Programación dinámica

Pedro O. Pérez M., MTI

Análisis y diseño de algoritmos
Tecnológico de Monterrey

pperezm@tec.mx

04-2019

Contenido

Definición

La programación dinámica, al aigual que dividir y conquistar, resuelve problemas combinanco soluciones a subproblemas; pero a diferencia de esta, se aplica cuando los subproblemas se solapan, es decir, cuando comparten problemas más pequeños. Aquí la técnica cobra importancia, ya que calcula cada subproblema una sola vez; esto es, parte del principio de no calcular dos veces la misma información. Por tanto, utiliza estructuras de almacenamiento como vectores, tablas, arreglos, archivos, con el fin de almacenar los resultados parciales a medida que se resuelven los subcasos que contribuyen a la solución definitiva.

- Es una técnica ascendente que, normalmente, empieza por los subcasos más pequeños y más sencillos. Combinando sus soluciones, obtenemos las respuestas para los subcasos cada vez más grandes, hasta que llegamos a la solución del problema original.
- Se aplica muy bien a problemas de optimización. El mayor número de aplicaciones se encuentran en problemas que requieren maximización o minimización, ya que se pueden hallar múltiples soluciones y así evaluar para determinar cuál es la óptima.

Forma general

La forma general de las soluciones desarrolladas mediante programación dinámica requiere los siguientes pasos:

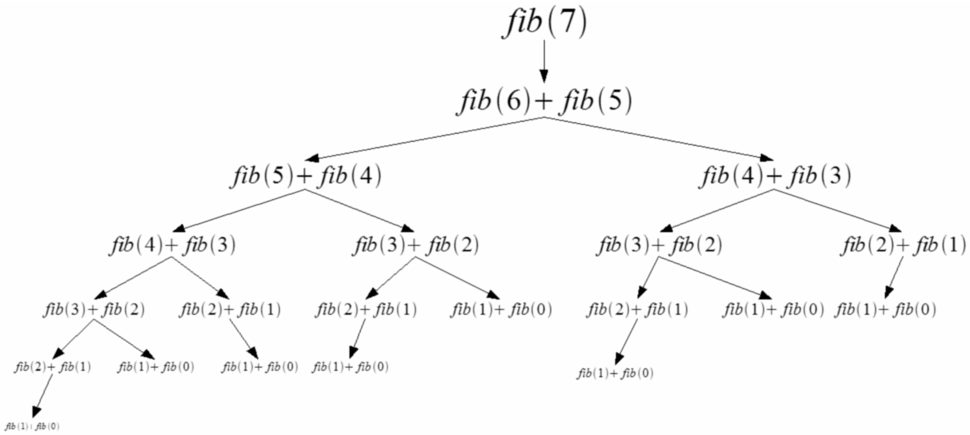
1. Plantear la solución, mediante una serie de decisiones que garanticen que será óptima, es decir, que tendrá la estructura de una solución óptima.
2. Encontrar una solución recursiva de la definición.
3. Calcular la solución teniendo en cuenta una tabla en la que se almacenen soluciones a problemas parciales para su reutilización, y así evitar un nuevo cálculo.
4. Encontrar la solución óptima utilizando la información previamente calcular y almacenada en las tablas.

Principio de optimalidad de Bellman: Cualquier subsecuencia de decisiones de una secuencia óptima de decisiones que resuelve un problema también debe ser óptima respecto al subproblema que resuelve.

Procedure 1 FIBONACCI

Input: n : *Integer*

```
if n < 1 then
    return −1
else if n = 1 or n = 2 then
    return 1
else
    return FIBONACCI(n − 1) + FIBONACCI(n − 2)
end if
```


fib(7)
fib(6)+ fib(5)
fib(5)+ fib(4)
fib(4)+ fib(3)
fib(4)+ fib(3)
fib(3)+ fib(2)
fib(3)+ fib(2)
fib(2)+ fib(1)
fib(3)+ fib(2)
fib(2)+ fib(1)
fib(2)+ fib(1)
fib(1)+ fib(0)
fib(2)+ fib(1)
fib(1)+ fib(0)
fib(1)+ fib(0)
fib(2)+ fib(1)
fib(1)+ fib(0)
fib(1)+ fib(0)
fib(1)+ fib(0)
fib(1)+ fib(0)
fib(1)+ fib(0)

