Algoritmos y Estructura de datos III - Sección TQ – 1er Semestre - 2018
Prof. Cristian Cappo **- Técnicas de diseño de algoritmos**
10-MAYO-2018 – Tarea #9 – A entregar por Educa hasta el viernes **18/MAYO/2018** 23:55 hs. **TP=60**

Ejercicio #1 (5p)
Escriba un programa en Java utilizando la técnica DyV que dado un arreglo de números, retorne el mayor de los elementos. Analice su tiempo de ejecución con su ecuación de recurrencia, resuélvala. Compare su rendimiento con el algoritmo de fuerza bruta.

Ejercicio #2 (5p)
Escriba un método de exponenciación basada en la técnica de DyV basada en la fórmula de abajo. Escriba su ecuación de recurrencia, resuélvala y compare el tiempo de ejecución con el algoritmo de fuerza bruta. Genere la tabla de comparación.

$$a^n = \begin{cases} (a^{n/2})^2 & si \quad n\,es\ par\ y\ positivo \\ (a^{(n-1)/2})^2 . a & si \quad n\,es\ impar\ y\ mayor\ a\ 1 \\ a & si \quad n = 1 \end{cases}$$

Ejercicio #3 (10p)
Diseñe un método genérico en Java que implemente un algoritmo DyV de tiempo *O(n log n)* que reciba un arreglo de elementos comparables A, los ordene y retorne el número de ***inversiones***. Recuerde, dado el par A[i] y A[j] del arreglo A, una inversión si i < j y A[i] > A[j]. Fundamente el tiempo de ejecución solicitado.

Ejercicio #4 (10p)
Considere el problema de la multiplicación de dos enteros X e Y de *n* dígitos decimales. Construya un algoritmo utilizando la técnica de divide y vencerás teniendo en cuenta lo siguiente:
Dividir

X en A y B dígitos donde A son los dígitos altos y B los bajos entonces $X = A.10^{n/2} + B$
Y en C y D dígitos donde C son los dígitos altos y D los bajos entonces $Y = C.10^{n/2} + D$

Se sabe que

$XY = AC10^n + [(A-B)(D-C) + AC + BD]10^{n/2} + BD$

Entonces la recurrencia es:

T(n) = 1 *para n = 1*
T(n) = 3T(n/2) + O(n) *para n > 1*

a) Resuelva la fórmula de recurrencia. *1p*
b) Implemente el algoritmo en Java (Utilice la clase *BigInteger* para ello) *5p*
c) Compare esta técnica con la versión de multiplicación usada en la escuela (que también debe implementarla). Construya una tabla de comparación (en tiempo) entre ambas implementaciones. Indique a partir de que números se nota la diferencia de tiempo. *4p*

Ejercicio #5 (10p)
Utilizando un algoritmo voraz resuelva el problema de minimización de la cantidad de billetes de "vuelto". Implemente un algoritmo en Java para calcular la cantidad de billetes/monedas de cada denominación dado un monto, que se recibe como parámetro, presumiendo que se tiene una cantidad ilimitada de unidades de cada denominación. Considere que las denominaciones (enteras) también son un parámetro del problema recibidos en un arreglo.

```
int [] minimizarVuelto ( long monto, int [] denominaciones)
```

En caso de que reciba un monto que no es posible obtener con alguna combinación entonces generar una excepción.

¿Obtiene su solución siempre la cantidad óptima de billetes, es decir mínima? Explique porque si o porque no. Si no es óptima proponga un algoritmo basado en Programación Dinámica que lo encuentre. Hint: pruebe con denominaciones

Ejercicio #6 (10p)
Utilizando el código de multiplicación óptima de matrices presentada en el capítulo 15 de libro *Introduction to Algorithm de Cormen et al. 2nd Edition*. 2001.

