Algoritmos y Estructuras de Datos III

Programación Dinámica

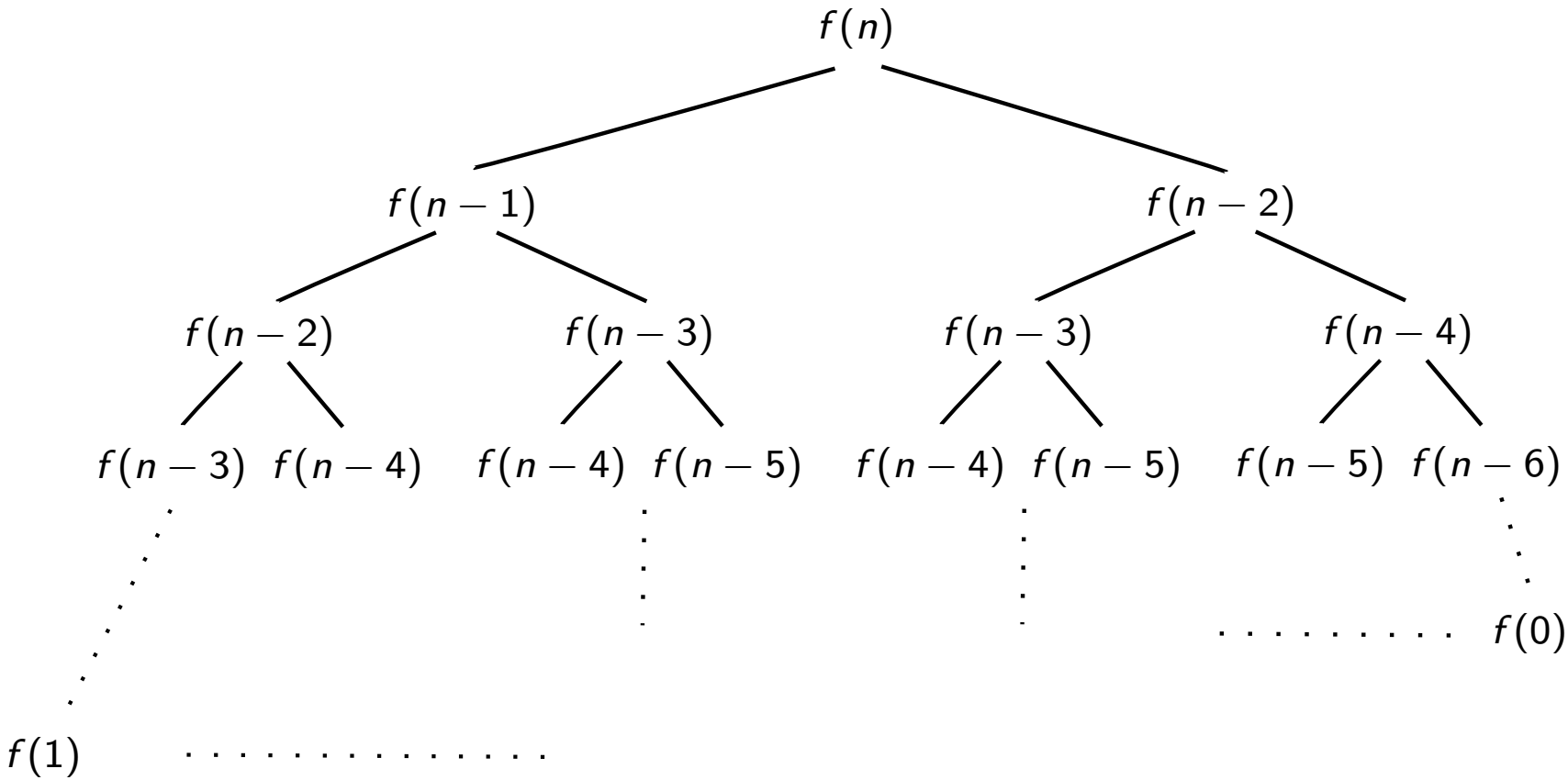
Guido Tagliavini Ponce

24 de agosto de 2016

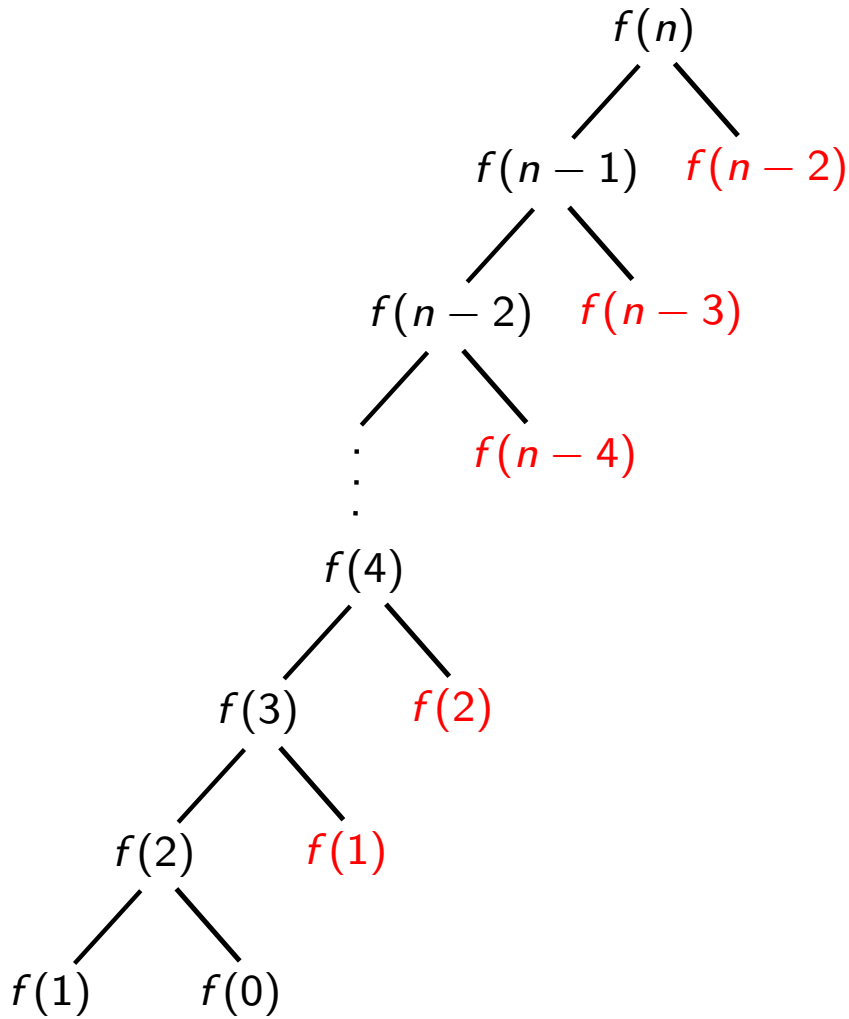
- **Problema:** dar un algoritmo que calcule el n-ésimo número de Fibonacci.
- Usamos la recurrencia que ya conocemos:

$$f(k) = f(k-1) + f(k-2)$$

- Ejecución del algoritmo recursivo naïve:



- Complejidad: $\Omega(2^{n/2})$
- Problema: calculamos muchas veces cada subproblema.
- ¿Y si nos vamos guardando el resultado de cada subproblema que calculamos?



- ¿Es más rápido? $\Theta(n)$

¿Qué es *Programación Dinámica*?

- Técnica que consiste en resolver un problema recursivo guardando y reutilizando la solución de cada subproblema que calculamos.
- Es necesario que sea recursivo, porque necesitamos la noción de subproblema.
- Resulta útil cuando en el árbol de la recurrencia hay muchos subproblemas repetidos. Esto se denomina *solapamiento de subproblemas*.

