DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Algoritmos y Estructuras de Datos III

Segundo cuatrimestre 2021 *(dictado a distancia)*

Técnicas de diseño de algoritmos

Técnicas de diseño de algoritmos

- Fuerza bruta
- Búsqueda con retroceso (*backtracking*)
- Algoritmos golosos
- Recursividad
- *Dividir y conquistar* (*divide and conquer*)
- Programación dinámica
- Heurísticas y algoritmos aproximados

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso *la mejor alternativa posible localmente* según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso *la mejor alternativa posible localmente* según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso *la mejor alternativa posible localmente* según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- En cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras.

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso *la mejor alternativa posible localmente* según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- En cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras.
- Una decisión tomada nunca es revisada y no se evaluan alternativas.

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso *la mejor alternativa posible localmente* según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- En cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras.
- Una decisión tomada nunca es revisada y no se evaluan alternativas.
- Fáciles de inventar e implementar y generalmente eficientes.

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso *la mejor alternativa posible localmente* según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- En cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras.
- Una decisión tomada nunca es revisada y no se evaluan alternativas.
- Fáciles de inventar e implementar y generalmente eficientes.
- Pero no siempre *funcionan*: algunos problemas no pueden ser resueltos por este enfoque.

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso *la mejor alternativa posible localmente* según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- En cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras.
- Una decisión tomada nunca es revisada y no se evaluan alternativas.
- Fáciles de inventar e implementar y generalmente eficientes.
- Pero no siempre *funcionan*: algunos problemas no pueden ser resueltos por este enfoque.
  - Habitualmente, proporcionan *heurísticas* sencillas para *problemas de optimización*.
  - En general permiten construir soluciones razonables, pero sub-óptimas.

Algoritmos golosos

Idea: Construir una solución seleccionando en cada paso *la mejor alternativa posible localmente* según una función de selección, sin considerar (o haciéndolo débilmente) las implicancias de esta selección.

- En cada etapa se toma la decisión que parece mejor basándose en la información disponible en ese momento, sin tener en cuenta las consecuencias futuras.
- Una decisión tomada nunca es revisada y no se evaluan alternativas.
- Fáciles de inventar e implementar y generalmente eficientes.
- Pero no siempre *funcionan*: algunos problemas no pueden ser resueltos por este enfoque.
  - Habitualmente, proporcionan *heurísticas* sencillas para *problemas de optimización*.
  - En general permiten construir soluciones razonables, pero sub-óptimas.
- Función de selección.

Algoritmos golosos - El problema de la mochila (*continuo*)

Datos de entrada:

- Capacidad $C \in \mathbb{R}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{N}$ de objetos.
- Peso $p_i \in \mathbb{R}_+$ del objeto i, para $i = 1, \ldots, n$.
- Beneficio $b_i \in \mathbb{R}_+$ del objeto i, para $i = 1, \ldots, n$.

Algoritmos golosos - El problema de la mochila (*continuo*)

Datos de entrada:

- Capacidad $C \in \mathbb{R}_+$ de la mochila (peso máximo).
- Cantidad $n \in \mathbb{N}$ de objetos.
- Peso $p_i \in \mathbb{R}_+$ del objeto i, para $i = 1, \dots, n$.
- Beneficio $b_i \in \mathbb{R}_+$ del objeto i, para $i = 1, \dots, n$.

Problema: Determinar qué objetos debemos incluir en la mochila sin excedernos del peso máximo C, de modo tal de *maximizar* el beneficio total entre los objetos seleccionados. En la versión más simple de este problema vamos a suponer que podemos poner *parte* de un objeto en la mochila (*continuo*).

Algoritmos golosos - El problema de la mochila (*continuo*)

Algoritmo(s) goloso(s): Mientras no se haya excedido el peso de la mochila, agregar a la mochila el objeto i que ...

Algoritmos golosos - El problema de la mochila (*continuo*)

Algoritmo(s) goloso(s): Mientras no se haya excedido el peso de la mochila, agregar a la mochila el objeto i que ...

