EX 2018-2

4. Programación dinámica

En clases estudiamos el problema de programar el mayor número de tareas en una misma máquina, en que cada tarea k tiene una hora de inicio s_k y una hora de finalización f_k, y la máquina solo puede ejecutar una tarea a la vez. Vimos que el problema se puede resolver usando un algoritmo codicioso en tiempo O(n), si hay n tareas en total y vienen ordenadas por sus horas de finalización.

Considera ahora que cada tarea tiene, además, un valor v_k, y queremos maximizar la suma de los valores de las tareas realizadas (y no el número de tareas realizadas).

a) Muestra que el algoritmo codicioso anterior no resuelve esta versión más general del problema.

Bssta un ejemplo:

k: s_k–f_k, v_k
1: 3–10, 1
2: 5–15, 3
3: 12–20, 1

En este caso, el algoritmo codicioso elige las tareas 1 y 3, con un valor total de 1+1 = 2; pero la elección de la tarea 2 habría producido un valor de 3.

Muestra que el problema se puede resolver mediante programación dinámica. En particular, suponemos nuevamente que las tareas vienen ordenadas por sus horas de finalización: $f_1 \leq f_2 \leq \ldots \leq f_n$. Entonces, para cada tarea j, definimos b[j] como la tarea g que termina más tarde antes del inicio de j; b[j] = 0 si ninguna tarea satisface esta condición.

Sea T es una solución óptima al problema. Revisamos las tareas "de atrás hacia adelante": obviamente, la tarea n pertenece a T, o bien la tarea n no pertenece a T.

b) Argumenta clara y precisamente que en cada una de estas dos situaciones el problema puede resolverse a partir de encontrar la solución óptima a un problema del mismo tipo pero más pequeño.

Si la tarea n no pertenece a T, entonces T es igual a la solución óptima para las tareas 1, …, n–1; es decir, un problema del mismo tipo pero más pequeño.

En cambio, si la tarea n pertenece a T, entonces ninguna tarea q, b[n] < q < n, puede pertenecer a T, por lo que T debe incluir, además de la tarea n, una solución óptima para las tareas 1, …, b[n]; es decir, nuevamente, un problema del mismo tipo pero más pequeño.

c) Generalizando, podemos decir que si T_j es la solución óptima al problema de las tareas 1, …, j, y su valor es OPT(j), entonces el problema original consiste en buscar T_n y encontrar su valor OPT(n). Escribe una ecuación recursiva para encontrar OPT(j), en función de v_j, b[j] y j–1.

La generalización se traduce a

si j pertenece a T_j , entonces OPT(j) = v_j + OPT(b[j])

si j no pertenece a T_j , entonces OPT(j) = OPT(j–1)

Por lo tanto, ***OPT(j) = max{ v_j + OPT(b[j]) , OPT(j–1) }*** … (falta agregar condición de borde)

d) Explica claramente cuál sería la dificultad práctica de tratar de resolver el problema aplicando directamente la ecuación recursiva anterior.

La ecuación recursiva hace dos llamadas recursivas, pero (a diferencia de *quicksort* o *mergesort*, en que las llamadas recursivas son sobre conjuntos disjuntos de datos) estas pueden ser llamadas para resolver el mismo problema, o para resolver problemas que ya han sido resueltos; es decir, las llamadas son sobre conjuntos de datos no necesariamente disjuntos y, por lo tanto, pueden terminar resolviendo un mismo problema muchas veces. Así, ***el número total de problemas efectivamente resueltos puede ser mucho mayor que el número total de problemas diferentes***.

e) Explica precisamente cómo la programación dinámica ayuda a resolver la dificultad de d).

La P.D. parte resolviendo los problemas más pequeños (problemas que no dan origen a llamadas recursivas) primero y va almacenando estos problemas (convenientemente codificados) y sus resultados en una tabla. Así, cuando aparece un problema para resolver, primero se busca en la tabla a ver si ese problema ya está resuelto, en cuyo caso se usa el resultado que está en la tabla y no se vuelve a resolver el problema.

