Algoritmos y Estructuras de Datos II Clase 12

Carreras:

Licenciatura en Informática

Ingeniería en Informática

2024

Unidad III

Técnicas de diseño de algoritmos

1
1 1
1 2 1
1 3 3 1
1 4 6 4 1
1 5 10 10 5 1
1 6 15 20 15 6 1
1 7 21 35 35 21 7 1
1 8 28 56 70 56 28 8 1
1 9 36 84 126 126 84 36 9 1
1 10 45 120 210 252 210 120 45 10 1
1 11 55 165 330 462 462 330 165 55 11 1
1 12 66 220 495 792 924 792 495 220 66 12 1
1 13 78 286 715 1287 1716 1716 1287 715 286 78 13 1
1 14 91 364 1001 2002 3003 3432 3003 2002 1001 364 91 14 1
1 15 105 455 1365 3003 5005 6435 6435 5005 3003 1365 455 105 15 1

Programación Dinámica(1)

Programación Dinámica

Surgen dificultades cuando algunos problemas se resuelven con:

- **Algoritmos Recursivos**

Inconveniente: tiempo de ejecución de orden exponencial y por tanto impracticable.

- **Algoritmos Divide & Conquer**

Inconveniente: los subproblemas obtenidos no son independientes sino que existe solapamiento entre ellos.

- **Algoritmos Greedy**

Inconveniente: no sirven para <u>todos</u> los problemas de optimización.

- En todos estos casos no se puede obtener una solución y es cuando la ***Programación Dinámica*** puede ofrecer una solución de eficiencia aceptable.
- La eficiencia de la **Programación Dinámica** consiste en resolver los subproblemas una sola vez, guardando sus soluciones en una tabla para su futura utilización.

Programación Dinámica

- ***Programación Dinámica*** traducción del ingles: ***Dynamic Programming***, es una técnica general de diseño de algoritmos usada comúnmente para resolver problemas de optimización.

- Richard Bellman, un prominente matemático de USA, fue pionero en el estudio sistemático de la programación dinámica en los años 1950. Inventó la técnica y la presentó como un método general para optimizar un proceso multi-etapa de decisiones.

- La palabra “*programación*” en esta técnica significa “*planificación* ” y no se refieren a programación de computadoras.

Programación Dinámica

La mayor aplicación de la Programación Dinámica es en la resolución de *problemas de optimización*. En este tipo de problemas se pueden encontrar distintas soluciones, cada una con un valor, y lo que se elige es una *solución de valor óptimo* (máximo o mínimo).

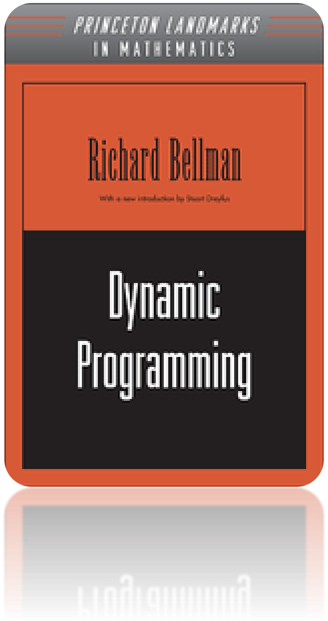
Ventajas de la Programación Dinámica:

- Es un método capaz de resolver de manera eficiente problemas cuya solución ha sido abordada por otras técnicas y ha fracasado.
- Mayor eficiencia.
- El problema se convierte en problema multi-etapas.
- Ayuda a resolver el problema etapa por etapa en forma sistemática.

Programación Dinámica

La solución de problemas mediante esta técnica se basa en el llamado principio de optimalidad enunciado por *Bellman en 1957* y que dice:

"En una secuencia de decisiones óptima toda subsecuencia ha de ser también óptima".



- Observar que aunque este principio parece evidente no siempre es aplicable y por tanto es necesario verificar que se cumple para el problema en cuestión.

