Facultad de Ciencias Exactas, Ingeniería y Agrimensura
Escuela de Ciencias Exactas y Naturales
Departamento de Ciencias de la Computación
R-212 — Estructuras de Datos y Algoritmos I

Práctica A3

Estrategias y Algoritmos

En todos los problemas: considere la eficiencia tanto temporal como espacial de las soluciones.

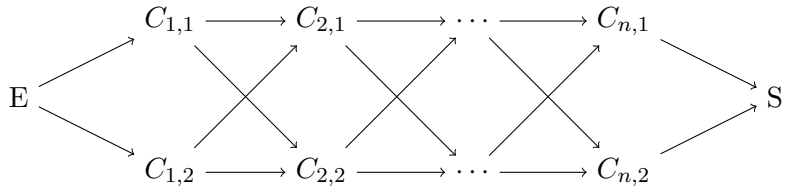
1 (Fibonacci). Implemente una función que compute el n-ésimo número de Fibonacci usando programación dinámica. Debería funcionar con tiempo en $O(n)$ y memoria en $O(1)$.

2 (Bowling). En una línea, tenemos N "bolos" de Bowling, cada uno con un valor asociado (un entero, posiblemente negativo). El juego consiste en bajar los bolos para maximizar el puntaje. Está permitido bajar un solo bolo, obteniendo tantos puntos como el valor del mismo, o dos bolos *consecutivos* a la vez, consiguiendo como puntaje el *producto* de ambos valores. Diseñe un algoritmo que, dado N y los puntajes de cada bolo, compute el máximo puntaje que puede obtenerse, junto con la forma de obtenerlo.

9 -5 -7 4 2 1 -6

Figura 1: Ejemplo de entrada. El puntaje máximo es 53.

3 (Línea de Ensamble). En la fábrica del Sr. Harry Drof, hay dos líneas de ensamble, cada una con N máquinas. Las máquinas de ambas líneas cumplen exactamente las mismas funciones, pero demoran tiempos distintos en completar su labor. Queremos buscar la manera de fabricar un único producto, pasándolo por todas las etapas, posiblemente cambiando de línea si fuera conveniente, aunque esto lleva un tiempo fijo T. Diseñe un algoritmo que, a partir del tiempo que tarda cada máquina (dado por una matriz $C_{i,j}$) y el tiempo de transferencia T, compute el tiempo mínimo requerido para fabricar un producto, junto con el camino por la fábrica que lo logra.



4 (Distancia de Edición). Dadas dos palabras, podemos transformar una en otra siguiendo las siguientes operaciones (en cualquier posición):

- Agregar un caracter
- Borrar un caracter
- Reemplazar un caracter por otro

La mínima cantidad de pasos para transformar una palabra v en la palabra w se conoce como *distancia de edición* (la denotamos $d(v, w)$), y es una distancia en el sentido matemático (por ejemplo, cumple la desigualdad triangular: $d(u, v)+d(v, w) \geq d(u, w)$). Diseñe un algoritmo que, dadas dos palabras, compute su distancia de edición. Extienda su algoritmo para además mostrar cómo realizar la transformación de una palabra en otra.

5 (Subarreglo de Suma Máxima - DC). Diseñe un algoritmo Divide-y-Vencerás que, dado un arreglo A de longitud n, compute la suma máxima de un subarreglo del mismo (un subarreglo debe ser *contiguo*). ¿Cuál es la complejidad?

6 (Subarreglo de Suma Máxima). Use la siguiente idea para desarrollar un algoritmo de tiempo lineal para el problema del subarreglo de suma máxima. El subarreglo de suma máxima termina en alguna posición j, así que podemos restringirnos a computar los subarreglos de suma máxima que terminen en cada posición $j = 0, \ldots, n-1$ y tomar el máximo de todos ellos. Conociendo un subarreglo de suma máxima que termina en j, mostrar que puede computarse un subarreglo de suma máxima terminando en $j+1$ en tiempo $O(1)$.

7 (Cambio de Divisas). Supongamos que queremos cambiar alguna cantidad de una moneda M_i a una nueva moneda M_f. En vez de hacer el cambio directo, a que tiene una tasa t_{if}, podemos hacer el cambio pasando por cualquier otra moneda, de las cuales hay n. Para cada par (i, j) de monedas, hay una tasa de cambio t_{ij} asociada.

a) Diseñe un algoritmo que, dada la matriz de tasas t, compute la mejor "tasa equivalente" para cambiar M_i a M_j.

b) Supongamos que, para cada transacción, nos cobran una comisión que es un porcentaje del monto. ¿Cómo cambia el problema y la solución?

c) Supongamos ahora que debemos pagar una comisión fija por la *cantidad* de transacciones. Es decir, si hacemos k transacciones, debemos pagar un monto fijo c_k extra. ¿Cambia el problema?