I3 2018-2

4. Rutas más cortas / programación dinámica

I. (3pts) El canal DCC TV tiene un nuevo programa llamado EDD donde los participantes tienen la chance de concursar y ganar dinero en premios. El juego consiste en un triángulo de pelotas, donde cada pelota tiene impreso un número íntegro, tal como se muestra en la figura.

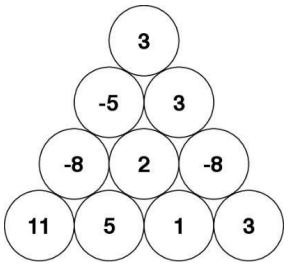


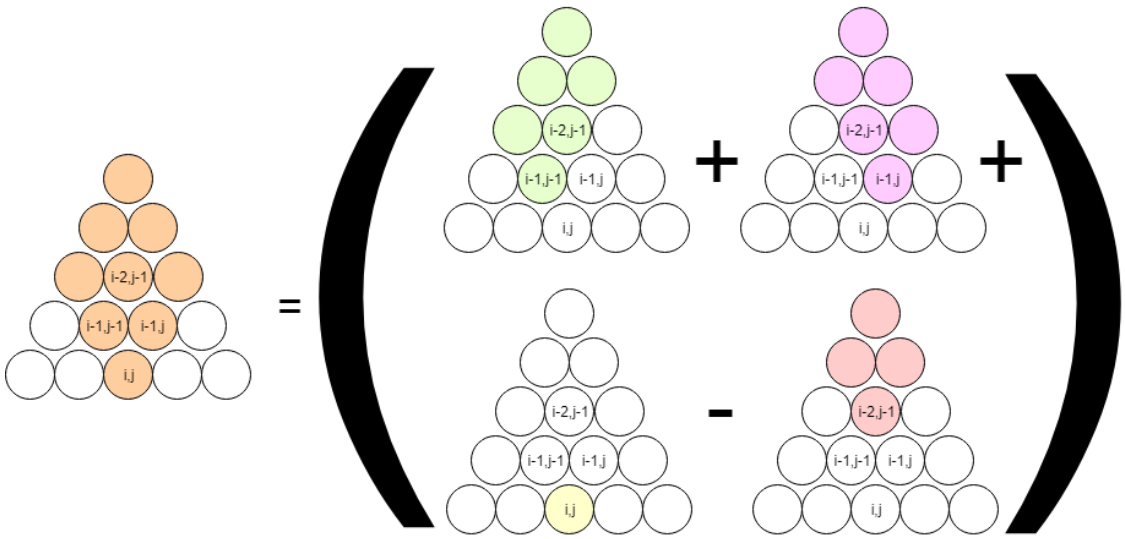
Figura 1

El jugador puede elegir una pelota de la pirámide y sacarla o no sacar ninguna. En caso de que saque una pelota, se quedará con los puntos de esa pelota y ***todas las que estén arriba de ella***. En caso de que saque ninguna, se quedará con 0 puntos. En la figura 1, para sacar la pelota 1, se necesitan sacar también las pelotas 2, -8, -5, 3 y 3 por lo que el valor de sacar la pelota 1 sería -4. El presentador está preocupado por el máximo premio que se pueda llevar un concursante, por lo que se te pide ayuda a ti para resolver este problema.

Esta pirámide se representa como arreglo de arreglos M donde M[0] es el nivel superior y M[n] es el nivel de más abajo. Cada nivel M[i] tiene i+1 elementos.

a) Haz la ecuación de recurrencia que permite calcular el premio obtenido de sacar una pelota específica: *P(i, j, M)* donde la pelota elegida es M[i][j].

Respuesta: [1.5 pts] Se acepta como respuesta correcta una función de recurrencia de forma matemática o un pseudocódigo que exprese la ecuación siguiente:



$$p(i,j,M) = \begin{cases} 0 & , \quad i,j < 0 \; o \; j > i \\ M[i][j] + p(i-1,j-1,M) + p(i-1,j,M) - p(i-2,j-1,M), & else \end{cases}$$

Esto es:
Si falta 1 término de la ecuación (normalmente falta el -*p(i-2, j-1, M)*), se da 1 punto de 1.5

b) Crea una solución iterativa de este problema en la cual se calcula el premio para toda pelota de la pirámide (Versión bottom-up).