Procedure 2 FIBONACCI_WITH_MEMORY1

Input: n : *Integer*, A : *Array*

if $n < 1$ **then**
  return -1
else if $n = 1$ **or** $n = 2$ **then**
  return 1
else if $A[n] <> -1$ **then**
  return $A[n]$
else
  $A[n] \leftarrow FIBONACCI(n-1) + FIBONACCI(n-2)$
  return $A[n]$
end if

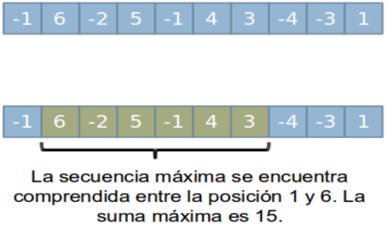
Procedure 3 FIBONACCI_WITH_MEMORY2

Input: n : *Integer*, A : *Array*

```
if n < 1 then
  return −1
else
  A[1] ← 1
  A[2] ← 1
  for i ← 3 to n do
    A[i] ← A[i − 1] + A[i − 2]
  end for
end if
```

Secuencia de suma máxima

Dado un arreglo de n números enteros positivos y negativos, encontrar los i elementos del arreglo cuya suma se la máxima posible.



Procedure 4 MAX_SUM(A:Array)

Input: $n : Integer, A : Array$

    $sum \leftarrow 0$
    $ans \leftarrow 0$
    for $i \leftarrow 1$ to $A.length$ **do**
        $sum \leftarrow sum + A[i]$
        $ans \leftarrow MAX(ans, sum)$
        if $sum < 0$ **then**
            $sum \leftarrow 0$
        end if
    end for
    return ans

Cambio de monedas

Dado un sistema monetario S con N monedas de diferentes denominaciones y una cantidad de cambio C, calcular el menor número de monedas del sistema monetario S equivalente a C.

Ejemplos:

- **Input** : s[] = 1, 3, 4 c = 6
  Output : 2
  Explanation : The change will be (3 + 3) = 2

$$C[j] = \begin{cases} \infty & \text{if } j < 0, \\ 0 & \text{if } j = 0, \\ 1 + \min\limits_{1 \leq i \leq k} \{C[j - d_i]\} & \text{if } j \geq 1 \end{cases}$$

Procedure 5 COIN_CHANGE

Input: $S : Array, c : Integer$

$Aux : Array[0, c]$
$INIT\_ARRAY(A, \infty)$
$Aux[0] \leftarrow 0$
for $j \leftarrow 1$ to $S.length$ **do**
  for $j \leftarrow S[i]$ to c **do**
    $Aux[j] \leftarrow MIN(1 + Aux[j - S[i]], Aux[j])$
  end for
end for
return Aux[c]

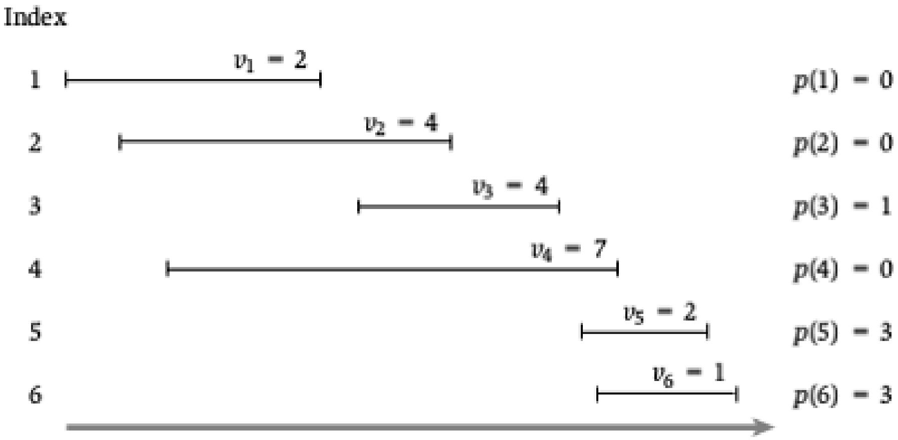
Programación de actividades

Te dan N actividades con sus tiempos de inicio (S_i) y finalización (F_i). Selecciona el número máximo de actividades que puede realizar una sola persona, asumiendo que una persona solo puede trabajar en una sola actividad a la vez. [1]

Ejemplo:

- **Input** :
  start[] = (1, 3, 0, 5, 8, 5)
  finish[] = (2, 4, 6, 7, 9, 9)
  Output : 4

[1] https://goo.gl/1RG25M


Index
1
$v_1 = 2$
$p(1) = 0$
2
$v_2 = 4$
$p(2) = 0$
3
$v_3 = 4$
$p(3) = 1$
4
$v_4 = 7$
$p(4) = 0$
5
$v_5 = 2$
$p(5) = 3$
6
$v_6 = 1$
$p(6) = 3$

Procedure 6 ACTIVITIES_SELECTION

Input: $A : Activity\_Array, Opt : Array$

$Aux : Array[0..A.length]$

$Aux[0] \leftarrow 0$

for $i \leftarrow 1$ to $A.length$ **do**

$Aux[i] \leftarrow MAX(A[i] + Aux[Opt[i]], Aux[j-1])$

end for

Números feos

Los números feos son números cuyos únicos factores primos son 2, 3 o 5 (o combinación de ellos). La secuencia 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... muestra los primeros 11 números feos. Por convención, se incluye 1.

Como vemos en la diapositiva anterior, la secuencia de los primeros números feos es 1, 2, 3, 4, 5, 6, 8, 9, 10, 12, 15, ... Es más fácil de determinar la forma en que se genera está secuencia si la dividimos entre sus tres factores:

- 1x2, 2x2, 3x2, 4x2, 5x2...
- 1x3, 2x3, 3x3, 4x3, 5x3...
- 1x5, 2x5, 3x5, 4x5, 5x5...

¿Cuál se imprime primero?

Procedure 7 UGLY_NUMBER

Input: n : *Integer*

A : *Array*$[n]$

$A[1] = 1$

$i2 \leftarrow 1$

$i3 \leftarrow 1$

$i2 \leftarrow 1$

$ugly\_number \leftarrow 0$

for $i \leftarrow 1$ to n **do**

next slide

end for

return $ugly\_number$

$ugly\_number = MIN((A[i2] * 2), (A[i3] * 3), (A[i5] * 5))$
$A[i] \leftarrow ugly\_number$
if $ugly\_number == (A[i2] * 2)$ **then**
  $i2 \leftarrow i2 + 1$
end if
if $ugly\_number == (A[i3] * 3)$ **then**
  $i3 \leftarrow i3 + 1$
end if
if $ugly\_number == (A[i5] * 5)$ **then**
  $i5 \leftarrow i5 + 1$
end if

¿Qué es una subsecuencia?

Considera una cadena A = [a, b, c, d]. Una subsecuencia(también llamada subcadena) se obtiene eliminando 0 o más símbolos (no necesariamente consecutivos) de A.

Por ejemplo:

- Los ejemplos [a, b, c, d] , [a], [b], [c], [d], [a, d], [a, c], [b, d] son subsecuencias de A.
- Pero [d, b], [d, a], [d, a, c] no son subsecuencias de A.

Subsecuencia creciente más larga

Dado un arreglo de n símbolos comparables, dar la subsecuencia creciente más larga. Por ejemplo, A = [-7, 10, 9, 2, 3, 8, 8, 1], la subsecuencia creciente más larga es [-7, 2, 3, 8].

Procedure 8 LIS

Input: $A : Array$

$Aux : Array$
$Aux[1] \leftarrow 1$
for $i \leftarrow 2$ to n **do**
  $count \leftarrow 0$
  for $j \leftarrow 1$ to $(i-1)$ **do**
    if $A[j] < A[i]$ **then**
      $count \leftarrow MAX(count, Aux[j])$
    end if
  end for
  $Aux[i] \leftarrow count + 1$
end for

$count \leftarrow 0$
for $i \leftarrow 1$ to n **do**
    $count \leftarrow MAX(count, Aux[i])$
end for
return $count$

Cuenta el número de formas posibles

Queremos dar un cambio de C pesos y tenemos un suministro infinito de monedas de valor S = $[S_1, S_2, ..., S_n]$ pesos, ¿de cuántas maneras podemos dar el cambio? Por ejemplo para C = 4, S = [1, 2, 3], existen cuatro formas de dar el cambio: [1, 1, 1, 1], [1, 1, 2], [2, 2], [1, 3]. Para C = 10, S = [2, 5, 3 6], existen cinco soluciones: [2, 2, 2, 2, 2], [2, 2, 3, 3], [2, 2, 6], [2, 3, 5], [5, 5].

