3:     return 0
4:   let  $q \leftarrow -\infty$ 
5:   for let  $i = 0$  to  $l$  do
6:      $q \leftarrow \max(q, p[i] + \text{Cortar}(p, n - i))$ 
7:   return  $q$ 
```

---

- ¿Cual es la complejidad del algoritmo?

# Segundo intento

- Crear un arreglo
- Cada vez que se encuentra una solución, almacenarla en el arreglo
- De esa manera, se evita tener que calcular multiples veces la misma solución
- Simplemente consiste en modificar el algoritmo que ya existe con un arreglo diseñado para guardar soluciones intermedias

# Segundo intento

---

## Algorithm 4 CortarMemorizado

---

```
1: procedure CORTARMEMORIZADO( $p, l$ )  
2:   let  $res \leftarrow \text{int}[l + 1]$   
3:   for let  $i \leftarrow 0$  upto  $l$  do  
4:      $res[i] \leftarrow -\infty$   
5:   return CortarMemorizadoAux( $p, l, res$ )
```

---



---

**Algorithm 5** CortarMemorizadoAux

---

```
1: procedure CORTARMEMORIZADOAUX( $p, l, res$ )
2:   if  $res[l] \geq 0$  then
3:     return  $res[l]$ 
4:   let  $q \leftarrow -\infty$ 
5:   if  $n \equiv 0$  then
6:      $q \leftarrow 0$ 
7:   else
8:     for  $i \leftarrow 1$  to  $n$  do
9:        $q \leftarrow \max(q, p[i] + \text{CortarMemorizadoAux}(p, n - i, res))$ 
10:   $r[n] \leftarrow q$ 
11:  return  $q$ 
```

---

¿Podemos eliminar la recursion?

# Yes We Can!



# Tercer intento

---

## Algorithm 6 CortarMemorizadoInvertido

---

```
1: procedure CORTARMEMORIZADOINVERTIDO( $p, l$ )
2:   let  $res \leftarrow \text{int}[l + 1]$ 
3:   for let  $i \leftarrow 0$  upto  $l$  do
4:     let  $q \leftarrow -\infty$ 
5:     for let  $j \leftarrow 1$  to  $i$  do
6:        $q \leftarrow \max(q, p[i] + res[j - i])$ 
7:      $r[j] \leftarrow q$ 
8:   return  $r[n]$ 
```

---