8 (LCS: Subsecuencia común más larga). 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 $\langle 2, 3, 5, 7 \rangle$. (En definitiva, una subsecuencia se forma "borrando" algunos elementos de una secuencia, sin reordenar.)

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. Definimos X_i como el prefijo $\langle x_1, x_2, \ldots, x_i \rangle$ 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, se puede demostrar (fácilmente) que:

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

Utilizando estos resultados, escriba un algoritmo que encuentre la longitud de una LCS entre dos secuencias X e Y usando la estrategia de programación dinámica. Calcule la complejidad del mismo. Modifíquelo también para, además de encontrar la longitud, imprimir una LCS.

9 (Costo de Multiplicación de Matrices). Se desea obtener el producto de n matrices $A_1, A_2, \ldots, A_n$. Dado que el producto de matrices es asociativo, se pueden 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:

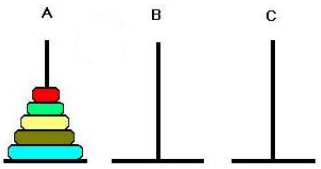
- $10 \times 100 \times 5 + 5 \times 3 \times 30 + 10 \times 5 \times 30 = 6950$
- $5 \times 3 \times 30 + 100 \times 5 \times 30 + 10 \times 100 \times 30 = 31950$

Como vemos, es importante decidir de cómo organizar el producto. 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. Escriba un programa para resolver este problema.

10 (Torres de Hanoi[1]).

Este juego consiste de tres postes verticales, uno de los cuales tiene apilados N discos de tamaño decreciente, ubicándose el disco más chico en el tope de la pila, y los restantes se encuentran vacíos.



El juego consiste en pasar todos los discos desde el poste ocupado a uno de los otros postes vacíos, siguiendo tres simples reglas:

- Sólo se puede mover un disco a la vez.
- Sólo se puede desplazar el disco que se encuentra en el tope de la pila.
- Un disco de mayor tamaño no puede estar sobre uno más pequeño.

Escriba un programa que imprima las instrucciones para pasar todos los discos del poste A al poste C.

11 (Hano*i*). Adapte su solución anterior para imprimir solamente el paso i-ésimo de la solución. Su algoritmo debería correr en tiempo $O(\lg N)$ y memoria $O(1)$.

12 (Problema del Cambio). El problema del cambio consiste en determinar, dado un número entero C, una combinación de monedas de 1, 5, 10 y 20 centavos que sumen C y que use la mínima cantidad de monedas posible.

a) ¿Siempre es posible sumar C? Justifique.

b) Programar un algoritmo *greedy* para resolver este problema.

c) ¿Este algoritmo resuelve el problema para cualquier monedas de cualquier denominación? Justifique.

d) ¿Qué pasa si las denominaciones son de la forma $1, c, c^2, \ldots, c^n$ para algún entero $c \geq 2$?

13 (Selección de Actividades). Sea $S = \{a_1, a_2, \ldots, a_n\}$ un conjunto de n actividades que necesitan hacer uso de un recurso, el cual sólo puede ser usado por a lo sumo una actividad a la vez (por ejemplo, podemos pensar en clases que deben dictarse en una misma aula). Cada actividad a_i tiene una hora de inicio s_i y una hora de finalización f_i, donde $0 \leq s_i < f_i < \infty$. En caso de seleccionarse, la actividad a_i hará uso exclusivo del recurso durante el intervalo de tiempo semi-abierto $[s_i, f_i)$. Escriba un programa

[1] Este juego fue introducido por el matemático Édouard Lucas en 1883 junto a una "leyenda" inventada por él. En esta leyenda inventada, unos monjes estaban a cargo de mover 64 discos de oro siguiendo las reglas dadas, y al terminar su tarea, el universo dejaría de existir.

que determine la máxima cantidad de actividades que se pueden seleccionar sin que haya superposición entre ellas, utilizando un algoritmo greedy.

14 (Planificación de Actividades). Partiendo del problema anterior, supongamos ahora que queremos ejecutar todas las actividades (sin excepción), y podemos usar muchas aulas distintas para hacerlo. Diseñe un algoritmo que compute el mínimo número de aulas necesario para esto.

15 (Archivos). 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 acotada d.

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. Determine 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. Determine si este algoritmo greedy encuentra solución óptima en todos los casos.

