INF 263 – Algoritmia

Estrategias Algorítmicas
Programación Dinámica

MAG. JOHAN BALDEON

MAG. RONY CUEVA

2021-1

Programación Dinámica

Es una técnica de diseño de algoritmos inventado por un prominente matemático de los Estados Unidos, Richard Bellman, en la década de 1950 como un método general para optimizar los procesos de decisión de múltiples etapas.

Por lo tanto, la palabra "programación" en el nombre de esta técnica significa "Planificación" y no se refiere a la programación de computadoras necesariamente. Después de probar su valor como una herramienta importante de las matemáticas aplicadas, la programación dinámica llega a ser considerada, en los círculos de la informática, como una técnica de diseño de algoritmo que no tiene que limitarse a tipos especiales de problemas de optimización.

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Programación Dinámica

A veces, la forma natural de dividir una instancia sugerida por la estructura del problema nos lleva a considerar varias **subinstancias superpuestas**

- Si se resuelve cada uno de estos en forma independiente, a su vez crearán un gran número de subinstancias idénticas
- Si no se presta atención a esta duplicación, es probable que se termine con un algoritmo ineficiente
- Si, por otro lado, se aprovecha de la duplicación y se resuelve cada subinstancia sólo una vez, salvando la solución para su uso posterior, resultará en un algoritmo más eficiente

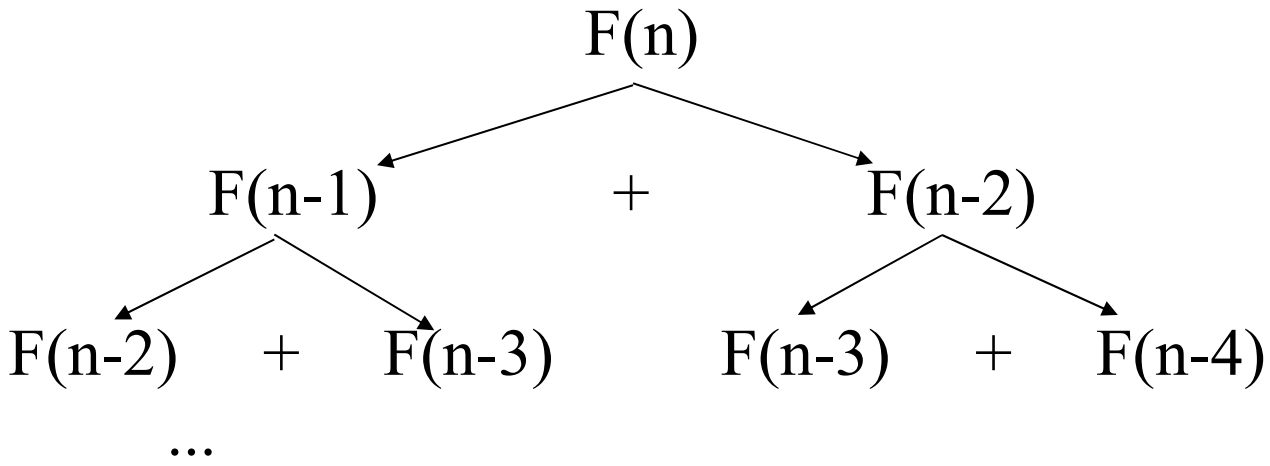
BRASSARD, G.; BRATLEY, P. **Algorithmics: theory and practice**. New Jersey: Prentice Hall, 1987. ISBN 0-13-023243-2.

Ejemplo 1: Números de Fibonacci

- Recordar la definición de los números de Fibonacci:

$$F(n) = F(n-1) + F(n-2)$$
$$F(0) = 0$$
$$F(1) = 1$$

- Calcular el n-ésimo número de Fibonacci recursivamente (top-down):



LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Ejemplo: Números de Fibonacci

- Calcular el n-ésimo número de Fibonacci usando iteraciones bottom-up y almacenando los resultados:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(2) = 1+0 = 1$$
$$\ldots$$
$$F(n-2) =$$
$$F(n-1) =$$
$$F(n) = F(n-1) + F(n-2)$$

0	**1**	**1**	**. . .**	***F*(*n*-2)**	***F*(*n*-1)**	***F*(*n*)**

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Programación Dinámica

La idea subyacente es **evitar el cálculo** de una misma instancia **dos veces**

Normalmente, se usa un **tabla de resultados** conocidos, que se llenan con las subinstancias resueltas.