Tradúzcalo a Java y complete la implementación de tal forma que pueda obtenerse una matriz resultante de multiplicar ***n*** matrices aplicando el algoritmo de optimización mencionado. Para ello debe diseñar un `TDA Matriz`. Además provea otro método independiente que retorne una cadena donde muestre la parentización resultante de la optimización. Una posible forma de aplicación ejemplo sería:

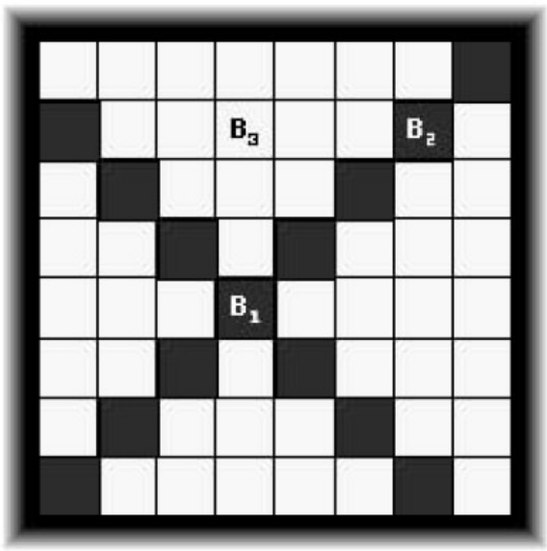
```
Matriz A, B, C, D, E, F;
Matriz R;
Matriz ListaMatrices [ ] = new Matriz [6];
String parentizacion;

ListaMatrices[0] = A;
ListaMatrices[1] = B;
ListaMatrices[2] = C;
     ..
R = Matriz.multiplicar ( ListaMatrices ) ;  // Quiero multiplicar A.B.C.D.E.F y dejar el
                                            // resultado en R, en ese orden.
                                            // Fijarse que debe controlar la compatibilidad
                                            // ListaMatrices puede implementar como un simple
                                               vector o como una clase propia.
                                            //
// Retornar la parentización, algo así como ((A1*(((A2*A3)*A4)*A5))*A6) (Para las 6 matrices)
parentización = Matriz.obtenerParentizacionOptima ( ListaMatrices ) ;

System.out.printf("\n%s\", parentizacion);
```

Ejercicio #7 (5p)

Un alfil es una pieza utilizada en el juego de ajedrez el cual solo puede tener movimientos en diagonal desde su posición actual. Dos alfiles se atacan uno con otro si uno de ellos esta en el camino del otro. En la figura de abajo, las posiciones del tablero de ajedrez pintadas de gris representan las posiciones alcanzables por el alfil B1 desde su posición actual. B1 y B2 se están atacando mutuamente, mientras que B1 y B3 no lo están. También los alfiles B2 y B3 no se encuentran en posición de ataque.



Diseñe un algoritmo de backtracking que dado dos números enteros n y k, determine la cantidad de formas que se puede colocar k alfiles en un tablero de n x n tal que ninguno de ellos se encuentren en posición de ataque.

Indique claramente las optimizaciones introducidas en su algoritmo para evitar una exploración completa. Indique una cota de su algoritmo en base a n y k.

Ejercicio #8 (5p)

Usted tiene una secuencia de ***n*** números positivos o negativos: $x_1, x_2, ..., x_n$. Tiene que seleccionar el subconjunto de números de suma máxima, sujeto a la restricción de que no puede escoger dos elementos que son adyacentes(esto es, si elige x_i no puede elegir x_{i-1} o x_{i+1}).

Diseñe una estrategia de tiempo polinomial para encontrar el subconjunto de máxima suma de acuerdo a la restricción impuesta.

En este sencillo ejemplo de números positivos, existen varias subsecuencias que se puede analizar aparte de los elementos individuales:

x1	x2	x3	x4	x5	x6	x7
7	8	9	3	2	4	8

Algunos ejemplos:

x1, x3 => 7 + 9 = *16*
x3, x5 => 9 + 2 = 11
x5, x7 => 2 + 8 = 10
x2, x4 => 8 + 3 = *11*
x4, x6 => 3 + 4 = 7
x1, x3, x5 => 7 + 9 + 2 = 18
x3, x5, x7 => 9 + 2 + 8 = *19*
x2, x4, x6 => 8 + 3 + 4 = 15
x1, x3, x5, x7 => 7 + 9 + 2 + 8 = 26 (es la subsecuencia de suma máxima para el ejemplo)