- Cuando el principio de optimalidad no es aplicable, es probable que no sea posible atacar el problema que se quiere resolver empleando programación dinámica.

- ***Dynamic Programming***
  Richard E. Bellman 'With a new introduction by Stuart Dreyfus
  Princeton University Press, 2010.

Programación Dinámica

- El ***principio de optimalidad*** resulta aplicable en muchas situaciones.

- Es posible enunciarlo en la forma siguiente:

La solucion óptima de cualquier caso no trivial de un problema es una combinación de soluciones óptimas de algunos de sus subcasos.

- La mayor dificultad para transformar este principio en un algoritmo es que no suele ser evidente cuales son los subcasos relevantes para el caso considerado.

Programación Dinámica

Resolver un problema como una *secuencia de decisiones* equivale a dividirlo en subproblemas de tamaño menor, en principio más fácilmente resolubles.

La ventaja sobre Divide & Conquer es que la Programación Dinámica se puede aplicar cuando la subdivisión de un problema conduce a:

- Una gran cantidad de subproblemas.
- Subproblemas cuyas soluciones parciales se solapan.
- Grupos de subproblemas de muy distinta complejidad.

Programación Dinámica

Existen dos atributos claves que un problema debe tener para que se pueda aplicar la metodología de Programación Dinámica:

- ***Subestructura optima:*** la solución al problema de optimización se puede obtener por la combinación de soluciones optimas a sus subproblemas.
- ***Superposición de subproblemas:*** el espacio de subproblemas debe ser mas chico, el algoritmo resolverá los mismos subproblemas, en lugar de generar nuevos subproblemas. Se puede conseguir de dos maneras:
  - Top-down: solución con formulación recursiva, con almacenamiento de soluciones en tabla.
  - Bottom-up: resolver los subproblemas primero y usar esas soluciones para construir y llega a la solución de problemas mayores.

Programación Dinámica

Concretamente, para que un problema pueda ser resuelto con la técnica de programación dinámica, debe cumplir con ciertas características:

- El problema puede ser dividido en etapas.
- La solución al problema es alcanzada a través de una secuencia de decisiones, una en cada etapa.
- Dicha secuencia de decisiones tiene que cumplir el principio de óptimo.
- Cada etapa tiene un número de estados asociados a ella.
- La decisión óptima de cada etapa depende solo del estado actual y no de las decisiones anteriores.
- La decisión tomada en una etapa determina cual será el estado de la etapa siguiente.

Programación Dinámica

En grandes líneas, el diseño de un algoritmo de Programación Dinámica consta de los siguientes pasos:

1. **Planteamiento** de la solución como una *sucesión de decisiones* y verificación de que ésta cumple el principio de óptimo.

2. **Definición** *recursiva* o *iterativa* de la solución.

3. **Cálculo** del valor de la solución óptima mediante una *tabla* en donde se almacenan soluciones a problemas parciales para reutilizar los cálculos.

4. **Construcción** de la *solución óptima* haciendo uso de la información contenida en la tabla generada en el paso anterior.

Programación Dinámica

- A diferencia de la estrategia greedy, se producen *varias secuencias de decisiones* y solamente al final se sabe cuál es la mejor de ellas.

- Se puede suponer que un problema se resuelve tras tomar un secuencia: $\boldsymbol{d_1, d_2, \ldots, d_n}$ de decisiones.

- Si hay *d* opciones posibles para cada una de las decisiones, una técnica de fuerza bruta exploraría un total de d^n secuencias posibles de decisiones (explosión combinatoria).

- La técnica de programación dinámica evita explorar todas las secuencias posibles por medio de la resolución de subproblemas de tamaño creciente y almacenamiento en una tabla de las soluciones óptimas de esos subproblemas para facilitar la solución de los problemas más grandes.

Programación Dinámica

Divide & Conquer	Programación Dinámica

Programación Dinámica

Divide & Conquer	**Programación Dinámica**
El problema se divide en subproblemas más pequeños e independientes entre sí.	El problema se divide en subproblemas que se solapan entre sí.

