Programación Dinámica [1]

- El método de programación dinámica sirve para resolver problemas combinando las soluciones de subproblemas.

- Normalmente es usada para resolver problemas de optimización.

- Al construir un algoritmo usando la estrategia de programación dinámica es necesario:

1. Caracterizar la estructura de una solución optima.
2. Definir recursivamente *el valor* de una solución óptima.
3. Computar *el valor* de una solución en forma *bottom-up*.
4. [Opcional] Construir una solución óptima a partir de la información computada.

[1] Basado en el libro *Algorithms* de Cormen, Leiserson y Rivest

¿Cuándo usar Programación Dinámica?

Hay dos condiciones que se deben cumplir antes de comenzar a pensar en una solución a un problema de optimización usando programación dinámica.

- **Sub-estructura óptima**. Un problema tiene sub-estructura óptima cuando la solución óptima a un problema se puede componer a partir de soluciones óptimas de sus sub-problemas.
- **Superposición de Problemas**. El cálculo de la solución óptima implica resolver muchas veces un mismo sub-problemas. La cantidad de sub-problema es “pequeña”.

Resolviendo un Problema con Programación Dinámica

- Sopongamos el siguiente problema: dada una cadena de n matrices $\langle A_1, \ldots, A_n \rangle$, donde para cada i $(1 \leq i \leq n)$ la matriz A_i tiene dimensión $p_{i-1} \times p_i$, encuentre una forma de multiplicar las matrices que minimice el número de multiplicaciones escalares a realizar. *Observación 1*: La forma óptima de multiplicar una cadena de matrices está determinado por el número de multiplicaciones a realizar. Para multiplicar una matriz de $p \times q$ por una de $q \times r$ son necesarias pqr operaciones escalares de multiplicación.

  Observación 2: Si multiplicamos tres matrices de 10×100, 100×5 y 5×50, podemos hacerlo con 7500 $((A_1 A_2) A_3)$ o 75000 $(A_1 (A_2 A_3))$ operaciones.

Sin programación dinámica

- Resolver este problema sin programación dinámica implica calcular el número de operaciones para cada posible orden de multiplicación de matrices.

- ¿Cuántos posibles ordenes hay?

  Respuesta: Muchos.

  Sea $P(n)$ el número de órdenes posibles en una cadena de n matrices. Es sencillo ver que:

$$P(n) = \begin{cases} 1 & \text{si } n = 1 \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{si } n \geq 2 \end{cases}$$

  La solución a esta ecuación es

$$P(n) = \frac{1}{n}\binom{2n-2}{n-1} = \Omega(4^n/n^{3/2})$$

Con programación dinámica

- Supongamos que tenemos la forma óptima de multiplicar las matrices $\langle A_1, \ldots, A_n \rangle$. Al nivel más alto, la solución se verá como la multiplicación de dos matrices que resultan de calcular los productos $A_1 \cdots A_k$ y $A_{k+1} \cdots A_n$, ambos **en forma óptima**, para algún k ($1 \leq k \leq n$).

- Lo anterior implica que el problema tiene sub-estructura óptima.

- Supongamos que llamamos $m[i,j]$ al número óptimo de multiplicaciones escalares a realizar al multiplicar $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.

  $m[i,j]$ se puede escribir recursivamente por:

$$m[i,j] = \begin{cases} 0 & \text{si } i = j \\ \min_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{si } i < j \end{cases}$$

- Supongamos que $s[i,j]$ es la forma en que, al nivel más alto, se divide el producto de $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, de manera óptima. Entonces,

$$m[i,j] = \begin{cases} i & \text{si } i = j \\ \text{argmin}_{i \leq k < j}\{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\} & \text{si } i < j \end{cases}$$

- Si se observan cuidadosamente estas expresiones, nos daremos cuenta que es posible resolver el problema de forma recursiva tradicional, pero **muchos cálculos se deberán rehacer**. Esto significa que existe superposición de problemas.

Bottom-Up: Lo más fácil

- La estrategia *bottom-up* consiste en resolver primero los subproblemas más pequeños, **almacenar su solución**, y luego resolver los problemas más complejos, usando los resultados almacenados.

- Es claro que calcular $m[i, i]$ es muy sencillo... ¿cuál tipo de problema es el que le sigue en complejidad?

  Resp: calcular $m[i, i + 1]$, con $1 \leq i < n$.