Problema: Camino en una matriz

- Se tiene una matriz M de $m \times n$ números naturales.
- Queremos recorrer M desde la casilla $(1, 1)$ hasta la (m, n).
- Sólo podemos movernos de a un casillero, hacia abajo o hacia la derecha.
- Queremos minimizar la suma de los valores de los casilleros por los que pasamos.
- Dar un algoritmo que calcule la mínima suma de un camino de $(1, 1)$ a (m, n).

Formulación recursiva

- La condición necesaria para poder usar PD es tener una formulación recursiva.
- Dicho de otra forma, necesitamos una f como la de Fibonacci.
- Pero este problema no parece tener nada de recursivo.
- Propongo:

$$f(i,j) = \text{suma de un camino óptimo desde } (i,j) \text{ hasta } (m,n)$$

- El problema nos pide calcular $f(1,1)$.
- Necesitamos plantear una recurrencia para esta f.

Formulación recursiva

- Observación clave: un camino desde la posición (i, j) está formado por un camino desde $(i+1, j)$ o desde $(i, j+1)$.
- Si el camino desde (i, j) es óptimo, ¿puede no ser óptimo el camino desde la casilla siguiente?
- Afirmo:

$$f(i,j) = M[i,j] + \text{mín}\{f(i+1,j), f(i,j+1)\}$$

- Algunos casos particulares:

$$f(m,j) = M[m,j] + f(m,j+1)$$

$$f(i,n) = M[i,n] + f(i+1,n)$$

$$f(m,n) = M[m,n]$$

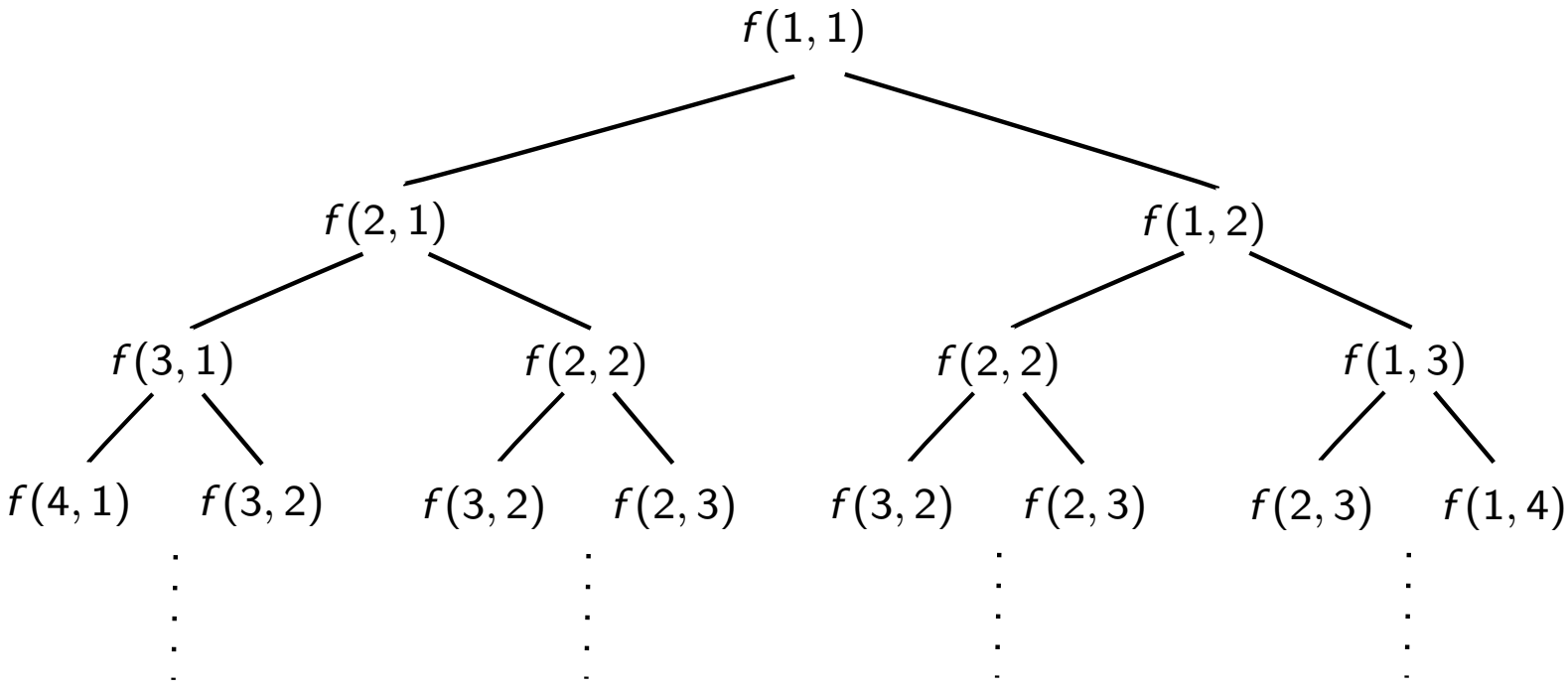
- Esta recurrencia es traída a ustedes gracias al Principio de Optimalidad.