Respuesta: [1.5 pts] Para hacer una respuesta iterativa para toda la pirámide debo crear una tabla R que contenga las respuestas e implementarlo de la siguiente forma:

```
premios(M):
    R = Tabla de las mismas dimensiones de M
    for i = 0..n:
        for j = 0..i:
            R[i][j] = M[i][j]
            if i > 0:
                if j > 0:
                    R[i][j] += R[i-1][j-1]
                if j < i:
                    R[i][j] +=R[i-1][j]
                if i - 2 >= 0 and j > 0:
                    R[i][j] -=R[i-2][j-1]
    return R
```

Se acepta como respuesta correcta la implementación iterativa del problema modelado en 4a) independiente de si es correcto para el problema.

II. ***(3pts)*** La gran ventaja del algoritmo de Floyd-Warshall frente a Dijkstra es que permite tener las rutas precalculadas y almacenadas. El tiempo que toma en construir la matriz de costos mínimos entre todos los pares de nodos es O(|V|3). Digamos el algoritmo de Floyd-Warshall es muy complicado de entender por lo que queremos utilizar el algoritmo de Dijkstra para calcular esta matriz de costos mínimos. Calcule la complejidad de este método.
Considere que el algoritmo de Dijkstra toma tiempo O(|E| log(|V|)) en su implementación con un min heap como cola de prioridad.

Respuesta: [3 pts] Para rellenar la matriz de distancias mínimas hace falta obtener los valores de min_dist(i, j) para todos los pares de nodos i, j. Dijkstra permite obtener el valor de min_dist desde un nodo i hasta todos los nodos del grafo, por lo que con una llamada a Dijkstra se rellena una fila de la matriz. Por lo tanto solo es necesario usar Dijkstra |V| veces. Esto tiene un costo de O(|V||E|log(|V|)).

Se dio 0 puntos por usar Dijkstra por cada par de nodos.

Se dio 2 puntos por dar mal la complejidad a pesar de haber explicado bien el procedimiento.

EX 2018-1

2. Programación dinámica

Queremos dar vuelto de S pesos usando el menor número posible de monedas. Si los valores de las monedas, en pesos, ordenados de mayor a menor son $\{v_1, v_2, \ldots, v_n\}$ (es decir, $v_1 > v_2 > \ldots > v_n = 1$), y tenemos una cantidad suficientemente grande de monedas de cada valor, entonces:

a) Muestra que la estrategia codiciosa de dar tantas monedas como sea posible de valor v_1, seguido de dar tantas monedas como sea posible de valor v_2, y así sucesivamente con v_3, etc., no siempre usa el menor número posible de monedas para totalizar S pesos. [***1 pto***]

b) Demuestra, en cambio, que el problema siempre puede resolverse usando programación dinámica. En particular, sea $z(S,n)$ el problema de encontrar el menor número de monedas necesarias para totalizar la cantidad S, con monedas de valor $\{v_1, v_2, \ldots, v_n\}$; entonces:

b1) Dada una solución a $z(S,n)$, identifica subpartes de la solución que sean soluciones óptimas para algunos subproblemas (del mismo tipo, pero más pequeños); o bien, identifica subproblemas cuyas soluciones óptimas puedan ser usadas para construir una solución a $z(S,n)$. [***1.5 pts.***]

b2) Escribe la recurrencia que relaciona la solución a $z(S,n)$ con las soluciones óptimas a los subproblemas; luego, generaliza esta recurrencia de modo que sea aplicable para resolver los subproblemas; y, finalmente, escribe los casos iniciales (es decir, los casos cuyos valores se determinan sin aplicar la recurrencia). [***2 pts.***]

b3) Escribe el algoritmo de programación dinámica, típicamente un algoritmo iterativo. (Si, en cambio, escribes un algoritmo recursivo, asegúrate de incluir los tests necesarios para evitar hacer cálculos redundantes). [***1.5 pts.***]