BRASSARD, G.; BRATLEY, P. **Algorithmics: theory and practice**. New Jersey: Prentice Hall, 1987. ISBN 0-13-023243-2.

Programación Dinámica

La programación dinámica es una técnica ***bottom-up*** (de abajo hacia arriba)

- Se comienza con la subinstancia más pequeña:
  - La "más simple"
- Mediante la combinación de sus soluciones, se obtienen las respuestas a subinstancias de tamaño creciente, hasta que finalmente se llega a la solución de la instancia original.

BRASSARD, G.; BRATLEY, P. **Algorithmics: theory and practice**. New Jersey: Prentice Hall, 1987. ISBN 0-13-023243-2.

El principio de optimalidad

La programación dinámica se utiliza a menudo para resolver problemas de optimización que satisfacen el principio de **optimalidad**

- En una secuencia óptima de decisiones o elecciones, cada subsecuencia también debe ser óptima
- Si bien este principio puede parecer obvio, no siempre se aplica.

BRASSARD, G.; BRATLEY, P. **Algorithmics: theory and practice**. New Jersey: Prentice Hall, 1987. ISBN 0-13-023243-2.

El principio de optimalidad

El principio de optimalidad se puede definir de la siguiente manera

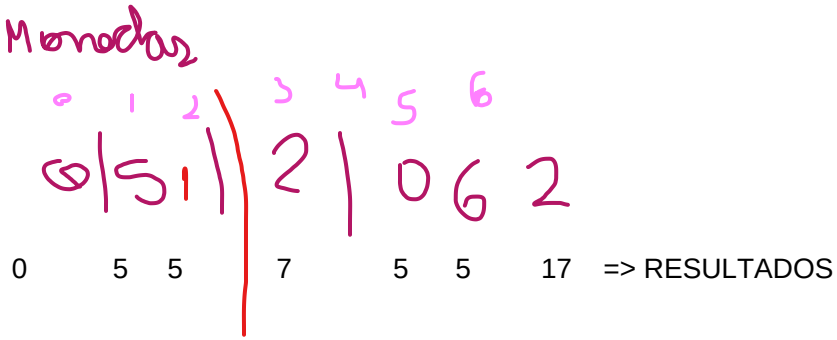
- La solución óptima para cualquier caso no trivial es una combinación de soluciones óptimas de algunas de sus subinstancias
- La dificultad en la transformación de este principio en un algoritmo es que no suele ser evidente cuales subinstancias son relevantes para el problema

BRASSARD, G.; BRATLEY, P. **Algorithmics: theory and practice**. New Jersey: Prentice Hall, 1987. ISBN 0-13-023243-2.

Ejemplo: Coin Row Problem

Existe una fila de n monedas cuyos valores son los enteros positivos $c_1, c_2, \ldots, c_n$, no necesariamente distintos. El objetivo es escoger la máxima cantidad de dinero considerando que no se pueden seleccionar al mismo tiempo dos monedas adyacentes en la fila inicial

Ejemplo: 5, 1, 2, 10, 6, 2. ¿Cuál es la mejor selección?



LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Ejemplo: Coin Row Problem

Sea F(n) la cantidad máxima que se puede obtener con `n` monedas.

