Diseño de Algoritmos - Algoritmos II

Nazareno Aguirre
Departamento de Computación
Facultad de Ciencias Exactas, Físico-Químicas y Naturales
Universidad Nacional de Río Cuarto

Clase 5: La Técnica Programación Dinámica

Estrategias de Diseño de Algoritmos

Las estrategias de diseño de algoritmos nos brindan herramientas para atacar el problema de construir soluciones algorítmicas para problemas. Hemos visto ya algunas de estas estrategias, principalmente Fuerza Bruta y Divide & Conquer/Decrease & Conquer.

Veremos ahora una nueva estrategia denominada Programación Dinámica, que puede entenderse como una optimización a soluciones Divide & Conquer/Decrease & Conquer, basada en la inversión del cálculo de las soluciones a subproblemas inducida por las invocaciones recursivas de una solución D&C.

Programación Dinámica

Dado un problema P, para el cual:

- ya se cuenta con una solución Divide & Conquer/Decrease & Conquer (top-down, en el sentido que para resolver el problema lo divide en subproblemas más pequeños, y combina sus soluciones para resolver el problema original), PERO
- para cada problema particular,
  - los subproblemas a resolver NO son independientes, sino que comparten a su vez subproblemas (es decir, se "solapan"), y por lo tanto requieren recomputar varias veces las soluciones a algunos subproblemas comunes.
  - el número total de subproblemas a resolver está acotado por una cantidad polinomial, respecto al tamaño del problema que los origina.

Resolver el problema mediante Programación Dinámica consiste en invertir el orden de cómputo de soluciones a subproblemas de P (es decir, computarlas bottom-up), y almacenar los resultados de manera tal de no tener que recomputarlos.

Programación Dinámica cambia eficiencia por espacio de almacenamiento, pero en muchos casos este intercambio es extremadamente beneficioso.

Ejemplo: Números de Fibonacci

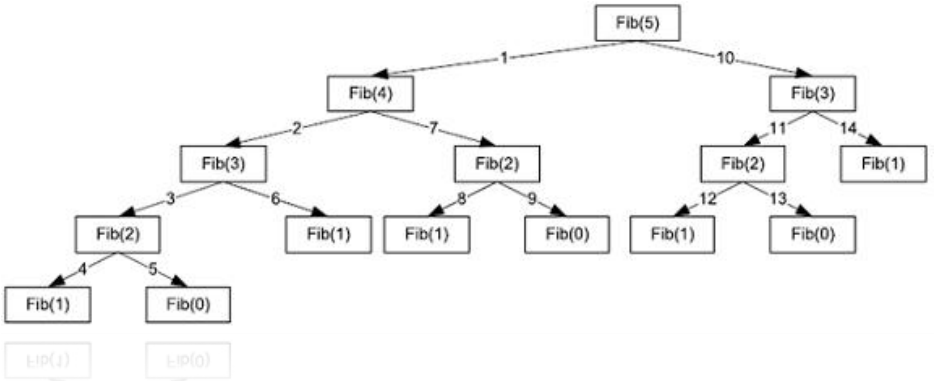
Consideremos el ya conocido problema de computar el n-ésimo número de Fibonacci. Hacer esto recursivamente es simple:

```
fib :: Int -> Int
fib 0 = 1
fib 1 = 1
fib n = (fib (n-1)) + (fib (n-2))
```

Resolver fib n demanda recomputar muchas veces un cantidad importante de subproblemas comunes al cálculo de fib (n-1) y fib (n-2).

Ejemplo: Números de Fibonacci (cont.)

Veamos, por ejemplo, que con 6 como parámetro para fib, ya tenemos una cantidad significativa de problemas que deben recomputarse:



Números de Fibonacci mediante Programación Dinámica

Si pensamos en resolver Fibonacci mediante Programación Dinámica, debemos invertir el orden de cómputo de soluciones a subproblemas (desde los casos base hacia arriba), y almacenar estas soluciones para construir las soluciones a problemas más "grandes". Una forma directa de hacer esto es la siguiente:

```
fib :: Int -> Int
fib n = head (fibList n)

fibList :: Int -> [Int]
fibList n | n==0 = [1]
          | n==1 = [1,1]
          | n>1  = ((head l)+(head (tail l))):l
          where l = fibList (n-1)
```

Números de Fibonacci mediante Programación Dinámica

La versión mediante Programación Dinámica de Fibonacci es O(n), mientras que la versión D&C es $O(2^n)$.

Con respecto al espacio ocupado, la versión mediante Programación Dinámica es O(n). Podemos mejorar aún más este algoritmo, pues sólo necesitamos los dos últimos valores almacenados en la lista, y no la lista completa:

```
fib :: Int -> Int
fib n = fst (fibTuple n)

fibTuple :: Int -> (Int,Int)
fibTuple n  | n == 0 = (1,0)
            | n == 1 = (1,1)
            | n > 1 = ((fst t)+(snd t),fst t)
            where t = fibTuple (n-1)
```

Ejemplo: Cálculo de "Distancia de Edición" entre dos Cadenas

Consideremos el siguiente problema:

> Dadas dos cadenas s1 y s2, calcular el número mínimo de operaciones que permiten transformar s1 en s2, donde las operaciones admisibles son:
>
> - reemplazar un caracter particular en una cadena por otro caracter
> - insertar un nuevo caracter en una posición dada de alguna de las cadenas
> - eliminar un caracter de una posición dada en alguna de las cadenas

La distancia de edición (o distancia de Levenshtein) tiene numerosas aplicaciones, entre ellas en diccionarios para procesadores de textos (e.g., para sugerir palabras alternativas cuando se encuentra una que no pertenece al diccionario), procesos de ayuda en buscadores, etc.