- Ya tenemos la formulación recursiva.
- Nuestro algoritmo de PD ejecutará la recurrencia, guardando cada valor de $f(i, j)$ que compute.
- Usamos un diccionario para guardar esos resultados.

Algoritmo

```
1. Inicializo diccionario dicc, con dicc(i, j) = nil
    para cada i, j
2. func(i, j):
      // quiero que func(i, j) = f(i, j)
      if dicc(i, j) != nil
         return dicc(i, j)
      if i = m and j = n
         dicc(m, n) <- M[m, n]
      else if i = m
         dicc(m, j) <- M[m, j] + func(m, j + 1)
      else if j = n
         dicc(i, n) <- M[i, n] + func(i + 1, n)
      else
         dicc(i, j) <- M[i, j] + min{func(i + 1, j),
                                     func(i, j + 1)}
      return dicc(i, j)
3. Solucion del problema: func(1, 1)
```

- ¿Es realmente útil guardar resultados de subproblemas en este caso?
- Hay solapamiento de subproblemas:



- ¿Cómo representamos el diccionario?

Otro enfoque

- Los algoritmos recursivos tienen cierto *overhead* de espacio y tiempo.
- Queremos usar la idea de no repetir cómputos, pero obtener un algoritmo iterativo.
- Suena confuso, porque venimos hablando de recursión desde el principio de la clase.
- Si bien la formulación es recursiva, el algoritmo que use esa fórmula no necesariamente tiene que serlo.

Otro enfoque

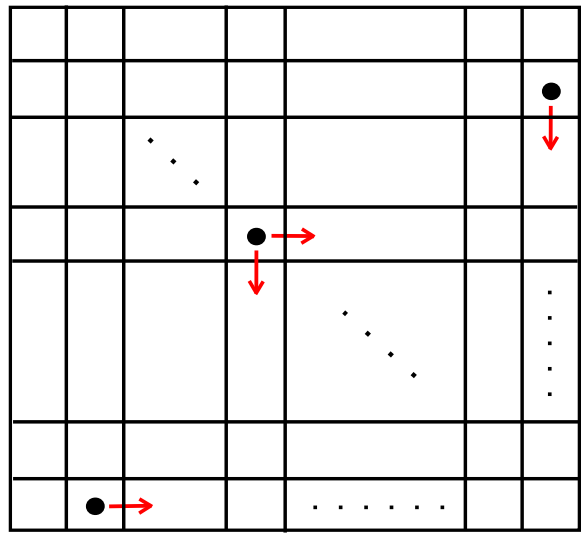
- Al intentar resolver el subproblema (i, j), nuestro algoritmo sabe que depende de dos subproblemas idénticos pero más chicos, específicamente $(i + 1, j)$ y $(i, j + 1)$, por lo que los resuelve recursivamente.
- Este enfoque, en el que recorremos primero los subproblemas grandes y luego los más chicos, se denomina *top-down*.
- La recursión nos da la capacidad de *frenar* el cómputo para (i, j) por un momento, y cuando $(i + 1, j)$ e $(i, j + 1)$ estén resueltos, volver y terminar.
- Si queremos evitar usar recursión, no podemos *frenar* la ejecución. Necesitamos que al intentar calcular (i, j), los subproblemas $(i + 1, j)$ e $(i, j + 1)$ ya estén resueltos.
- Solución: empezar por los subproblemas más chicos, moviéndonos hacia los más grandes. Este otro enfoque recibe el nombre de *bottom-up*.

