Programación II

Bloque temático 1. Lenguajes de programación

Bloque temático 2. Metodología de programación

Bloque temático 3. Esquemas algorítmicos

Tema 8. Programación dinámica

8.1 Introducción a la Programación Dinámica

PD (*Dynamic Programming*) es un esquema algorítmico que se basa en la utilización de una tabla con soluciones parciales

- la tabla se va llenando con las soluciones de los subcasos
  - empezando por los subcasos más pequeños y construyendo con ellos los grandes
- hasta llegar al caso que se desea resolver
- suelen ser algoritmos *temporalmente eficientes* aunque con requerimientos de *memoria adicional* elevados

PD puede superar en eficiencia a DyV ya que evita repetir cálculos

- no recalcula resultados parciales (se sacan de la tabla)

PD puede resolver problemas sin solución óptima con voraces

- puesto que las decisiones no se toman “a ciegas” sino considerando todos los subcasos

8.2 Utilización de la Programación Dinámica

En general, la programación dinámica se puede aplicar en los mismos casos en que utilizaríamos un algoritmo voraz

- **para resolver *problemas de optimización***
  - **minimizar o maximizar, bajo determinadas condiciones, el valor de una función: $f(x_1,x_2,..,x_n)=c_1x_1+c_2x_2+..+c_nx_n$**
- **en los que se cumple el *principio de optimalidad* (basta con que se cumpla de una forma menos estricta que para los voraces)**

Principio de optimalidad “relajado”: “la solución óptima de un problema es una combinación de soluciones óptimas de *algunos* de sus subcasos”

- **la PD resuelve todos los subcasos, lo que nos permite identificar los que conducen a la solución óptima**
- **en voraces puede no ser posible identificar esos subcasos por que no existe la función de selección apropiada**

Diseño de algoritmos de programación dinámica

Se realiza en los siguientes pasos:

1. **Verificar que la solución puede alcanzarse a partir de una sucesión de decisiones y que ésta cumple el principio de optimalidad**
2. **Encontrar una expresión recursiva para la solución**
3. **Utilizar la expresión recursiva para rellenar la tabla de soluciones parciales hasta encontrar la solución óptima al problema planteado**
4. **Reconstruir la solución desandando sobre la tabla el camino que nos ha llevado a la solución óptima**

8.3 Algoritmo para “dar cambio”

Este mismo problema ya le resolvimos con un algoritmo *voraz*, pero *no siempre obtenía la solución óptima*, p.e.:

- **Dar cambio con el antiguo sistema monetario ingles**
  - **corona (30p), florín (24p), chelín (12p), 6p, 3p, penique**
- **Solución del algoritmo voraz para 48p: 30p+12p+6p**
  - **solución óptima: 24p+24p**

El problema verifica el principio de optimalidad:

- **si la solución óptima para una cantidad c contiene la moneda `m` (de valor v_m)**
- **se verifica que la solución óptima para c es una moneda `m` más la solución óptima para $c-v_m$**

El algoritmo basado en *programación dinámica* que vamos a ver *siempre obtiene la solución óptima*

Algoritmo para “dar cambio” (cont.)

Supongamos que deseamos cambiar una cantidad de `cCambiar` unidades en un sistema monetario que tiene `M` monedas

- **de valores v_m ($0 \leq m \leq M-1$) (p.e.: $M=3$ y $v_0=1$, $v_1=4$, $v_2=6$)**

Para resolver el problema mediante programación dinámica crearemos una tabla `tabla[0..M-1, 0..cCambiar]`

- **`tabla[m,c]` representa el número mínimo de monedas de tipo `m` o menor necesario para devolver una cantidad `c`**
  - **p.e.: `tabla[1,7]` es 4 (una moneda de $v_1=4$ y tres de $v_0=1$)**

Debemos idear una expresión recursiva que nos permita construir la tabla, así, para `tabla[m,c]` elegiremos el mínimo de:

- **no utilizar monedas de valor v_m, en ese caso: `tabla[m,c]=tabla[m-1,c]`**
- **o incluir una moneda de valor v_m, en ese caso: `tabla[m,c]=1+tabla[m,c-`v_m`]`**

Algoritmo para “dar cambio” (cont.)