Respuesta:

a) Basta con dar un contraejemplo donde usar la estrategia greedy no llega al óptimo. Esto se puede hacer cuando los valores de las monedas no son divisores de las monedas más grandes ***[1pto]***.

b1) Digamos que la moneda más grande usada en nuestra solución a $z(S,n)$ es v_k. Entonces el problema $z(S-v_k,n)$ fue resuelto de manera óptima para que $z(S,n)$ sea óptimo ***[1.5pts]***.

b2) Recurrencia ***[2pts]***:

$$z(S,n) = \begin{cases} \infty \; si \; S<0 \\ 0 \; si \; S=0 \\ \min\limits_{i=1..n} \left(z(S-v_i,n)+1 \right) \end{cases}$$

OJO: Hay muchas maneras de escribir esta recurrencia. Esta es solo una manera de hacerla.

b3)

Modo recursivo:

```
int z(S, n, T):
        si T[S] está en la tabla:
                return T[S]
        si S < 0:
                return infinity
        si S = 0:
                return 0

        minimo = infinity

        for i = 1…n:
                result = z(S-v_i, n, T) +1
                si result < minimo:
                        minimo = result
        T[S] = minimo
        return minimo
```

Modo iterativo:

```
int z(S, n):
        T = [-1 for i = 0…S]
        T[0] = 0
        for i = 1...S:
                si tengo una moneda de costo S:
                        T[i] = 1
                        Break
                minimo = infinity
                for j = 1…n:
                        si v_j ≤ S:
                                result = T[S-v_j] +1
                                si result < minimo:
                                        minimo = result
                T[S] = minimo
```

OJO: Hay muchas maneras de escribir estos algoritmos.

2015-2

5a. Un ladrón entra a una tienda llevando una mochila con capacidad de 10 kg. En la tienda, el ladrón encuentra tres tipos de objetos (aunque hay innumerables objetos de cada tipo): los objetos de tipo 1 pesan 4 kg y tienen un valor de 11; los de tipo 2, pesan 3 kg y valen 7; y los de tipo 3, pesan 5 kg y valen 12. ¿Con cuáles objetos debe llenar la mochila el ladrón para maximizar su valor sin exceder su capacidad? Resuelve este problema empleando **programación dinámica**; en particular:

a) Demuestra que el problema exhibe la propiedad de subestructura óptima.

Subestructura óptima significa que la solución óptima al problema original contiene soluciones óptimas a problemas más pequeños del mismo tipo; en este caso, ya sea mochilas de menor capacidad, o bien sólo uno o dos tipos de objetos, o bien mochilas de menor capacidad y sólo uno o dos tipos de objetos.

La mochila óptima puede contener o no objetos de tipo 1. Hay que analizar ambos casos y quedarse con el que produce el mejor resultado. Supongamos que la mochila óptima contiene al menos un objeto de tipo 1. Esto significa que su valor es 11 más el valor del resto de la mochina. Pero este valor debe corresponder a una mochila óptima de capacidad 6 kg (= 10 kg – 4 kg) con objetos de los tipos 1, 2 y 3: un problema similar al original, pero más pequeño.

Supongamos ahora que la mochila óptima no contiene objetos de tipo 1. Esto significa que es equivalente a una mochila óptima de capacidad 10 kg, pero que sólo contiene objetos de los tipos 2 y 3: nuevamente, un problema similar al original, pero más pequeño.

b) Plantea la solución recursivamente.

Si representamos la solución óptima al problema de una mochila con capacidad C y objetos de los tipos t_1, t_2 y t_3 por $[C, \{t_1,t_2,t_3\}]$, entonces, del análisis de **a**), tenemos que

$[10, \{1,2,3\}] = \max\{ 11 + [6, \{1,2,3\}], [10, \{2,3\}] \}$.

Y para una instancia intermedia cualquiera, gracias a la propiedad de subestructura óptima, suponiendo objetos de tipos $t_j, \ldots, t_k$ con valores $v(t_i)$ y pesos $w(t_i)$,

$[c, \{t_j, \ldots, t_k\}] = \max\{ v(t_j) + [c - w(t_j), \{t_{j+1}, \ldots, t_k\}], [c, \{t_{j+1}, \ldots, t_k\}] \}$

c) Desarrolla la formulación recursiva de la solución, de modo de responder la pregunta anterior.

