Práctica 8

1. Supongamos que disponemos de n archivos f_1, f_2 , ... , f_n con tamaños l_1, l_2, ..., l_n, y un disquete de capacidad $d < l_1 + l_2 + \ldots + l_n$.

a) Queremos maximizar el número de ficheros que ha de contener el disquete, y para eso ordenamos los ficheros por orden creciente de su tamaño y vamos metiendo ficheros en el disco hasta que no podamos meter más. Determinar si este algoritmo greedy encuentra solución óptima en todos los casos.

b) Queremos llenar el disquete tanto como podamos, y para eso ordenamos los ficheros por orden decreciente de su tamaño, y vamos metiendo ficheros en el disco hasta que no podamos meter más. Determinar si este algoritmo greedy encuentra solución óptima en todos los casos.

c) De los casos anteriores que satisfacen la forma greedy, programar una rutina que resuelva el problema.

2. El problema de la mochila 0-1 consiste en encontrar la manera óptima de cargar una mochila con objetos. Existen n objetos disponibles. El i-ésimo objeto cuesta v_i pesos, y pesa w_i kilos, donde v_i y w_i son enteros. Se pretende cargar el mayor valor en pesos posibles pero con un máximo W de kilos. Se debe decidir qué objetos tomar. Cada objeto puede ser elegido o no. No se puede tomar una parte de un objeto ni se puede elegir un objeto más de una vez.
Existe una variante, el problema de la mochila fraccional, de descripción similar, salvo que está permitido tomar fracciones de objetos.

a) Programar un algoritmo greedy para resolver el problema de la mochila fraccional.

b) Explicar por qué este algoritmo no se puede extender al problema de la mochila 0-1.

3. El problema del cambio consiste en determinar, dado un número entero C, una combinación de monedas de 1 ctvo., 5 ctvos., 10 ctvos. y 20 ctvos. que sumen C y que use la mínima cantidad de monedas posible.

a) Programar un algoritmo greedy para resolver este problema.

b) ¿Este algoritmo resuelve el problema para cualquier denominación de monedas? Justificar.

4. Las palabras *pelota* y *paleta* son muy similares. En efecto, una puede transformarse en otra cambiando solo dos letras. La palabra *palta* también es similar a *paleta* porque puede transformarse *palta* en *paleta* insertando la letra *e* entre la *l* y la *t*, o equivalentemente puede transformarse *paleta* en *palta* eliminando la *e* entre la *l* y la *t*. La distancia de edición de dos palabras s_1 y s_2, $d(s_1, s_2)$ se define como el mínimo número de operaciones requeridas para transformar s_1 en s_2 donde una operación puede ser:

- Sustituir una letra por otra.
- Insertar una letra.
- Borrar una letra.

a) Completar las siguientes igualdades, que son suficientes para definir la distancia:

$$\begin{aligned} d(\epsilon, \epsilon) &= \ldots \\ d(s_1, \epsilon) &= \ldots \\ d(\epsilon, s_2) &= \ldots \\ d(s_1 \cdot c_1, s_2 \cdot c_2) &= \text{mín}(\ldots) \end{aligned}$$

donde ϵ es la cadena vacía, s_1 y s_2 son cadenas, c_1 y c_2 son letras.

b) Utilizando la definición del ítem anterior, programar un algoritmo dinámico que calcule la distancia entre las dos palabras.

c) Calcular la complejidad temporal del algoritmo anterior.

5. Existe otra variante del problema de la mochila llamado la mochila no acotada. En este caso, nuevamente dada una lista de objetos, con su precio y peso, debemos llenar una mochila de peso W. Sin embargo, en esta versión, podemos poner en la mochila un objeto tantas veces como queramos (obviamente incrementando el precio y peso de manera proporcional).
Resolver esta variante utilizando un algoritmo dinámico.

6. Escribir una versión divide and conquer de un algoritmo que dado un arreglo busque los dos elementos más grandes de un arreglo. Hacer un análisis de su complejidad.

7. Escribir una función por backtracking que imprima todos los números de 6 cifras que no tengan repeticiones de dígitos.