Ejemplo: Cálculo de "Distancia de Edición" entre dos Cadenas (cont.)

Primero, intentemos resolver el problema de calcular la distancia de edición recursivamente:

```
distance :: String -> String -> Int
distance xs [] = length xs
distance [] xs = length xs
distance (x:xs) (y:ys)
  | x == y    = min (distance xs ys) (min ((distance (x:xs) ys)+1) ((distance xs (y:ys))+1))
  | otherwise = min ((distance xs ys)+1) (min ((distance (x:xs) ys)+1) ((distance xs (y:ys))+1))
```

Este algoritmo recursivo es extremadamente ineficiente.

Cálculo de "Distancia de Edición" mediante Programación Dinámica

Intentemos resolver el problema anterior mediante Programación Dinámica. Necesitamos invertir el orden de cómputo de soluciones a subproblemas, y hacerlo "bottom-up".

Cuáles son los subproblemas?

- pares de subcadenas de las cadenas s1 y s2 (todas?)

En qué almacenamos las soluciones?

- en una matriz, indexada por pares de subcadenas de s1 y s2

Cómo "recorremos" la matriz para "poblarla" con valores de distancias?

Cálculo de "Distancia de Edición" mediante Programación Dinámica

Almacenaremos las soluciones a subproblemas en una matriz. Notemos que sólo necesitamos computar distancias de sufijos de las cadenas originales del problema.

	0	1	2	3	4	5	6	7	8	9	10
0	0	1	2	3	4	5	6	7	8	9	10
1	1										
2	2										
3	3										
4	4										
5	5										
6	6										
7	7										
8	8										
9	9										
10	10										

D[i,j]: distancia de edición entre los últimos i caracteres de s1 y los últimos j caracteres de s2.

Ej.: s1 = "abcdefghij", s2 = "bcdefghijk"

Cómo llenamos el resto de la matriz?

Dónde buscamos la solución al problema?

Cálculo de "Distancia de Edición" mediante Programación Dinámica

El siguiente programa resuelve el problema de calcular la distancia de edición entre dos cadenas:

```
public static int computeLevenshteinDistance(String str1, String str2) {
       int[][] distance = new int[str1.length()+1][];

       for(int i=0; i<=str1.length(); i++){
           distance[i] = new int[str2.length()+1];
           distance[i][0] = i;
       }
       for(int j=0; j<str2.length()+1; j++)
           distance[0][j]=j;

       for(int i=1; i<=str1.length(); i++)
           for(int j=1;j<=str2.length(); j++) {
               distance[i][j]= minimum(distance[i-1][j]+1,
                                       distance[i][j-1]+1,
                                       distance[i-1][j-1]+((str1.charAt(i-1)==str2.charAt(j-1))?0:1));
           }

       return distance[str1.length()][str2.length()];
   }
```

Algunos Puntos Importantes sobre Programación Dinámica

- Para poder encarar una solución a un problema mediante Programación Dinámica debe contarse antes con una solución recursiva/Divide & Conquer/Decrease & Conquer para el problema
- Programación Dinámica es un ejemplo importante de intercambio de eficiencia en tiempo por eficiencia en espacio.
- En casos en que el almacenamiento de soluciones a subproblemas es realizado sobre una estructura del estilo de arreglos, listas o matrices, debe asegurarse que el "poblado" de la estructura con valores se haga en un orden adecuado: al computar un valor dado ya deben haberse computado los valores correspondientes a los subproblemas del problema corriente.

Ejemplo: Caminos de Costo Mínimo entre Vértices de un Grafo

Consideremos el ya conocido problema de computar el costo mínimo de "viajar" entre cada par de nodos, en un grafo dirigido con costos en los arcos.

Una forma de resolver este problema es mediante la siguiente función:

```
shortestPath g x y n : "costo mínimo de viajar en g desde x a y
                        pasando por nodos etiquetados con valores hasta n inclusive"

minimalCost g x y = shortestPath g x y (length (vertices g))

shortestPath g x y 0 = g x y
shortestPath g x y n =
     min (shortestPath g x y (n-1)) ((shortestPath g x n (n-1))+(shortestPath g n y (n-1)))
```

Caminos de Costo Mínimo entre Vértices de un Grafo con PD

El algoritmo anterior puede resolverse usando Programación Dinámica de manera mucho más eficiente. El pseudo-código de un algoritmo basado en Programación Dinámica y la solución recursiva anterior es el siguiente:

ALGORITHM *Floyd*$(W[1..n, 1..n])$

//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
$D \leftarrow W$ //is not necessary if W can be overwritten
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $D[i, j] \leftarrow \min\{D[i, j], D[i, k] + D[k, j]\}$
return D

Resumen de la Clase

- La Programación Dinámica es una técnica de diseño de algoritmos fuertemente ligada a recursión/D&C, que intenta mejorar la eficiencia de tales soluciones invirtiendo el orden de cómputo de soluciones a subproblemas.
- El cómputo de soluciones se realiza de manera "bottom-up" (desde los casos base hacia los casos más complejos), en lugar del típico mecanismo "top-down" correspondiente a la recursión.
- El uso de Programación Dinámica es un ejemplo importante de intercambio de eficiencia en tiempo por eficiencia en espacio.
- En casos en que el almacenamiento de soluciones a subproblemas es realizado sobre una estructura del estilo de arreglos, listas o matrices, debe asegurarse que la carga de la estructura con valores se haga en un orden adecuado: al computar un valor dado ya deben haberse computado los valores correspondientes a los subproblemas del problema corriente.
- La Programación Dinámica es una técnica particularmente útil en problemas de optimización.