Si se desarrolla la formulación anterior a partir de la primera ecuación y aplicando la segunda en los pasos intermedios, finalmente obtenemos que la solución óptima es un objeto de tipo 1 y dos de tipo 2: llenan la mochila exactamente y su valor total es 25.

7. Encuentra los costos de las rutas más cortas entre todos los pares de vértices para el siguiente grafo direccional representado por su matriz de adyacencias, **empleando el algoritmo de Floyd–Warshall**. P.ej., el costo de la arista que va del vértice 1 al vértice 2 es 51; y no hay arista del vértice 3 al vértice 4. En particular, muestra cada una de las siguientes **tres** matrices que produce el algoritmo.

	0	**1**	**2**	**3**	**4**	**5**
0	0	41				29
1		0	51		32	
2			0	50		
3	45			0		38
4			32	36	0	
5		29			21	0

	0	**1**	**2**	**3**	**4**	**5**
0	0	41				29
1		0	51		32	
2			0	50		
3	45	86		0		38
4			32	36	0	
5		29			21	0

	0	**1**	**2**	**3**	**4**	**5**
0	0	41	92		73	29
1		0	51		32	
2			0	50		
3	45	86	137	0	118	38
4			32	36	0	
5		29	80		21	0

	0	**1**	**2**	**3**	**4**	**5**
0	0	41	92	142	73	29
1		0	51	101	32	
2			0	50		
3	45	86	137	0	118	38
4			32	36	0	
5		29	80	130	21	0

EX 2015-2

4. [**Este problema vale doble**] Considera el siguiente grafo direccional con costos, con vértices a, b, c, d y e, representado mediante sus listas de adyacencias:

[a]: [b, 3] – [c, 8] – [e, –4] [b]: [d, 1] – [e, 7] [c]: [b, 4] [d]: [a, 2] – [c, –5] [e]: [d, 6]

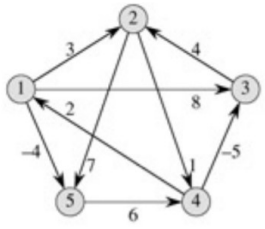
El algoritmo de Floyd-Warshall para determinar las rutas más cortas entre todos los pares de vértices en un grafo direccional con costos es el siguiente:

D = *matriz de adyacencias*
for k = 1 … n —n *es el número de vértices*
  for i = 1 … n —*para cada vértice* i
    for j = 1 … n —*para cada vértice* j
      $d_{ij} = \min(d_{ij}, d_{ik}+d_{kj})$
return D

a) Ejecuta el algoritmo de Floyd-Warshall sobre el grafo anterior; muestra el contenido de la matriz D después de cada iteración del índice k, y encuentra tanto las longitudes de las rutas más cortas **como las rutas propiamente tales**.

Ver p. 696 de Cormen et al. [2009].

EXTRACTO CORMEN:




```
FLOYD-WARSHALL(W)
1  n ← rows[W]
2  D^(0) ← W
3  for k ← 1 to n
4       do for i ← 1 to n
5             do for j ← 1 to n
6                   do d_ij^(k) ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return D^(n)
```

Figure 25.4 shows the matrices $D^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

$$D^{(0)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & \infty & -5 & 0 & \infty \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(0)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & \text{NIL} & 4 & \text{NIL} & \text{NIL} \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(1)} = \begin{pmatrix} 0 & 3 & 8 & \infty & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & \infty & \infty \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(1)} = \begin{pmatrix} \text{NIL} & 1 & 1 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & \text{NIL} & \text{NIL} \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(2)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & 5 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(2)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 1 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(3)} = \begin{pmatrix} 0 & 3 & 8 & 4 & -4 \\ \infty & 0 & \infty & 1 & 7 \\ \infty & 4 & 0 & 5 & 11 \\ 2 & -1 & -5 & 0 & -2 \\ \infty & \infty & \infty & 6 & 0 \end{pmatrix} \quad \Pi^{(3)} = \begin{pmatrix} \text{NIL} & 1 & 1 & 2 & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 2 & 2 \\ \text{NIL} & 3 & \text{NIL} & 2 & 2 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ \text{NIL} & \text{NIL} & \text{NIL} & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(4)} = \begin{pmatrix} 0 & 3 & -1 & 4 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(4)} = \begin{pmatrix} \text{NIL} & 1 & 4 & 2 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