Programación Dinámica

Divide & Conquer	**Programación Dinámica**
El problema se divide en subproblemas más pequeños e independientes entre sí.	El problema se divide en subproblemas que se solapan entre sí.
No se almacenan las soluciones a los subproblemas.	Se almacenan todas las soluciones a los subproblemas.

Programación Dinámica

Divide & Conquer	Programación Dinámica
El problema se divide en subproblemas más pequeños e independientes entre sí.	El problema se divide en subproblemas que se solapan entre sí.
No se almacenan las soluciones a los subproblemas.	Se almacenan todas las soluciones a los subproblemas.
Se repite el cálculo de subproblemas idénticos.	No se recalcula los subproblemas idénticos.

Programación Dinámica

Divide & Conquer	**Programación Dinámica**
El problema se divide en subproblemas más pequeños e independientes entre sí.	El problema se divide en subproblemas que se solapan entre sí.
No se almacenan las soluciones a los subproblemas.	Se almacenan todas las soluciones a los subproblemas.
Se repite el cálculo de subproblemas idénticos.	No se recalcula los subproblemas idénticos.
Top-Down.	Bottom-Up.

Programación Dinámica

Ejemplos ya conocidos:

- Fibonacci no recursivo
- Cálculo de coeficientes binomiales
- Algoritmo de Warshall
- Algoritmo de Floyd.

Programación Dinámica

Algunos Ejemplos nuevos:

- Monedas en fila
- Dar vuelto a un cliente
- Problema de la Mochila 0/1
- Suma exacta
- Multiplicación óptima de matrices
- Probabilidad entre dos equipos
- Camino mínimo en una dirección
- Problema del viajante
- Y muchos mas…

Problema: monedas en fila

Se tienen n monedas en una línea, cuyos valores son números enteros positivos $d1,d2,\ldots,dn$ no necesariamente distintos.

Se quiere tomar el ***máximo monto en monedas*** sin alzar dos monedas adyacentes en la fila.

Por ejemplo: 2, 1, 0.25, 0.01, 0.10, 5, 0.50, 0.05, 10

Problema: monedas en fila

Sea **F(*n*)** el ***máximo monto*** que se puede tomar de la fila de ***n*** monedas.

Entonces: **F(0)=0** y **F(1)=*d1***

Se avanzará tomando monedas de la fila decidiendo en el paso i=2,n entre:

- Tomar una moneda de valor ***di***:

  F(i) = *di* + F(i-2)

- No tomar una moneda de valor ***di***:

  F(i) = F(i-1)

Como es un problema de optimización:

F(i) = Máximo { *di* + F(i-2) , F(i-1) }   con i=2,n

Problema: monedas en fila

Ejemplo: Fila de 6 monedas de valor: 5, 1, 2, 10, 6, 2

F(i) = Máximo { di + F(i-2) , F(i-1) } , i=2,n

0	1	2	3	4	5	6
	5	1	2	10	6	2

Inicial: F(0)=0, F(1)=5

0	1	2	3	4	5	6
	5	1	2	10	6	2
0	5					

i=2: F(2)=max{1+0,5}=5

0	1	2	3	4	5	6
	5	1	2	10	6	2
0	5	5				

i=3: F(3)=max{2+5,5}=7

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

i=4: F(4)=max{10+5,7}=15

0	1	2	3	4	5	6
	5	1	2	10	6	2
0	5	5	7	15		

i=5: F(5)=max{6+7,15}=15

0	1	2	3	4	5	6
	5	1	2	10	6	2
0	5	5	7	15	15	

i=6: F(6)=max{2+15,15}=17

0	1	2	3	4	5	6
	5	1	2	10	6	2
0	5	5	7	15	15	**17**

Solución

Problema: dar vuelto a un cliente

- Ya se presentó un algoritmo greedy que resuelve el problema de dar vuelto a un cliente con el mínimo número de monedas.