Para derivar una recurrencia F(n), dividimos todas las selecciones permitidas de monedas en dos grupos:

- Aquellas con la última moneda
- Aquellas sin la última moneda

Entonces, tenemos la siguiente recurrencia:

$$F(n) = \max\{c_n + F(n-2),\ F(n-1)\} \text{ para } n > 1,$$

$$F(0) = 0,\ F(1)=c_1$$

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Ejemplo: Coin Row Problem

$F[0] = 0,\ F[1] = c_1 = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5					

$F[2] = \max\{1 + 0, 5\} = 5$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5				

$F[3] = \max\{2 + 5, 5\} = 7$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7			

$F[4] = \max\{10 + 5, 7\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15		

$F[5] = \max\{6 + 7, 15\} = 15$

index	0	1	2	3	4	5	6
C		5	1	2	10	6	2
F	0	5	5	7	15	15	

$F[6] = \max\{2 + 15, 15\} = 17$

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

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Ejemplo: Coin Row Problem

ALGORITHM *CoinRow*$(C[1..n])$

//Applies formula (8.3) bottom up to find the maximum amount of money
//that can be picked up from a coin row without picking two adjacent coins
//Input: Array $C[1..n]$ of positive integers indicating the coin values
//Output: The maximum amount of money that can be picked up
$F[0] \leftarrow 0; \quad F[1] \leftarrow C[1]$
for $i \leftarrow 2$ **to** n **do**
    $F[i] \leftarrow \max(C[i] + F[i-2],\ F[i-1])$
return $F[n]$

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Ejemplos Clásicos de Programación Dinámica

Longest Increasing Subsequence (LIS)

Dada una secuencia {X[0], X[1], …, X[n-1]}, determinar la **longitud** de la subsecuencia más larga de tal forma que todos los elementos de la subsecuencia estén ordenados en forma creciente

- La “subsecuencia” no es necesariamente continua

Ejemplo:

- Secuencia = {-7, 10, 9, 2, 3, 8, 8, 1}, N = 8
- LIS = {-7, 2, 3, 8}, **Longitud** = 4

HALIM, Steven and HALIM, Felix. **Competitive Programming**. 2da edición. 2011.

Longest Increasing Subsequence (LIS)

Sea LIS(i) la LIS que **termina en el índice `i`**

Índice	0	1	2	3	4	5	6	7
X	-7	10	9	2	3	8	8	1
LIS(i)	1	2	2	2	3	4	4	2

- LIS (0) = 1: {-7}
- LIS (1) = 2: {-7, 10}
- LIS (2) = 2: {-7, 9}
- LIS (3) = 2: {-7, 2}
- LIS (4) = 3: {-7, 2, 3}
- LIS (5) = 4: {-7, 2, 3, 8}
- LIS (6) = 4: {-7, 2, 3, 8}
- LIS (7) = 2: {-7, 1}

HALIM, Steven and HALIM, Felix. **Competitive Programming**. 2da edición. 2011.

Longest Increasing Subsequence (LIS)

Sea L(i) la longitud de la LIS que termina en i:

$$L(i) \begin{cases} 1 + \max_{j=0,\ldots,i-1}\{L(j) \mid X_j < X_i\}, & i > 0 \\ 1 & i = 0 \end{cases}$$

Lo anterior permite obtener la longitud de la subsecuencia, ¿cómo se obtiene los elementos que la conforman?

- Usar una estructura que permita almacenar el índice de la subsecuencia `L(j)` a la que se agregó `i`

Knapsack Problem

Dados n elementos de

pesos enteros: w_1 w_2 … w_n

valores: v_1 v_2 … v_n

y una mochila de capacidad entera W.

Encontrar el subconjunto más valioso de elementos que caben en la mochila.

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Knapsack Problem

Consideremos la instancia definida por los primeros *i* elementos y capacidad *j* ($j \leq W$)

Sea $V[i,j]$ el valor óptimo de dicha instancia. Luego:

$$V[i,j] \begin{cases} \max \{V[i\text{-}1, j], \boldsymbol{v}_i + V[i\text{-}1, j\text{-}\boldsymbol{w}_i]\} & , j\text{-}\boldsymbol{w}_i \geq 0 \\ V[i\text{-}1, j] & , j\text{-}\boldsymbol{w}_i < 0 \end{cases}$$

Condiciones iniciales: $V[0, j] = 0$ y $V[i, 0] = 0$

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Ejemplo: Knapsack Problem

Consideremos la instancia dada por los siguientes datos:

Elemento	Peso (w)	Valor (v)
1	2	$12
2	1	$10
3	3	$20
4	2	$15

Capacidad W = 5

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

Ejemplo: Knapsack Problem

$$V[i,j] \begin{cases} \max\{V[i-1, j], \boldsymbol{V}_i + V[i-1, j-\boldsymbol{W}_i]\} & , j-\boldsymbol{W}_i \geq 0 \\ V[i-1, j] & , \quad j-\boldsymbol{W}_i < 0 \end{cases}$$

Condiciones iniciales: $V[0, j] = 0$ y $V[i, 0] = 0$

3-1 = 2
reviso la columna ant

12 peso ant
10+0

lo bajo a 1 de capacidad
y como en esa columna es cero
le agrego el V2 => 10 + 0;

Capacidad *j*

Elementos *i*			0	1	2	3	4	5
		0	0	0	0	0	0	0
$w_1 = 2$	$v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1$	$v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3$	$v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2$	$v_4 = 15$	4	0	10	15	25	30	**37**

no puede llegar
a entrar, porque pasa
la capacidad

22
3-3= 0
+20
22

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN-13 978-0-13-231681-1

y lo evaluo con la que realmente esta presente
=> siempre resto la capacidad j, con la capacidad del i, para saber a cual ataco

22
3-2 = 1(miro esta col)
-> 10+15 =25

25 es mayor!

Resumen

La **programación dinámica** es una estrategia para resolver problemas con subproblemas que se sobreponen

Típicamente, estos subproblemas surgen de una **recurrencia** que relaciona la solución a un problema dado con soluciones a sus subproblemas (que son del mismo tipo)

Se sugiere resolver cada subproblema de menor tamaño una vez y **grabar los resultados en una tabla** de la que se puede obtener la solución al problema original

Ejemplo Extra: Sub-secuencia común más larga

- Problema común en comparación de genes
  - Dadas dos secuencias

$$S_1 = \text{ACCGGTCGAGTGCGCGGAAGCCGGCCGAA}$$

$$S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$$

encontrar la sub-secuencia común más larga, con caracteres no necesariamente consecutivos

  - Para el ejemplo, la solución es

$$\text{GTCGTCGGAAGCCGGCCGAA}$$

Bibliografía Recomendada

LEVITIN, A. **Introduction to The Design and Analysis of Algorithms**. 3ra edición. USA: Pearson, 2012. ISBN 0-13-231681-1.

- Cap8: Dynamic Programming
  - 8.1 Three Basic Examples
  - 8.2 The Knapsack Problem
    - No considerar la sección de Memory Functions

HALIM, Steven and HALIM, Felix. **Competitive Programming**. 2da edición. 2011.

- Cap3: Problem Solving Paradigms
  - 3.4 Dynamic Programming

Más ejemplos

- Variación: Mochila que permite repetición de elementos escogidos
- $C(w)$ = valor máximo logrado en una mochila de capacidad w

$$C(i) = \begin{cases} \max_j \{v_j + C(i - w_j) : i - w_j \geq 0\} \\ 0 \ \ si \ \ i = 0 \end{cases}$$

Examen 1 – 2016-1 (Pregunta 2)!

Más ejemplos

- Problema de cambio de monedas
- Se tiene una cantidad N y se tiene un conjunto de monedas de denominación $a_1, a_2, \ldots, a_m$. De cuántas formas podemos cambiar la cantidad N con las monedas dadas?
- Definimos $C(i, j)$ = cantidad de formas de cambiar la cantidad i con las monedas de denominación $a_1, a_2, \ldots, a_j$.
- El óptimo será $C(N, m)$

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

- Casos bases
  - $C(0, j) = 1$
  - $C(i, 0) = 0$

Lab3 – 2016-1 (Pregunta 2)!

Ejemplo Extra: Sub-secuencia común más larga

- Ejemplo
  - S_1 = B D C A B A y
  - S_2 = A B C B D A B