- ... tenga mayor beneficio b_i

Algoritmos golosos - El problema de la mochila (*continuo*)

Algoritmo(s) goloso(s): Mientras no se haya excedido el peso de la mochila, agregar a la mochila el objeto i que ...

- ... tenga mayor beneficio b_i
- ... tenga menor peso p_i

Algoritmos golosos - El problema de la mochila (*continuo*)

Algoritmo(s) goloso(s): Mientras no se haya excedido el peso de la mochila, agregar a la mochila el objeto i que ...

- ... tenga mayor beneficio b_i
- ... tenga menor peso p_i
- ... tenga mayor beneficio por unidad de peso, b_i/p_i

Algoritmos golosos - El problema de la mochila (*continuo*)

Datos de entrada:

$$C = 100,\ n = 5$$

	1	2	3	4	5
p	10	20	30	40	50
b	20	30	66	40	60
b/p	2.0	1.5	2.2	1.0	1.2

Algoritmos golosos - El problema de la mochila (*continuo*)

Datos de entrada:

$$C = 100, \; n = 5$$

	1	2	3	4	5
p	10	20	30	40	50
b	20	30	66	40	60
b/p	2.0	1.5	2.2	1.0	1.2

- mayor beneficio b_i: $66 + 60 + 40/2 = 146$.
- menor peso p_i: $20 + 30 + 66 + 40 = 156$.
- maximice b_i/p_i: $66 + 20 + 30 + 0{,}8 \cdot 60 = 164$.

Algoritmos golosos - El problema de la mochila (*continuo*)

- ¿Qué podemos decir en cuanto a la *calidad* de las soluciones obtenidas por estos algoritmos?

Algoritmos golosos - El problema de la mochila (*continuo*)

- ¿Qué podemos decir en cuanto a la *calidad* de las soluciones obtenidas por estos algoritmos?

- Se puede demostrar que la selección según máximo beneficio por unidad de peso, b_i/p_i, da una solución óptima.

Algoritmos golosos - El problema de la mochila (*continuo*)

- ¿Qué podemos decir en cuanto a la *calidad* de las soluciones obtenidas por estos algoritmos?
- Se puede demostrar que la selección según máximo beneficio por unidad de peso, b_i/p_i, da una solución óptima.
- ¿Qué podemos decir en cuanto a su *complejidad*?

Algoritmos golosos - El problema de la mochila (*continuo*)

- ¿Qué podemos decir en cuanto a la *calidad* de las soluciones obtenidas por estos algoritmos?
- Se puede demostrar que la selección según máximo beneficio por unidad de peso, b_i/p_i, da una solución óptima.
- ¿Qué podemos decir en cuanto a su *complejidad*?
- ¿Qué sucede si los elementos se deben poner enteros en la mochila?

Algoritmos golosos - El problema del cambio

Problema: Supongamos que queremos dar el vuelto a un cliente usando el mínimo número de monedas posibles, utilizando monedas de 1, 5, 10 y 25 centavos. Por ejemplo, si el monto es \$0,69, deberemos entregar 8 monedas: 2 monedas de 25 centavos, una de 10 centavos, una de 5 centavos y cuatro de un centavo.

Algoritmos golosos - El problema del cambio

Problema: Supongamos que queremos dar el vuelto a un cliente usando el mínimo número de monedas posibles, utilizando monedas de 1, 5, 10 y 25 centavos. Por ejemplo, si el monto es \$0,69, deberemos entregar 8 monedas: 2 monedas de 25 centavos, una de 10 centavos, una de 5 centavos y cuatro de un centavo.

Algoritmo goloso: Seleccionar la moneda de mayor valor que no exceda la cantidad restante por devolver, agregar esta moneda a la lista de la solución, y sustraer la cantidad correspondiente a la cantidad que resta por devolver (hasta que sea 0).