El problema de las dependencias

- ¿En qué orden tengo que resolver los subproblemas para que al momento de resolver (i, j), ya tenga calculados $(i + 1, j)$ e $(i, j + 1)$?
- En general, la pregunta es: ¿en qué orden tengo que resolver los subproblemas para no violar las *dependencias entre los subproblemas*?

El problema de las dependencias

- Como los subproblemas son pares ordenados de enteros, puedo visualizarlos a éstos y sus dependencias usando una matriz:



- Todo subproblema depende del subproblema de *abajo* y del de la *derecha*.
- Formas de recorrer la matriz de subproblemas respetando las dependencias:
  - Por filas: de derecha a izquierda y de abajo a arriba.
  - Por columnas: de abajo a arriba y de derecha a izquierda.
  - Por diagonales invertidas.

Algoritmo bottom-up

```
sol():
   // devuelve la solucion del problema
   // casos base
   dicc(m, n) <- M[m, n]
   for j <- n - 1, ..., 1
      dicc(m, j) <- M[m, j] + dicc(m, j + 1)
   for i <- m - 1, ..., 1
      dicc(i, n) <- M[i, n] + dicc(i + 1, n)
   // caso general - recorro por filas
   for i <- m - 1, ..., 1
      for j <- n - 1, ..., 1
         dicc(i, j) <- M[i, j] + min{dicc(i + 1, j),
                                     dicc(i, j + 1)}
   return dicc(1, 1)
```

Complejidad

- Para simplificar, asumamos que `dicc` define y accede en $O(1)$.
- **Bottom-up:** $O(mn)$.
- **Top-down:** $C_1 + C_2 + C_3$
  - $C_1 =$ costo de incialización de `dicc`
  - $C_2 =$ costo total de cómputos de valores de `dicc`
  - $C_3 =$ costo total de lecturas de valores de `dicc`

Complejidad

- $C_1 = O(mn)$
- $C_2 = \sum_{s \text{ subproblema}} costo(s)$, donde

  $costo(s)$ = costo de computar la fórmula para s, teniendo los resultados de los subproblemas necesarios

  - Por ejemplo, si $s = (i, j)$ con $i < m$ y $j < n$, $costo(s)$ es el costo de computar $M[i, j] + \text{mín}\{f(i+1, j), f(i, j+1)\}$ asumiendo que ya tenemos $f(-, -)$.
  - $C_2 = \sum_{(i,j)} O(1) = O(mn)$
- $C_3 = O(C_2)$ porque el costo de simplemente devolver una entrada de `dicc` es absorbido por el costo de realizar la llamada recursiva desde el subproblema padre. El costo de la llamada recursiva ya es parte de C_2, pues lo pagamos durante el cómputo de la fórmula del subproblema padre.

Complejidad

- En general, podemos estimar la complejidad temporal de una implementación, tanto top-down como bottom-up, como

$$\sum_{s \text{ subproblema}} costo(s)$$

- Usen esta regla de pulgar, mirando la f, para verificar que van por buen camino.
  - $subproblemas$ = todas las combinaciones válidas de argumentos de la f
  - $costo(s)$ = costo de calcular f teniendo los valores recursivos calculados
- Si todos los subproblemas tienen costo $O(C)$, la fórmula se simplifica a

$$O(\#subproblemas \times C)$$

Resumen

- **Paso 1:** Proponer una formulación recursiva, i. e. una f.
- **Paso 2:** Dar una recurrencia para f. Demostrar.
- **Paso 3:** Estimar si la complejidad cumple con lo requerido.
- **Paso 4:** Algoritmo. En el caso bottom-up, estudiar primero las dependencias entre subproblemas.

¡Preguntas!