- Este algoritmo greedy funciona *solamente en número limitado de casos.*

- Se puede plantear una solución más general usando programación dinámica.

- El método consiste en preparar una tabla que contenga resultados intermedios útiles, que serán combinados en la solución del problema.

Problema: dar vuelto a un cliente

- El sistema monetario tiene monedas de n denominaciones diferentes.
- Cada moneda de denominación i, con $1 \leq i \leq n$ tiene un valor de d_i unidades.

- Suponer además que los $d_i > 0$
- Se dispone de una cantidad ***ilimitada*** de monedas de cada denominación.
- El problema consiste en dar vuelto a un cliente por valor de M unidades con *el* ***menor número posible de monedas.***

Problema: dar vuelto a un cliente

Algunas aclaraciones:

- Si está disponible un suministro inagotable de monedas con un valor de una unidad, entonces siempre se puede encontrar una solución para el problema.
- De no ser así puede haber algunos valores para los que no exista una solución.
- Puede ser por ejemplo el caso que se tenga que dar un vuelto impar y todas las monedas disponibles tengan valor par.
- En todos los casos en que no haya solución, el algoritmo debe devolver como resultado ∞.

Problema: dar vuelto a un cliente

- Para resolver el problema por ***programación dinámica*** se prepara una tabla ***C(1..n,0..M)*****,** con una fila por cada denominación posible de moneda y una columna para el valor del vuelto que van de *0* a *M* unidades.

Moneda / Vuelto	*0*	*1*	*2*	*3*	*4*	…	*j*	…	*M*
moneda ***1: d1=…***									
moneda ***i: di=…***							***C(i,j)***		
…									
moneda ***n: dn=…***									

- En esa tabla ***C(i,j)*** será el número mínimo de monedas necesarias para pagar una cantidad de j unidades, con $0 \le j \le M$, con monedas de las denominaciones desde 1 hasta i, $1 \le i \le n$.

Problema: dar vuelto a un cliente

- La solución del problema estará en ***C(n,M)*** que tendrá la cantidad mínima de monedas para pagar un vuelto ***M*** con las ***n*** denominaciones de moneda.
- Para llenar la tabla comenzar por la primera columna que es $C(i,0)=0$ para todo i.

Moneda / Vuelto	0	1	2	3	4	…	*j*	…	*M*
moneda *1: d1=…*	0								
	0								
moneda *i: di=…*	0						*C(i,j)*		
…	0								
moneda *n: dn=…*	0								*C(n,M)*

- La tabla se puede llenar por fila de izquierda a derecha o por columna desde arriba hacia abajo.

Problema: dar vuelto a un cliente

Para dar una cantidad j usando monedas de valor entre 1 e i, se tienen dos alternativas:

1) No usar monedas de denominación i, entonces:

$$C(i,j)=C(i-1,j)$$

Moneda / Vuelto	0	1	2	3	4	…	j	…	M
moneda 1: $d1$=…									
moneda $i-1$: $di-1$=…							$C(i-1,j)$		
moneda i: di=…							$C(i,j)$		
…									
moneda n: dn=…									

Problema: dar vuelto a un cliente

La segunda alternativa para dar una cantidad j usando monedas de valor entre 1 e i, es:

2) <u>Usar al menos una moneda de denominación i</u>, en este caso se pagará di unidades con esa moneda y quedará pagar j-di unidades. Para eso se necesitan $C(i,j\text{-}di)$ unidades, así:

$$C(i,j)=1+C(i,j-di)$$

Moneda / Vuelto	**0**	**1**	***j-di***	**3**	**4**	**…**	***j***	**…**	***M***
moneda *1: d1=…*									
moneda *i: di=…*			***C(i,j-di)***		***+1***		***C(i,j)***		
…									
moneda *n: dn=…*									

Problema: dar vuelto a un cliente

Como se quiere minimizar el número de monedas utilizadas se opta entre la mejor de entre las las alternativas 1) y 2):

$$\mathbf{C(i,j) = Minimo\ (C(i\text{-}1,j)\ ,\ 1+C(i,j\text{-}di))}$$

Si $i=1$, el elemento $(i-1)$ está fuera de la tabla, en ese caso se supone que $C(i-1,j)$ tienen valor ∞.