Algoritmos golosos - El problema del cambio

```
darCambio(cambio)
    entrada: cambio ∈ ℕ
    salida: M conjunto de enteros

    suma ← 0
    M ← {}
    mientras suma < cambio hacer
        proxima ← masgrande(cambio,suma)
        M ← M ∪ {proxima}
        suma ← suma + proxima
    fin mientras
    retornar M
```

Algoritmos golosos - El problema del cambio

- Este algoritmo siempre produce la mejor solución *para estos valores de monedas*, es decir, retorna la menor cantidad de monedas necesarias para obtener el valor *cambio*.

Algoritmos golosos - El problema del cambio

- Este algoritmo siempre produce la mejor solución *para estos valores de monedas*, es decir, retorna la menor cantidad de monedas necesarias para obtener el valor *cambio*.

- Sin embargo, si también hay monedas de 12 centavos, puede ocurrir que el algoritmo no encuentre una solución óptima: si queremos devolver 21 centavos, el algoritmo retornará una solución con 6 monedas, una de 12 centavos, 1 de 5 centavos y cuatro de 1 centavos, mientras que la solución óptima es retornar dos monedas de 10 centavos y una de 1 centavo.

Algoritmos golosos - El problema del cambio

- Este algoritmo siempre produce la mejor solución *para estos valores de monedas*, es decir, retorna la menor cantidad de monedas necesarias para obtener el valor *cambio*.

- Sin embargo, si también hay monedas de 12 centavos, puede ocurrir que el algoritmo no encuentre una solución óptima: si queremos devolver 21 centavos, el algoritmo retornará una solución con 6 monedas, una de 12 centavos, 1 de 5 centavos y cuatro de 1 centavos, mientras que la solución óptima es retornar dos monedas de 10 centavos y una de 1 centavo.

- El algoritmo es goloso porque en cada paso selecciona la moneda de mayor valor posible, sin preocuparse que esto puede llevar a una mala solución, y nunca modifica una decisión tomada.

Algoritmos golosos - Tiempo de espera total en un sistema

Problema: Un servidor tiene n clientes para atender, y los puede atender en cualquier orden. Para $i = 1, \ldots, n$, el tiempo necesario para atender al cliente i es $t_i \in \mathbb{R}_+$. El objetivo es determinar en qué orden se deben atender los clientes para minimizar *la suma de los tiempos de espera* de los clientes.

Algoritmos golosos - Tiempo de espera total en un sistema

Problema: Un servidor tiene n clientes para atender, y los puede atender en cualquier orden. Para $i = 1, \ldots, n$, el tiempo necesario para atender al cliente i es $t_i \in \mathbb{R}_+$. El objetivo es determinar en qué orden se deben atender los clientes para minimizar *la suma de los tiempos de espera* de los clientes.

Si $I = (i_1, i_2, \ldots, i_n)$ es una permutación de los clientes que representa el orden de atención, entonces la suma de los tiempos de espera es

$$\begin{aligned} T &= t_{i_1} + (t_{i_1} + t_{i_2}) + (t_{i_1} + t_{i_2} + t_{i_3}) + \ldots \\ &= \sum_{k=1}^{n} (n - k + 1) t_{i_k}. \end{aligned}$$

Algoritmos golosos - Tiempo de espera total en un sistema

Algoritmo goloso: En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- Retorna una permutación $I_{\text{GOL}} = (i_1, \ldots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \ldots, n-1$.

Algoritmos golosos - Tiempo de espera total en un sistema

Algoritmo goloso: En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- Retorna una permutación $I_{\text{GOL}} = (i_1, \ldots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \ldots, n-1$.
- ¿Cuál es la *complejidad* de este algoritmo?

Algoritmos golosos - Tiempo de espera total en un sistema

Algoritmo goloso: En cada paso, atender al cliente pendiente que tenga menor tiempo de atención.

- Retorna una permutación $I_{\text{GOL}} = (i_1, \ldots, i_n)$ tal que $t_{i_j} \leq t_{i_{j+1}}$ para $j = 1, \ldots, n-1$.
- ¿Cuál es la *complejidad* de este algoritmo?
- Este algoritmo proporciona la *solución óptima*!

Programación Dinámica

Idea: Al igual que en D&C, el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

Programación Dinámica

Idea: Al igual que en D&C, el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

- Es aplicada típicamente a problemas de optimización combinatoria.