$$COUNT(type, value) = \begin{cases} 0, & \text{si } value < 0, \\ 1, & \text{si } value = 0, \\ COUNT(type + 1, value) & \\ +COUNT(type, value - coins[type]), & \text{si } value > type. \end{cases}$$

Procedure 9 COUNT

Input: $S : Array, c : Integer$

```
table : Matrix[0...c][S.length]
for i ← 0 to c do
    table[0][i] ← 1
end for
for i ← 1 to S.length do
    for j ← 1 to c do
        NEXT SLIDE
    end for
end for
return table[n][c]
```

```
acum ← 0
/* Can the currency S [j] be used to give the change? */
if (i − S[j]) ≥ 0 then
    acum ← table[(i − S[j])][j]
end if
/* If we do not use S[j] */
if j ≥ 1 then
    acum ← acum + table[i][j − 1]
end if
table[i][j] ← acum
```

Mochila 0/1

Sean N objetos no fraccionables de pesos w_i, y beneficios v_i, y sea C el peso máximo que puede llevar la mochila. El problema consiste en llenar la mochila con objetos, tal que se maximice el beneficio. Por ejemplo, W = [10, 4, 6, 12], V = [100, 70, 50, 10], y C = 12. ¿Cuál es la beneficio máximo que podemos obtener?

Procedure 10 KNAPSACK

Input: $W, V : Array, C : Integer$

```
M : Matrix[0...S.length][0..C]
n ← W.length
INIT(M, 0)
for i ← 1 to n do
  for j ← 1 to C do
    /* There is not space in the container */
    if j < W[i] then
      M[i][j] ← M[i − 1][j]
    else
      M[i][j] ← MAX(V[i] + M[i − 1][j − W[i]], M[i − 1][j])
    end if
  end for
end for
return M[n][C]
```

Subsecuencia común más larga

Dado dos cadenas, A y B, determinar la subsecuencia común más larga de ambas cadenas. Por ejemplo, si A = [c, d, c, c, f, g, e] y B = [e, c, c, e, g, f, e], ¿cuál la subsecuencia común más larga?

Ahora veamos como se plantea el problema: considere dos secuencias $A = [a_1, a_2, ..., a_m]$ y $B = [b_1, b_2, ..., b_n]$. Para resolver el problema nos fijaremos en los últimos dos símbolos: a_m y b_n . Como podemos ver hay dos posibilidades:

- Caso 1: $a_m = b_n$. En este caso, la subsecuencia común más larga debe contener a_m . Simplemente basta encontrar la subsecuencia común más larga de $[a_1, a_2, ..., a_{m-1}]$ y $[b_1, b_2, ..., b_{n-1}]$.
- Caso 2: $a_m \neq b_n$. En este caso, puede hacerse corresponder $[a_1, a_2, ..., a_m]$ con $[b_1, b_2, ..., b_{n-1}]$ y también $[a_1, a_2, ..., a_{m-1}]$ con $[b_1, b_2, ..., b_n]$, y nos quedamos con el mayor de los dos resultados.

Procedure 11 LCS

Input: $A, B : Array$

$M : Matrix[0..A.length][0..B.length]$
$m \leftarrow A.length$
$n \leftarrow B.length$
$INIT(M, 0)$
for $i \leftarrow 1$ to m **do**
  for $j \leftarrow 1$ to n **do**
    if $A[i] = B[j]$ **then**
      $M[i][j] \leftarrow 1 + M[i-1][j-1]$
    else
      $M[][] \leftarrow MAX(M[i-1][j], M[i][j-1])$
    end if
  end for
end for
return $M[m][n]$

Conectividad total en un grafo

Queremos determinar, para cualquier par de vértices, el costo del camino más corto.

Procedure 12 WARSHAL

Input: $AdjMat$: $Matrix$

$INIT(AdjMat, +\infty)$
for $k \leftarrow 1$ to $AdjMat.length$ **do**
  for $i \leftarrow 1$ to $AdjMat.length$ **do**
    for $j \leftarrow 1$ to $AdjMat.length$ **do**
      $acum \leftarrow AdjMat[i][k] + AdjMat[k][j]$
      if $acum < AdjMat[i][j]$ **then**
        $AdjMat[i][j] \leftarrow acum$
      end if
    end for
  end for
end for