Si $j<di$, en ese caso se supone que $C(i,j-di))$ tienen valor ∞.

Si $i=1$ y $j<d1$ entonces los dos elementos a comparar caen fuera de la tabla. En ese caso se asigna ∞ a $C(1,j)$ para indicar que es imposible pagar la cantidad j usando monedas de tipo 1.

Función DarVuelto (d,n,M)→entero≥0

Entrada: d: vector (1..n) cada d(i) es la denominación de la moneda i;
M: valor del vuelto que hay que pagar.
Salida: cantidad mínima de monedas necesarias para dar ese vuelto
Auxiliar: C(1..n,0..M) matriz de resultados parciales

```
    Para k=1,n hacer
        C(k,0)← 0          //1ª  columna de ceros
    Para i=1,n hacer
        Para j=1, M hacer
            Si i=1 entonces              // 1ª  fila
                Si j<d(i) entonces C(1,j)← ∞
                sino C(1,j)←1+ C(1,j-d(1))
            sino                          // (i>1) 2da fila en adelante
                Si j<d(i) entonces C(i,j)←C(i-1,j)        //no usa moneda d(i)
                sino C(i,j)←min (C(i-1,j) , 1+C(i,j-d(i)))  //elige minimo
            Fin si
        Fin Para j
    Fin Para i
    Retorna C(n,M)
Fin
```

No usar moneda i

Usar una moneda i

Problema: dar vuelto a un cliente

Ejemplo: n=3 monedas con valores de di=1, 4 y 6 unidades.

Se quiere dar vuelto de M=8 unidades con el número mínimo de monedas.

Tabla C(1..3,0..8)

Moneda / Vuelto	0	1	2	3	4	5	6	7	8
moneda 1: d1=1									
moneda 2: d2=4									
moneda 3: d3=6									

$$C(i,j) = \text{Minimo}(C(i-1,j),\ 1+C(i,j-di))$$

Problema: dar vuelto a un cliente

Ejemplo: n=3 monedas con valores de di=1, 4 y 6 unidades.

Se quiere dar vuelto de M=8 unidades con el número mínimo de monedas.

Tabla C(1..3,0..8)

Moneda / Vuelto	0	1	2	3	4	5	6	7	8
moneda 1: d1=1	0								
moneda 2: d2=4	0								
moneda 3: d3=6	0								

$$C(i,j) = \text{Minimo}(C(i-1,j),\ 1+C(i,j-di))$$

Problema: dar vuelto a un cliente

Ejemplo: n=3 monedas con valores de di=1, 4 y 6 unidades.

Se quiere dar vuelto de M=8 unidades con el número mínimo de monedas.

Tabla C(1..3,0..8)

Moneda / Vuelto	0	1	2	3	4	5	6	7	8
moneda 1: d1=1	0	1	2	3	4	5	6	7	8
moneda 2: d2=4	0								
moneda 3: d3=6	0								

C(i,j)= Minimo(C(i-1,j), 1+C(i,j-di))

Problema: dar vuelto a un cliente

Ejemplo: n=3 monedas con valores de di=1, 4 y 6 unidades.

Se quiere dar vuelto de M=8 unidades con el número mínimo de monedas.

Tabla C(1..3,0..8)

Moneda / Vuelto	0	1	2	3	4	5	6	7	8
moneda 1: d1=1	0	1	2	3	4	5	6	7	8
moneda 2: d2=4	0	1	2	3	1	2	3	4	2
moneda 3: d3=6	0								

$$C(i,j) = \text{Minimo}(C(i-1,j),\ 1+C(i,j-di))$$

Ej.