Programación Dinámica

Idea: Al igual que en D&C, el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

- Es aplicada típicamente a problemas de optimización combinatoria.
- También resulta adecuada para algunos problemas de naturaleza recursiva.

Programación Dinámica

Idea: Al igual que en D&C, el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

- Es aplicada típicamente a problemas de optimización combinatoria.
- También resulta adecuada para algunos problemas de naturaleza recursiva.
- Es adecuada para problemas que tienen las característas que le molestan a D&C.

Programación Dinámica

Idea: Al igual que en D&C, el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

- Es aplicada típicamente a problemas de optimización combinatoria.
- También resulta adecuada para algunos problemas de naturaleza recursiva.
- Es adecuada para problemas que tienen las característas que le molestan a D&C.
- **Evita repetir llamadas recursivas almacenando los resultados calculados.**

Programación Dinámica

Idea: Al igual que en D&C, el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

- Es aplicada típicamente a problemas de optimización combinatoria.
- También resulta adecuada para algunos problemas de naturaleza recursiva.
- Es adecuada para problemas que tienen las característas que le molestan a D&C.
- **Evita repetir llamadas recursivas almacenando los resultados calculados.**
- Esquema de *memoización*.

Programación Dinámica

Idea: Al igual que en D&C, el problema es dividido en subproblemas de tamaños menores que son más fáciles de resolver. Una vez resueltos estos subproblemas, se combinan las soluciones obtenidas para generar la solución del problema original.

- Es aplicada típicamente a problemas de optimización combinatoria.
- También resulta adecuada para algunos problemas de naturaleza recursiva.
- Es adecuada para problemas que tienen las característas que le molestan a D&C.
- **Evita repetir llamadas recursivas almacenando los resultados calculados.**
- Esquema de *memoización*.
- Funciones recursivas top-down, pero algunas veces una implementación no recursiva bottom-up puede llegar a resultar en algoritmos con mejor tiempo de ejecución.

Programación Dinámica - Coeficientes binomiales

- **Definición:** Si $n \geq 0$ y $0 \leq k \leq n$, se define

$$\binom{n}{k} = \frac{n!}{k!(n-k)!}$$

- **Teorema:** Si $n \geq 0$ y $0 \leq k \leq n$, entonces

$$\binom{n}{k} = \begin{cases} 1 & \text{si } k = 0 \text{ o } k = n \\ \binom{n-1}{k-1} + \binom{n-1}{k} & \text{si } 0 < k < n \\ 0 & \text{caso contrario} \end{cases}$$

Programación Dinámica - Coeficientes binomiales

Algoritmo recursivo

combiRec(n, k)

entrada: $n, k \in N$

salida: $\binom{n}{k}$

si $k = 0$ **o** $k = n$ **hacer**

```
    retornar 1
```

else

```
    retornar combiRec(n-1, k-1) + combiRec(n-1, k)
```

fin si

Programación Dinámica - Coeficientes binomiales

Algoritmo recursivo con memoización (top-down)

```
combiRec(n,k,T)
```
entrada: $n, k \in N$
salida: $\binom{n}{k}$

si $T[n][k] = NULL$ **hacer**
  si $k = 0$ **o** $k = n$ **hacer**
    $T[n][k] \leftarrow 1$
  else
    $T[n][k] \leftarrow$ `combiRec(`$n-1, k-1, T$`)` + `combiRec(`$n-1, k, T$`)`
  fin si
fin si
`retornar` $T[n][k]$

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0								
1								
2								
3								
4								
$\vdots$								
$k-1$								
k								
$\vdots$								
$n-1$								
n								

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1		1					
3	1			1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1		1					
3	1			1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1			1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1			1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3		1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3		1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1				1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4			1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4			1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6		1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6		1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

$n \backslash k$	0	1	2	3	4	$\cdots$	$k-1$	k
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
$\vdots$	$\vdots$					$\ddots$		
$k-1$	1						1	
k	1							1
$\vdots$	$\vdots$							
$n-1$	1							
n	1							