$$D^{(5)} = \begin{pmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{pmatrix} \quad \Pi^{(5)} = \begin{pmatrix} \text{NIL} & 3 & 4 & 5 & 1 \\ 4 & \text{NIL} & 4 & 2 & 1 \\ 4 & 3 & \text{NIL} & 2 & 1 \\ 4 & 3 & 4 & \text{NIL} & 1 \\ 4 & 3 & 4 & 5 & \text{NIL} \end{pmatrix}$$

Figure 25.4: The sequence of matrices $D^{(k)}$ and $\Pi^{(k)}$ computed by the Floyd-Warshall algorithm for the graph in Figure 25.1.

The running time of the Floyd-Warshall algorithm is determined by the triply nested **for** loops of lines 3-6. Because each execution of line 6 takes $O(1)$ time, the algorithm runs in time $\Theta(n^3)$. As in the final algorithm in Section 25.1, the code is tight, with no elaborate data structures, and so the constant hidden in the Θ-notation is small. Thus, the Floyd-Warshall algorithm is quite practical for even moderate-sized input graphs.

b) ¿Cómo podemos usar el resultado del algoritmo de Floyd-Warshall para detectar la presencia de un **ciclo con costo acumulado negativo**? Justifica.

Una posibilidad es ejecutar una iteración adicional, con $k = n+1$, y ver si alguno de los valores de la matriz D cambia. Si hay CCANs, entonces el costo de alguna ruta más corta deberá disminuir más allá del "mínimo" entontrado después de n iteraciones.

Lo otro es mirar los valores de la diagonal de D: hay un CCAN si y sólo si alguno de esos valores es negativo. **Demostración**. … (si la necesitan, díganme).

Este algoritmo es un algoritmo de programación dinámica; **muestra que es aplicable al problema**:

c) Demuestra que el problema tiene subestructura óptima.

Trivial. Lo hicimos varias veces en clases.

d) Demuestra que una formulación recursiva de la solución (~~p.ej., a partir de **b***~~[este "hint" no tenía *nada que ver*]) presenta la característica de *subproblemas traslapados*.

La formulación recursiva útil (que también da origen al algoritmo) es la siguiente: Consideremos rutas más cortas que sólo usan los vértices 1, 2, …, k como vértices intermedios. Una ruta más corta de i a j (la *ruta*) puede incluir o no al vértice k; como no sabemos, debemos calcular ambas posibilidades y quedarnos con la mejor (similarmente a como lo hicimos en otros problemas):

- si no lo incluye, entonces la *ruta* es idéntica a la ruta más corta de i a j que sólo usa los vértices 1 a k–1 como vértices intermedios;

- si lo incluye, entonces (por subestructura óptima) la *ruta* es la concatenación de la ruta más corta de i a k con la ruta más corta de k a j, ambas usando sólo los vértices 1 a k–1 como vértices intermedios.

EXAMEN 2014-1

4) Tenemos dos strings, $X = x_1x_2…x_m$ e $Y = y_1y_2…y_n$. Consideremos los conjuntos {1, 2, …, m} y {1, 2, …, n} que representan las posiciones (de los símbolos) en los strings X e Y, y consideremos un *emparejamiento* de estos conjuntos, es decir, un conjunto de pares ordenados tal que cada item aparece en a lo más un par. P.ej., si X = `hola` e Y = `ollas`, entonces los conjuntos son {1, 2, 3, 4} y {1, 2, 3, 4, 5}, un emparejamiento podría ser { (2,1), (3,2), (4,4) } (que empareja letras iguales entre ellas), y otro { (1,5), (4,1) } (que empareja letras extremas entre ellas).