De lo anterior, junto con las condiciones de contorno, se obtiene la expresión recursiva que nos permite calcular la tabla:

$$\mathrm{tabla}[m, c] = \begin{cases} 0 & \text{si} \quad c = 0 \\ c & \text{si} \quad m = 0 \\ \mathrm{tabla}[m-1, c] & \text{si} \quad v_m > c \\ \min(\mathrm{tabla}[m-1, c],\ 1 + \mathrm{tabla}[m, c - v_m]) & \text{en otro caso} \end{cases}$$

P.e. para el conjunto de tres monedas $v_0=1$, $v_1=4$ y $v_2=6$ y `cCambiar=8` la tabla `tabla[0..2,0..8]` será:

Valor moneda	cantidad a cambiar								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$v_0=1$	0	1	2	3	4	5	6	7	8
$v_1=4$	0	1	2	3	1	2	3	4	2
$v_2=6$	0	1	2	3	1	2	1	2	2

Número de monedas a devolver

Algoritmo para “dar cambio” (cont.)

Para obtener las monedas a devolver:

- **comenzamos en la última posición: `m=M-1` y `c=cCambiar`**
- **si `tabla[m,c] == tabla[m-1,c]`**
  - **la solución no incluye ninguna moneda de tipo `m` y pasamos a la posición `tabla[m-1,c]`**
- **sino (`tabla[m,c] == 1+tabla[m,c-`v_m`]`)**
  - **se incluye en la solución una moneda de tipo `m` y pasamos a la posición `tabla[m,c-`v_m`]`**
- **se continua de esta manera hasta alcanzar la fila `0`**
  - **se añade al cambio `tabla[0,c]` monedas y se finaliza**

Valor moneda	cantidad a cambiar								
	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
$v_0=1$	0	1	2	3	4	5	6	7	8
$v_1=4$	0	1	2	3	1	2	3	4	2
$v_2=6$	0	1	2	3	1	2	1	2	2

Implementación

```
método creaTabla(entero cCambiar, entero[0..M-1] v)
    retorna entero[0..M-1, 0..cCambiar]
  entero[0..M-1, 0..cCambiar] tabla
  desde m:=0 hasta M-1 hacer
    desde c:=0 hasta cCambiar hacer
      si
        c==0          => tabla[m,c] := 0
        m==0          => tabla[m,c] := c
        c < v[m]      => tabla[m,c] := tabla[m-1,c]
        en otro caso =>
          tabla[m,c] := min(tabla[m-1,c],
                            1+tabla[m,c-v[m]])
      fsi
    fhacer
  fhacer
  retorna tabla
fmétodo
```



```
método daCambio(entero[0..M-1] v,
    entero[0..M-1,0..cCambiar] tabla)
    retorna entero[0..M-1]
  entero[0..M-1] cambio
  m=M-1     c=cCambiar
  mientras m!=0 hacer
    si tabla[m,c] == tabla[m-1,c] entonces
      m--
    sino
      cambio[m]++
      c = c - v[m];
    fsi
  fhacer
  cambio[0] = tabla[0,c]
  retorna cambio
fmétodo
```


Prestaciones

Eficiencia temporal del algoritmo `creaTabla`:

- el bucle externo se hace `M` veces y el interno `cCambiar+1` veces
- luego su eficiencia es `O(M·cCambiar)`

Eficiencia temporal del algoritmo `daCambio`:

- ir desde la fila `M-1` hasta la `0`: `M` pasos
- ir desde la columna `cCambiar` hasta la `0`: tantos pasos como monedas hay en la solución (`cambio[M-1,cCambiar]`)
- luego su eficiencia es `O(M+cambio[M-1,cCambiar])`

Sus requerimientos de memoria son `O(M·cCambiar)` (la tabla)

La eficiencia temporal del voraz también era `O(M·cCambiar)`

- con mejores requerimientos de memoria: `O(1)`
- pero no es óptimo para algunos conjuntos de monedas

8.4 Bibliografía

[1] Brassard G., Bratley P., *Fundamentos de algoritmia*. Prentice Hall, 1997.

[2] Aho A.V., Hopcroft J.E., Ullman J.D., *Estructuras de datos y algoritmos*. Addison-Wesley, 1988.

[3] Sartaj Sahni, *Data Structures, Algoriths, and Applications in Java*. Mc Graw Hill, 2000