Programación Dinámica - Coeficientes binomiales

algoritmo $\mathit{combinatorio}(n,k)$

  entrada: dos enteros n y k

  salida: $\binom{n}{k}$

  para $i = 1$ **hasta** n **hacer**

    $A[i][0] \leftarrow 1$

  fin para

  para $j = 0$ **hasta** k **hacer**

    $A[j][j] \leftarrow 1$

  fin para

  para $i = 2$ **hasta** n **hacer**

    para $j = 2$ **hasta** $\mathtt{min}(i-1,\mathtt{k})$ **hacer**

      $A[i][j] \leftarrow A[i-1][j-1] + A[i-1][j]$

    fin para

  fin para

  retornar $A[n][k]$

Programación Dinámica - Multiplicación de n matrices

Problema: Por la propiedad asociativa del producto de matrices

$$M = M_1 \times M_2 \times \dots M_n$$

puede hacerse de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias.

Programación Dinámica - Multiplicación de n matrices

Problema: Por la propiedad asociativa del producto de matrices

$$M = M_1 \times M_2 \times \ldots M_n$$

puede hacerse de muchas formas. Queremos determinar la que minimiza el número de operaciones necesarias.

Si las dimensión de A es de 13×5, la de B de 5×89, la de C de 89×3 y la de D de 3×34, tenemos

- $((AB)C)D$ requiere 10582 multiplicaciones.
- $(AB)(CD)$ requiere 54201 multiplicaciones.
- $(A(BC))D$ requiere 2856 multiplicaciones.
- $A((BC)D)$ requiere 4055 multiplicaciones.
- $A(B(CD))$ requiere 26418 multiplicaciones.

Programación Dinámica - Multiplicación de n matrices

- Para multiplicar todas las matrices de forma óptima, deberemos multiplicar las matrices 1 a i por un lado y las matrices $i+1$ a n por otro lado y luego multiplicar estos dos resultados, para algún $1 \leq i \leq n-1$, que es justamente lo que queremos determinar.

Programación Dinámica - Multiplicación de n matrices

- Para multiplicar todas las matrices de forma óptima, deberemos multiplicar las matrices 1 a i por un lado y las matrices $i+1$ a n por otro lado y luego multiplicar estos dos resultados, para algún $1 \leq i \leq n-1$, que es justamente lo que queremos determinar.

- Estos dos subproblemas, $M_1 \times M_2 \times \ldots M_i$ y $M_{i+1} \times M_{i+2} \times \ldots M_n$ deben estar resueltos, a su vez, de forma óptima, es decir realizando la mínima cantidad de operaciones.

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \ldots M_j$.
Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

Programación Dinámica - Multiplicación de n matrices

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \dots M_j$.
Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \dots, n$, $m[i][i] =$

Programación Dinámica - Multiplicación de n matrices

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \ldots M_j$.
Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \ldots, n$, $m[i][i] = 0$

Programación Dinámica - Multiplicación de n matrices

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \ldots M_j$.
Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \ldots, n$, $m[i][i] = 0$
- Para $i = 1, 2, \ldots, n-1$, $m[i][i+1] =$

Programación Dinámica - Multiplicación de n matrices

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \ldots M_j$.
Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \ldots, n$, $m[i][i] = 0$
- Para $i = 1, 2, \ldots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$

Programación Dinámica - Multiplicación de n matrices

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \dots M_j$. Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \dots, n$, $m[i][i] = 0$
- Para $i = 1, 2, \dots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- Para $s = 2, \dots, n-1$, $i = 1, 2, \dots, n-s$,

  $m[i][i+s] =$

Programación Dinámica - Multiplicación de n matrices

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \dots M_j$. Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \dots, n$, $m[i][i] = 0$
- Para $i = 1, 2, \dots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- Para $s = 2, \dots, n-1$, $i = 1, 2, \dots, n-s$,

  $$m[i][i+s] = \min_{i \leq k < i+s} (m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s])$$