Decimos que un emparejamiento M es una *alineación* si no hay pares que "se crucen": si (j, k), $(j', k') \in M$, y si $j < j'$, entonces $k < k'$; p.ej., el primer emparejamiento anterior es una alineación, pero el segundo no. Dado M, un *gap* (brecha) es una posición de X o Y que no está en M; p.ej., para la alineación anterior, la posición 1 de X, y las posiciones 3 y 5 de Y son gaps.

Finalmente, podemos asociar un costo a una alineación: cada gap incurre un costo fijo $\delta > 0$; además, si para cualquier par de símbolos u, v del alfabeto del que provienen X e Y hay un costo $\alpha(u, v)$ de emparejamiento, entonces cada par (i, j) de M tiene un costo $\alpha(x_i, y_j)$. El *costo de la alineación* es la suma de los costos debidos a sus gaps más los costos debidos a sus emparejamientos. P.ej., el costo de la alineación anterior es $3\delta + \alpha($`o`, `o`$) + \alpha($`l`, `l`$) + \alpha($`a`, `a`$)$.

En estas condiciones, dados dos strings, X e Y, queremos encontrar la *alineación de costo mínimo*. Muestra que este problema puede ser resuelto mediante programación dinámica. En particular,

a) Muestra que —explica por qué o explica cómo— el problema puede ser visto como el resultado de una secuencia de decisiones.

En una alineación de costo mínimo M, ya sea $(m, n) \in M$ o $(m, n) \notin M$; en este último caso, ya sea la posición m de X o bien la posición n de Y no aparecen emparejadas en M. Esta última afirmación se demuestra por contradicción: de lo contrario, habría pares "cruzados" y no tendríamos propiamente una *alineación*, según la definición de más arriba.

Si $(m, n) \in M$, entonces pagamos el costo $\alpha(x_m, y_n)$ y alineamos lo mejor posible $x_1 \ldots x_{m-1}$ con $y_1 \ldots y_{n-1}$; es decir, $\text{opt}(m,n) = \alpha(x_m, y_n) + \text{opt}(m-1, n-1)$.

Si, en cambio, la posición m de X no está emparejada, pagamos el costo δ del gap y alineamos lo mejor posible $x_1 \ldots x_{m-1}$ con $y_1 \ldots y_n$; es decir, $\text{opt}(m,n) = \delta + \text{opt}(m-1, n)$. Y similarmente si la posición n de Y no está emparejada, obteniendo $\text{opt}(m,n) = \delta + \text{opt}(m,n-1)$.

b) Muestra que —explica por qué o explica cómo— se verifica el *principio de optimalidad*, cuando se aplica al estado del problema que resulta al tomar una decisión.

De la argumentación de **a**), cuando decimos "y alineamos lo mejor posible".

Si M es una alineación óptima y sacamos el par que contiene las posiciones m y/o n, entonces el resto de los pares, M', es una alineación óptima para los demás símbolos de X e Y. Si no fuera así y hubiera una alineación M'' mejor que M' para el resto de los símbolos, entonces podríamos cambiar M' por M'' en M y obtener una alineación mejor que M, contradiciendo que M sea óptima.

c) A partir de **a**) y **b**), plantea una ecuación recursiva para calcular el costo de la alineación de costo mínimo (la ecuación debe estar planteada, entre otros, en términos de los costos de subalineaciones óptimas); incluye las condiciones de borde —es decir, cuando la recursión ya no aplica más— que permiten resolver la ecuación.

Para $i \geq 1$ y $k \geq 1$,

$$\text{opt}(i, k) = \min\{ \alpha(x_i, y_k) + \text{opt}(i-1, k-1), \delta + \text{opt}(i-1, k), \delta + \text{opt}(i, k-1) \}$$

con $\text{opt}(i, 0) = \text{opt}(0, i) = i\delta$, para todo i (ya que la única forma de alinear una palabra de i letras con una de 0 letras es usar i gaps)

EXAMEN 2013-1

D) Considera el problema de determinar las **rutas más cortas entre todos los pares de vértices** de un grafo direccional $G = (V, E)$ con costos en las aristas.