- C(2,4)=min (C(1,4), 1+C(2,4-4)) = min(4,1+0)=1

Problema: dar vuelto a un cliente

Ejemplo: n=3 monedas con valores de di=1, 4 y 6 unidades.

Se quiere dar vuelto de M=8 unidades con el número mínimo de monedas.

Tabla C(1..3,0..8)

Moneda / Vuelto	0	1	2	3	4	5	6	7	8
moneda 1: d1=1	0	1	2	3	4	5	6	7	8
moneda 2: d2=4	0	1	2	3	1	2	3	4	2
moneda 3: d3=6	0	1	2	3	1	2	1	2	2

$$C(i,j) = \text{Minimo}(C(i-1,j),\ 1+C(i,j-di))$$

Ej.

- C(3,8)=min (C(2,8), 1+C(3,8-6)) =min(2,1+2)=2

Problema: dar vuelto a un cliente

De la tabla construida se puede conocer *la cantidad de monedas que se necesitan para dar el vuelto.*

De la misma tabla también se puede determinar *cuales son esas monedas* mediante una estrategia greedy que avanza hacia arriba o hacia atrás en la tabla.

Se necesita dar vuelto por valor j, usando monedas de las denominaciones $1,2,\ldots,i$.

- La entrada $C(i,j)$ almacena cuantas monedas se va a necesitar.
- Si $C(i,j)=C(i\text{-}1,j)$ entonces no se usan monedas de la denominación y pasa a $C(i\text{-}1,j)$ para seguir analizando.
- Si $C(i,j)=1+C(i,j\text{-}di)$ entonces se usa 1 moneda de la denominación i que vale di y se avanza hasta $C(i,j\text{-}di)$ para seguir.
- Si $C(i\text{-}1,j)=1+ C(i,j\text{-}di)$ se puede elegir cualquiera de los dos caminos anteriores.

Así se puede llegar hasta $C(1,0)$ que no tiene nada que pagar.

Función MonedasDelVuelto (d,C,n,M)→vector

Entrada: d: vector (1..n) cada d(i) es la denominación de la moneda i;
C(1..n,0..M) matriz de resultados del algoritmo DarVuelto
M: valor del vuelto

Salida: L(1..n) vector con cant. de monedas de c/clase usadas en vuelto M

```
    i←n; j←M
    Para k=1,n hacer  L(k) ← 0
    Si C(n,M) < ∞ entonces
          Mientras i>0 and j>0 hacer
                    Si C(i,j)= C(i-1,j) entonces
                              i←i-1
                    sino
                              L(i)← L(i)+1
                              j←j-d(i)
                    Fin si
          Fin Mientras
    Fin si
    Retorna L
Fin
```

Problema: dar vuelto a un cliente

Costo del algoritmo

- Para dar vuelto a un cliente por valor de M unidades se prepara una tabla $C(1..n,0..M)$, esto es n filas y $M+1$ columnas.

- El algoritmo completa esta tabla por lo que tiene un costo asociado de:

$$T(n,M) \in O(n*M)$$

- Par obtener los valores de esas monedas la búsqueda retrocede de $C(n,M)$ hasta $C(1,0)$ dando n-1 pasos hacia arriba y $C(n,M)$ pasos a la izquierda.

- El tiempo total requerido para recuperar las monedas usadas es:

$$T(n,M) \in O(n+C(n,M)).$$

Problema: dar vuelto a un cliente

Ejercicio:

n=5 monedas con valores de di=1, 2, 5, 6, 8 unidades.

Se quiere dar vuelto de M=10 unidades con el número mínimo de monedas.

Tabla *C*(1..5,0..10)

Moneda / Vuelto	0	1	2	3	4	5	6	7	8	9	10
moneda 1: d1=1											
moneda 2: d2=2											
moneda 3: d3=5											
moneda 4: d3=6											
moneda 5: d3=8											

$$C(i,j) = min(C(i-1,j), 1+C(i,j-di))$$