Programación Dinámica - Multiplicación de n matrices

Llamamos $m[i][j]$ solución del subproblema $M_i \times M_{i+1} \times \ldots M_j$. Suponemos que las dimensiones de las matrices están dadas por un vector $d \in N^{n+1}$, tal que la matriz M_i tiene $d[i-1]$ filas y $d[i]$ columnas para $1 \leq i \leq n$. Entonces:

- Para $i = 1, 2, \ldots, n$, $m[i][i] = 0$
- Para $i = 1, 2, \ldots, n-1$, $m[i][i+1] = d[i-1]d[i]d[i+1]$
- Para $s = 2, \ldots, n-1$, $i = 1, 2, \ldots, n-s$,

  $$m[i][i+s] = \min_{i \leq k < i+s} (m[i][k] + m[k+1][i+s] + d[i-1]d[k]d[i+s])$$

La solución del problema es $m[1][n]$.

Programación Dinámica - Subsecuencia común más larga

Dada una secuencia, una subsecuencia se obtiene eliminando 0 o más símbolos. Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $[4, 7, 8, 2, 5, 3]$, $[2, 7]$ no lo es.

Programación Dinámica - Subsecuencia común más larga

Dada una secuencia, una subsecuencia se obtiene eliminando 0 o más símbolos. Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $[4, 7, 8, 2, 5, 3]$, $[2, 7]$ no lo es.

Problema: Encontrar la subsecuencia común mas larga (scml) de dos secuencias dadas.

Programación Dinámica - Subsecuencia común más larga

Dada una secuencia, una subsecuencia se obtiene eliminando 0 o más símbolos. Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $[4, 7, 8, 2, 5, 3]$, $[2, 7]$ no lo es.

Problema: Encontrar la subsecuencia común mas larga (scml) de dos secuencias dadas.

- Es decir, dadas dos secuencias A y B, queremos encontrar entre todas secuencia que son tanto subsecuencia de A como de B la de mayor longitud.

Programación Dinámica - Subsecuencia común más larga

Dada una secuencia, una subsecuencia se obtiene eliminando 0 o más símbolos. Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $[4, 7, 8, 2, 5, 3]$, $[2, 7]$ no lo es.

Problema: Encontrar la subsecuencia común mas larga (scml) de dos secuencias dadas.

- Es decir, dadas dos secuencias A y B, queremos encontrar entre todas secuencia que son tanto subsecuencia de A como de B la de mayor longitud.
- Por ejemplo, si $A = [9, 5, 2, 8, 7, 3, 1, 6, 4]$ y $B = [2, 9, 3, 5, 8, 7, 4, 1, 6]$ las scml es $[9, 5, 8, 7, 1, 6]$.

Programación Dinámica - Subsecuencia común más larga

Dada una secuencia, una subsecuencia se obtiene eliminando 0 o más símbolos. Por ejemplo, $[4, 7, 2, 3]$ y $[7, 5]$ son subsecuencias de $[4, 7, 8, 2, 5, 3]$, $[2, 7]$ no lo es.

Problema: Encontrar la subsecuencia común mas larga (scml) de dos secuencias dadas.

- Es decir, dadas dos secuencias A y B, queremos encontrar entre todas secuencia que son tanto subsecuencia de A como de B la de mayor longitud.
- Por ejemplo, si $A = [9, 5, 2, 8, 7, 3, 1, 6, 4]$ y $B = [2, 9, 3, 5, 8, 7, 4, 1, 6]$ las scml es $[9, 5, 8, 7, 1, 6]$.
- Si resolvemos este problema por fuerza bruta, listaríamos todas las subsecuencias de S_1, todas las de S_2, nos fijaríamos cuales tienen en común, y entre esas elegiríamos la más larga.

Programación Dinámica - Subsecuencia común más larga

Dadas las dos secuencias $A = [a_1, \ldots, a_r]$ y $B = [b_1, \ldots, b_s]$, existen dos posibilidades, $a_r = b_s$ o $a_r \neq b_s$. Analicemos cada caso:

Programación Dinámica - Subsecuencia común más larga

Dadas las dos secuencias $A = [a_1, \ldots, a_r]$ y $B = [b_1, \ldots, b_s]$, existen dos posibilidades, $a_r = b_s$ o $a_r \neq b_s$. Analicemos cada caso:

1. $a_r = b_s$: La scml entre A y B se obitene colocando al final de la scml entre $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_{s-1}]$ al elemento a a_r (o b_s porque son iguales).