De los casos anteriores que satisfacen la forma *greedy*, programe una rutina que resuelva el problema. Si algún caso no funciona con la solución propuesta, dé un contraejemplo para demostrar que no consigue una solución óptima. ¿Cómo resolvería entonces el problema?

16 (Mochila No-acotada). Se quiere encontrar la manera óptima de cargar una mochila con objetos. Existen n clases de objetos disponibles. Los objetos de tipo i cuestan v_i pesos, y pesan w_i kilos, donde v_i y w_i son enteros. Se pretende cargar el mayor posible valor en pesos, pero con la restricción de que no podemos cargar un peso mayor a un entero W. Se puede elegir cualquier cantidad entera de objetos de cada tipo. Programe un algoritmo dinámico para resolver este problema.

17 (Mochila 0-1). Adapte su solución al problema de la mochila no acotada si agregamos la restricción de que cada objeto es único, y puede tomarse a lo sumo una vez. Si tuviera que algunos objetos aparecen alguna cantidad $N > 1$ de veces pero finita, ¿cómo resolvería el problema?

18 (Mochila Fraccionaria). Suponga ahora que sí puede tomar "parte" de un objeto, obteniendo un valor proporcional al peso tomado. (Si antes estábamos tomando lingotes de oro, podemos pensar que ahora es oro en polvo.) Diseñe un algoritmo que resuelva este problema. ¿Cuál es la complejidad? ¿Cómo se compara al anterior?

19 (Extremos de un Arreglo). Queremos encontrar los extremos (i.e. el mínimo y máximo) de un arreglo de enteros. Una solución es ir calculando ambos a la vez a medida que recorremos el arreglo. ¿Cuántas comparaciones realiza esta solución (en mejor y peor caso, si los hay)?

Ahora, notemos que para un arreglo de 2 elementos podemos encontrar los extremos con una *única* comparación. Diseñe un algoritmo divide-y-vencerás siguiendo esta idea. ¿Cuántas comparaciones hace esta versión?

20 (Planificación de Procesos). Supongamos que tenemos n procesos, donde cada uno tarda un tiempo t_i en completarse, y no podemos ejecutar dos a la vez. Queremos minimizar el tiempo promedio que tarda en completarse un proceso, desde el comienzo de la ejecución. Por ejemplo, si ejecutamos procesos de duración 3, 5, 1, y 2 en ese orden, el tiempo promedio es $\frac{3+(3+5)+(3+5+1)+(3+5+1+2)}{4}$. Diseñe un algoritmo que compute el tiempo promedio mínimo y la planificación que lo obtiene.

21* (Optimizando Búsquedas en un ABB). Supongamos que tenemos n pares clave valor (k_i, v_i) que

tenemos que insertar en un ABB. Asumimos que las claves están en orden creciente $k_0 < k_1 < \ldots < k_{n-1}$. Nos interesa minimizar el tiempo de búsqueda promedio para las consultas al árbol. Si no tuviéramos más información, podríamos pensar en armar un árbol perfectamente balanceado.

Supongamos que conocemos las *probabilidades* p_i de que se consulte cada clave k_i. Queremos minimizar entonces el *costo promedio* dado por $\sum_i p_i d_i$, donde d_i es la profundida a la que quedó la clave k_i.

(Esto es simplemente un promedio ponderado de las profundidades según la probabilidad que tienen. En términos simples, si una clave k_5 es 10 veces más probable de ser buscada que una clave k_{12}, entonces la profundidad de k_5 importa 10 veces más.)

Diseñe un algoritmo que compute el mejor costo promedio posible. El mismo debería también generar un árbol óptimo. (Nota: en este problema estamos ignorando el costo de las búsquedas de claves que no están en el árbol.)

22 (Optimizando – 2). Supongamos la siguiente variante del problema anterior:

- Nuestro árbol no tiene valores en los nodos internos, sólo en las hojas.
- Las claves k_i no están dadas, podemos generarlas a gusto (y por lo tanto no hay ningún orden que mantener).

¿Cómo resolvería este problema...?

23* (Fábrica de Alfajores). En una fábrica de alfajores, se cuenta con una cantidad grande N de los mismos, todos juntos en un contenedor. Se quiere separarlos a todos para empaquetarlos, pero sólo podemos hacerlo vía las siguiente operaciones:

- Agregar un alfajor al conjunto
- Quitar un alfajor del conjunto
- Si la cantidad es par, reducir la cantidad a la mitad

Se quiere saber, dado un $N < 2^{64}$, la cantidad mínima de pasos para conseguir un sólo alfajor. Por ejemplo, para $N = 31$, puede hacerse en 6 pasos. Para $N = 31415926535$, puede hacerse en 44.