El algoritmo

Matrix-Chain-Order(p)

$n \leftarrow length[p] - 1$
for $i \leftarrow 0$ **to** n
    do $m[i,i] \leftarrow 0$
for $l \leftarrow 2$ **to** n
    do for $i \leftarrow 1$ **to** $n - l + 1$
        do $j \leftarrow i + l - 1$
            $k = s[i,j] = \text{argmin}_{i \leq k < j} \{m[i,k] + m[k+1,j] + p_{i-1}p_kp_j\}$
            $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
return m, s

Es sencillo verificar que el tiempo de ejecución de este algoritmo es $O(n^3)$.

Otro Problema: La mayor subsecuencia común (PMSC)

- Una **subsecuencia** de una secuencia dada $S = \langle s_1, s_2, \ldots, s_n \rangle$ es S con algunos (posiblemente ningún) elementos removidos.
- Si X e Y son secuencias entonces Z es una **subsecuencia común** de X e Y si Z es subsecuencia de X e Y.
- Si X e Y son secuencias entonces Z es una **mayor subsecuencia común** (MSC) de X e Y si Z es subsecuencia común de X e Y y no hay otra más larga.
- Entonces, ¿cómo es posible encontrar la mayor subsecuencia de dos secuencias dadas $X = \langle x_1, \ldots, x_m \rangle$ e $Y = \langle y_1, \ldots, y_n \rangle$?
- La solución de fuerza bruta debe probar $2^{\min\{m,n\}}$ subsecuencias...

Sub-estructura óptima en el MSC

- Sean $X = \langle x_1, \ldots, x_m \rangle, Y = \langle y_1, \ldots, y_n \rangle$ y sea $Z = \langle z_1, \ldots, z_k \rangle$ una MSC de ellas.

- Para simplificar la notación decimos que si $X = \langle x_1, \ldots, x_n \rangle$, entonces $X_j = \langle x_1, \ldots, x_j \rangle$ $(0 \leq j \leq n)$.

- El MSC tiene sub-estructura óptima. De hecho,

1. Si $x_m = y_n$, entonces $z_k = x_m = y_n$ y además Z_{k-1} es una MSC de X_{m-1} y Y_{n-1}.
2. Si $x_m \neq y_n$, entonces Z la más grande entre las MSC's de X_{m-1} y Y_n y de X_m y Y_{n-1}.

Una expresión recursiva para el MSC

- Supongamos que llamamos $c[i,j]$ al largo de la MSC entre X_i e Y_k.

- La expresión recursiva para $c[i,j]$ es la siguiente:

$$c[i,j] = \begin{cases} 0 & \text{si } i = 0 \text{ o } j = 0 \\ c[i-1, j-1] + 1 & \text{si } i, j > 0 \text{ y } x_i = y_j \\ \text{máx}\{c[i-1, j], c[i, j-1]\} & \text{si } i, j > 0 \text{ y } x_i \neq y_j \end{cases}$$

- Para encontrar la subsecuencia podemos definir $b[i,j]$ con el siguiente significado:
  - $b[i,j]$ tiene valor 1 si la MSC de X_i y X_j contiene a x_i ($x_i = x_j$).
  - $b[i,j]$ tiene valor $\uparrow$ si el último elemento de la MSC de X_i y X_j es igual al último elemento de la MSC de X_{i-1} y Y_j.
  - $b[i,j]$ tiene valor $\downarrow$ si el último elemento de la MSC de X_i y X_j es igual al último elemento de la MSC de X_i y Y_{j-1}.

- A partir de $b[i,j]$ es muy sencillo construir una MSC.

Algoritmo para la MSC

```
MSC(X, Y)
m ← largo[X]
n ← largo[Y]
for i ← 1 to m
do c[i, 0] ← 0
for j ← 1 to n
do c[0, j] ← 0
for i ← 1 to m
do for i ← 1 to n
    do if x_i = y_j
          then c[i, j] ← c[i − 1, j − 1] + 1
               b[i, j] ← 1
          else if c[i − 1, j] ≥ c[i, j − 1]
                  then c[i, j] ← c[i − 1, j]
                       b[i, j] ←'↑'
                  else c[i, j] ← c[i, j − 1]
                       b[i, j] ←'↓'
return c, b
```

Su tiempo de ejecución de es $O(mn)$.