Programación Dinámica - Subsecuencia común más larga

Dadas las dos secuencias $A = [a_1, \ldots, a_r]$ y $B = [b_1, \ldots, b_s]$, existen dos posibilidades, $a_r = b_s$ o $a_r \neq b_s$. Analicemos cada caso:

1. *$a_r = b_s$:* La scml entre A y B se obitene colocando al final de la scml entre $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_{s-1}]$ al elemento a a_r (o b_s porque son iguales).

2. *$a_r \neq b_s$:* La scml entre A y B será la más larga entre las scml entre $[a_1., \ldots, a_{r-1}]$ y la scml entre $[b_1., \ldots, b_s]$ y $[a_1., \ldots, a_r]$ y $[b_1., \ldots, b_{s-1}]$.
   Esto es, calculamos el problema aplicado a $[a_1., \ldots, a_{r-1}]$ y $[b_1., \ldots, b_s]$ y, por otro lado, el problema aplicado a $[a_1., \ldots, a_r]$ y $[b_1., \ldots, b_{s-1}]$, y nos quedamos con la más larga de ambas.

Programación Dinámica - Subsecuencia común más larga

Esta forma recursiva de resolver el problema ya nos conduce al algoritmo.

Esta forma recursiva de resolver el problema ya nos conduce al algoritmo.
Si llamamos $l[i][j]$ a la longitud de la scml entre $[a_1., \ldots, a_i]$ y $[b_1., \ldots, b_j]$, entonces:

Programación Dinámica - Subsecuencia común más larga

Esta forma recursiva de resolver el problema ya nos conduce al algoritmo.
Si llamamos $l[i][j]$ a la longitud de la scml entre $[a_1., \dots, a_i]$ y $[b_1., \dots, b_j]$, entonces:

- $l[0][0] = 0$
- Para $j = 1, \dots, s$, $l[0][j] = 0$
- Para $i = 1, 2, \dots, r$, $l[i][0] = 0$
- Para $i = 1, \dots, r$, $j = 1, \dots, s$
  - si $a_i = b_j$: $l[i][j] = l[i-1][j-1] + 1$
  - si $a_i \neq b_j$: $l[i][j] = \text{máx}\{l[i-1][j], l[i][j-1]\}$

Y la solución del problema será $l[r][s]$.

Programación Dinámica - Subsecuencia común más larga

```
scml(A,B)
   entrada: A,B secuencias
   salida: longitud de a scml entre A y B

   l[0][0] ← 0
   para i = 1 hasta r hacer l[i][0] ← 0
   para j = 1 hasta s hacer l[0][j] ← 0
   para i = 1 hasta r hacer
         para j = 1 hasta s hacer
               si A[i] = B[j]
                     l[i][j] ← l[i − 1][j − 1] + 1
               sino
                     l[i][j] ← máx{l[i − 1][j], l[i][j − 1]}
               fin si
         fin para
   fin para
   retornar l[r][s]
```

Heurísticas

- Dado un problema Π , un algoritmo heurístico es un algoritmo que intenta obtener soluciones de buena calidad para el problema que se quiere resolver pero no necesariamente lo hace en todos los casos.
- Sea Π un problema de optimización, I una instancia del problema, $x^*(I)$ el valor óptimo de la función a optimizar en dicha instancia. Un algoritmo heurístico obtiene una solución con un valor que se espera sea cercano a ese óptimo pero no necesariamente el óptimo.
- Si H es un algoritmo heurístico para un problema de optimización llamamos $x^H(I)$ al valor que devuelve la heurística.

Algoritmos aproximados

Decimos que H es un algoritmo ϵ – aproximado para el problema Π si para algún $\epsilon > 0$

$$|x^H(I) - x^*(I)| \leq \epsilon |x^*(I)|$$