8. Modificar el problema visto en clase de backtracking de manera que pare al encontrar la primer solución.

9. Escribir una función por backtracking que resuelva Sudoku.

10. Resolver el problema de la mochila 0-1 utilizando un algoritmo dinámico. Analizar su complejidad.

11. Dada una secuencia $X = \langle x_1, x_2, \ldots, x_m \rangle$, se se dice que una secuencia $Z = \langle z_1, z_2, \ldots, z_k \rangle$ es una subsecuencia de X si existe una secuencia estrictamente creciente de índices $\langle i_1, i_2, \ldots, i_k \rangle$ de X tal que $\forall j = 1, \ldots, k \cdot x_{i_j} = z_j$. Por ejemplo, $Z = \langle B, C, D, B \rangle$ es una subsecuencia de $X = \langle A, B, C, B, D, A, B \rangle$ con la correspondiente secuencia de índices $\langle 2, 3, 5, 7 \rangle$. Dadas dos secuencias X e Y, se dice que una secuencia Z es una subsecuencia en común de ambas si es una subsecuencia tanto de X como de Y.
El problema de la subsecuencia en común más larga (LCS) consiste en encontrar una subsecuencia en común de máxima longitud entre dos secuencias X e Y.
Sea $X = \langle x_1, x_2, \ldots, x_n \rangle$ una secuencia, se define X_i como el prefijo $\langle x_1, x_2, \ldots, \rangle$ de X. Sean $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$ dos secuencias y $Z = \langle z_1, z_2, \ldots, z_k \rangle$ una LCS de ambas, demostrar que:

- $x_m = y_n \implies z_k = x_m = y_n$ y Z_{k-1} es una LCS de X_{m-1} e Y_{n-1}.
- $x_m \neq y_n \wedge z_k \neq x_m \implies Z$ es una LCS de X_{m-1} e Y.
- $x_m \neq y_n \wedge z_k \neq y_n \implies Z$ es una LCS de X e Y_{n-1}.

En base al resultado del punto anterior, escriba un algoritmo que encuentre la longitud de una LCS entre dos secuencias X e Y usando la técnica de programación dinámica. Calcule el orden de complejidad temporal y espacial del algoritmo. Explique informalmente cómo puede modificarse el algoritmo para que además de encontrar la longitud de una LCS entre dos secuencias, también encuentre una LCS. Indique el orden O de complejidad temporal que requeriría encontrar la LCS.

12. Se desea obtener el producto de n matrices $A_1, A_2, \ldots, A_n$. Dado que el producto de matrices es asociativo, se puede elegir diferentes maneras para obtener el resultado final. Por ejemplo, si $n = 4$ tenemos, entre otras, las siguientes formas:

- $(A_1 \times A_2) \times (A_3 \times A_4)$
- $A_1 \times (A_2 \times (A_3 \times A_4))$

Asumiendo que un algoritmo para multiplicar dos matrices $A_{m\times p}$ y $B_{p\times n}$ realiza $m \times p \times n$ multiplicaciones, podemos observar que la elección de asociatividad que realicemos será determinante para la complejidad de nuestro algoritmo.
Volviendo al ejemplo, supongamos que A_1, A_2, A_3, A_4 son de orden 10×100, 100×5, 5×3 y 3×30 respectivamente. Entonces la cantidad de multiplicaciones para cada una de las formas de asociatividad vistas arriba son:

- $10 \times 100 \times 5 + 5 \times 3 \times 30 + 10 \times 5 \times 30 = 6950$
- $5 \times 3 \times 30 + 100 \times 5 \times 30 + 10 \times 100 \times 30 = 31950$

Como vemos, es importante decidir de qué forma vamos a multiplicar la matrices. Podemos expresar el problema de multiplicación de matrices como sigue:
Dadas n matrices $A_1, A_2, \ldots, A_n$, donde A_i tiene dimensiones $p_{i-1} \times p_i$, debemos dar una expresión con todos los paréntesis de $A_1 \times A_2 \times \ldots A_n$. de forma tal que el número de multiplicaciones se minimice al calcular el producto final.
Usando programación dinámica, resuelva este problema.

13. Dado un arreglo de puntos, escribir un algoritmo divide and conquer que determine cuál es la mínima distancia entre dos puntos. *Ayuda:* Dividir el plano en dos (según su coordenadas) de manera que haya la misma cantidad de puntos de cada lado. Tener en cuenta el siguiente teorema: un rectángulo de d por $2d$ puede contener a lo sumo seis puntos de manera que cualquiera dos puntos estén a distancia al menos d.

14. Considere un tablero con $n \times n$ cuadros y una función costo $c(i, j)$ que retorna un costo asociado a cada cuadro (i, j). Se debe mover una ficha de dama desde la parte inferior del tablero hacia la parte superior del mismo. En cada paso se puede mover la ficha hacia uno de los siguientes cuadros:

- El cuadro inmediatamente superior.
- El cuadro que se encuentra uno arriba y uno a la izquierda (si la ficha no estaba en la columna izquierda).
- El cuadro que se encuentra uno arriba y uno a la derecha (si la ficha no estaba en la columna derecha).

a) Utilice programación dinámica para diseñar un algoritmo que retorne el mínimo costo de mover una ficha desde cualquier cuadro de la parte inferior del tablero hacia cualquier cuadro de la parte superior del mismo. El costo del camino es la suma de los costos de los cuadros por los que pasa la ficha.

b) Adapte el algoritmo del ítem anterior para obtener la secuencia de cuadros del camino de mínimo costo.