13) Enuncia las dos propiedades características de los problemas de optimización que pueden ser resueltos mediante programación dinámica.

Las dos propiedades características son: Propiedad de Subestructura óptima y Propiedad de subproblemas traslapados.

- Subestructura óptima: La solución óptima al problema original puede ser construida con soluciones óptimas de subproblemas.
- Subproblemas traslapados: Si uno utiliza un algoritmo recursivo para resolver el problema, entonces existirá al menos un subproblema que será resuelto varias veces.

14) Muestra que este problema cumple ambas propiedades.

Para la propiedad de subestructura óptima, suponga que p(u,v) es el camino de menor costo desde u hasta v. Sea x el nodo inmediatamente anterior a v en el camino p (por lo que (x,v) es una arista). Entonces suponga que p(u,x) no es el camino más corto desde u hasta x. Entonces existe otro camino p'(u,x) que sí es óptimo. Luego, agregue la arista (x,v) a este nuevo camino. Y por lo tanto, tenemos que:

$$w(p'(u,x))+w(x,v) < w(p(u,x))+w(x,v)$$

Por lo que el camino p' seguido por (x,v) es mucho mejor que el camino p(u,v) original. Luego, p(u,v) no es óptimo y hay una contradicción por suponer que no tiene subestructura óptima. Por lo tanto, este problema exhibe subestructura óptima.

Así, para ir desde u hasta v, considere la siguiente recursión, donde d(x,y) es el costo del camino más corto desde x hasta y:

d(x,x) = 0
d(x,y) = w(x,y) si (x,y) es una arista
d(x,y) = +infinito si y no es alcanzable desde x
d(x,y) = min{d(x,u)+d(u,y) | u está en V}

Para la propiedad de subproblemas traslapados, suponga que quiere encontrar el valor de d(x,y) y para ello, necesita encontrar el valor de d(x,u) y d(u,y). También necesita encontrar el valor de d(x,a) y d(a,y). Para encontrar el valor de d(x,u) necesita de d(x,a) y d(a,u). Luego tiene que calcular al menos dos veces el valor de d(x,a) y este problema tiene subproblemas traslapados.

15) Plantea un algoritmo eficiente para resolverlo; justifica que tu algoritmo es eficiente.

(No basta con decir que el algoritmo corresponde al de Floyd-Warshall, debe plantearse el algoritmo)

Suponga que tenemos una matriz W donde W(i,j) indica el costo de ir desde el nodo i hasta el nodo j. Sea D otra matriz, inicialmente vale D=W y que tenemos N nodos. El algoritmo es:

```
D=W;
for k=1..N{
  for i=1..N{
    for j=1..N{
      if(D(i,j)>D(i,k)+D(k,j))
        D(i,j) = D(i,k)+D(k,j)
    }
  }
}
```

El algoritmo toma tiempo O(n^3). Es más eficiente que ejecutar N veces Dijkstra (que toma tiempo O((V+E)log V)) y más eficiente que ejecutar N veces Bellman-Ford (que toma tiempo O(VE))

16) El ***diámetro*** de G es la más larga de las rutas más cortas en G. A partir de tu algoritmo para 15), da un algoritmo eficiente que imprima la *secuencia de vértices* del diámetro de G.

Si uno simplemente ejecuta el algoritmo anterior, obtiene una matriz D con las distancias más cortas entre todos los pares de vértices; el mayor valor en esta matriz es la magnitud del diámetro del grafo; pero falta la secuencia de vértices.

Para esto, entre otras posibilidades, [Cormen et al., 2001] sugiere construir una matriz P iterativamente, similarmente y en paralelo a la de las distancias más cortas, pero que represente las rutas propiamente tales. Si en la iteración k me conviene incluir el vértice k para ir de u a v, entonces $P[u, v] = P[k, v]$ de la iteración k–1; de lo contrario, $P[u, v] = P[u, v]$ de la iteración k–1. Así, al finalizar Floyd-Warshall, $P[u, v]$ contiene el vértice anterior a v en la ruta más corta de u a v.

Finalmente, buscamos en D la más larga de las distancias más cortas, y usamos P para reconstruir la ruta correspondiente.