Programación Dinamica

Ernesto Rodriguez - Juan Roberto Alvaro Saravia

Universidad Francisco Marroquin

ernestorodriguez@ufm.edu - juanalvarado@ufm.edu

Programación Dinamica

- 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

```
procedure FIBONACCI(n)
    if n ≡ 0 then
        return 0
    if n ≡ 1 then
        return 1
    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:
  `Fibonacci`$(n-1)+$`Fibonacci`$(n-2)$
  - Ambos casos deben llamar `Fibonacci`$(n-3)$, `Fibonacci`$(n-4)$, ect
  - Cada llamada recursiva crea un arbol de ejecición 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

Fibonacci mejorado

Algorithm 2 FibonacciLineal

```
procedure FibonacciLineal(n)
    let cache ← int[n + 1]
    cache[0] ← 0
    cache[1] ← 1
    for let i = 2 upto n do
        cache[i] ← cache[i − 1] + cache[i − 2]
    return cache[n]
```

- ¿Cual es la complejidad respecto a n?

Programación Dinamica

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

- ¿Cuantas possibles combinaciones puede cortarse una barra de longitud l?
- ¿Es factible explorar todas las combinaciones para encontrar una solución optima?
- ¿Es necesario considerar todas las posibles combinaciones para encontrar una solución optima?

Planteamiento del Problema

- Se utilizaran sumas para denotar cortes: $l = i_0 + i_1 + \ldots + i$, 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: $r_n = \mathtt{max}(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1)$
    - 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

Primer intento

Algorithm 3 Cortar

```
procedure CORTAR(p, l)
    if n ≡ 0 then
        return 0
    let q ← −∞
    for let i = 0 to l do
        q ← max(q, p[i] + Cortar(p, n − i))
    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

Segundo intento

Algorithm 4 CortarMemorizado

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

Segundo intento

Algorithm 5 CortarMemorizadoAux

```
procedure CortarMemorizadoAux(p, l, res)
    if res[l] ≥ 0 then
        return res[l]
    let q ← −∞
    if n ≡ 0 then
        q ← 0
    else
        for i ← 1 to n do
            q ← max(q, p[i] + CortarMemorizadoAux(p, n − i, res))
    r[n] ← q
    return q
```

¿Podemos eliminar la recursion?


Yes We Can!
WAR PRODUCTION CO-ORDINATING COMMITTEE

Tercer intento

Tercer intento

Algorithm 6 CortarMemorizadoInvertido

```
procedure CortarMemorizadoInvertido(p, l)
    let res ← int[l + 1]
    for let i ← 0 upto l do
        let q ← −∞
        for let j ← 1 to i do
            q ← max(q, p[i] + res[j − i]
        r[j] ← q
    return r[n]
```

¿Cual es la complejidad de este algoritmo?

Cuarto intento

- Reducción drastica de complejidad
- Requiere un arreglo auxiliar, sin embargo, la implementación original consume memoria mediante la recursión
- **Problema:** Este algoritmo solamente nos retorna las ganancias optimas, no los cortes que se deben realizar. ¿Solución?

Cuarto intento

- Reducción drastica de complejidad
- Requiere un arreglo auxiliar, sin embargo, la implementación original consume memoria mediante la recursión
- **Problema:** Este algoritmo solamente nos retorna las ganancias optimas, no los cortes que se deben realizar. ¿Solución?

Cuarto intento

- Reducción drastica de complejidad
- Requiere un arreglo auxiliar, sin embargo, la implementación original consume memoria mediante la recursión
- **Problema:** Este algoritmo solamente nos retorna las ganancias optimas, no los cortes que se deben realizar. ¿Solución?
  - Guardar las longitudes de todos los cortes realizados, no solo el optimo.

Quinto intento

Quinto intento

Algorithm 7 CortarMemorizadoInvertido

```
procedure CortarMemorizadoInvertido(p, l)
    let res ← int[l + 1]
    let cortes ← int[]
    for let i ← 0 upto l do
        let q ← −∞
        for let j ← 1 to i do
            if p[i] + res[j − i] > q then
                q ← p[i] + res[j − i]
            q ← max(q, p[i] + res[j − i]
            push(i, cortes)
        r[j] ← q
    return r[n]
```

¿Cual es la complejidad de este algoritmo?