UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA	**Programación III** Código de asignatura (marcar con una X): Sistemas: __ 402048 (plan viejo) __ 532044 (plan nuevo) Gestión: __ 41204- (plan viejo) __ 542046 (plan nuevo)	**Prueba Presencial** **Original** Septiembre de 2004 Duración: **2 horas** Material permitido: **NINGUNO**

Apellidos: _____ DNI: _____

Nombre: _____ Centro donde entregó la práctica: _____ e-mail: _____

Cuestión 1 (2 puntos). ¿Qué significa que el tiempo de ejecución de un algoritmo está "en el orden exacto de f(n)"? Demostrar que $T(n)=n^3+9\cdot n^2\cdot\log(n)$ está en el orden exacto de n^3.

La definición formal es $\Theta(f(n)) = O(f(n)) \cap \Omega(f(n))$, y considerando que si

$\lim_{n\to\infty} f(n)/g(n) \in \Re^+$ entonces $f(n) \in \Theta(g(n))$

según la relación del apartado 3.3 (Ver p. 100 del texto base) y además aplicamos el teorema de L´Hopital (p. 38) obtenemos que:

$\lim_{n\to\infty} f(n)/g(n) = \lim_{n\to\infty} f'(n)/g'(n)$ con lo que:

$\lim_{n\to\infty} n^3 + 9n^2 \cdot \log(n) / n^3 = \lim_{n\to\infty} 3n^2 + 18n\cdot\log(n) + 9n / 3n^2 = \lim_{n\to\infty} 6n + 18\cdot\log(n) + 27 / 6n =$

*$\lim_{n\to\infty} 6n +18 / 6n = 1$ con lo que **$n^3 + 9n^2 \cdot \log(n) \in \Theta(n^3)$***

También será válido hallar las constantes c y d según la definición de la notación Theta (p.100)

Cuestión 2 (2 puntos). Implementar una versión recursiva de una función que tome un vector y le dé estructura de montículo.

A partir de una implementación recursiva en MODULA-2 de Flotar (P. ej.):

```
PROCEDURE Flotar(i: INTEGER; VAR T: TipoMonticulo);

VAR
  i_padre: INTEGER;

BEGIN
```

La resolución del problema debe incluir, por este orden:

1. Elección razonada del esquema algorítmico
2. Descripción del esquema usado e identificación con el problema
3. Estructuras de datos
4. Algoritmo completo a partir del refinamiento del esquema general
5. Estudio del coste

```
  i_padre := i DIV 2 ;


  IF (i > 1) AND T[i]<T[i_padre]) THEN
     Intercambia(T[i],T[i_padre]);
     Flotar(i_padre,T);
  END;

END Flotar;
```

Se puede implementar la creación de montículo como:

```
procedimiento crear_monticulo(T[1..n],i)

     si i>n
          devuelve(T)
     sino
          flotar(i,T)
          crear_monticulo(T,i+1)
     fsi
fprocedimiento
```

La llamada inicial será

```
crear_monticulo(T,2)
```

Otra versión pero en este caso sin usar funciones auxiliares es sustituir el código de flotar por su versión iterativa e insertarlo en la función anterior.

Cuestión 3 (1 punto). ¿En qué se diferencian los algoritmos de Prim y de Kruskal? Discute tanto los algoritmos como su complejidad en los casos peores de cada uno.

En primer lugar difieren en la forma de crear el camino mínimo. En el caso de Prim la solución es siempre un AEM y en el otro caso, lo son las componentes conexas pero sin referencia al grafo inicial, salvo al final del mismo.

En segundo lugar en términos de coste, el algoritmo de Kruskal requiere un tiempo que está en $\Theta(a \cdot \log n)$ *con* ***a*** *el número de aristas, por lo que en el caso peor si el grafo es denso y* ***a*** *se acerca a* n^2*, entonces es menos eficiente que el de Prim que es cuadrático, ocurriendo lo contrario para grafos dispersos.*

Problema (5 puntos). Sean dos vectores de caracteres. El primero se denomina *texto* y el segundo *consulta*, siendo éste de igual o menor longitud que el primero. Al vector *consulta* se le pueden aplicar, cuantas veces sea necesario, los siguientes tres tipos de operaciones, que añaden cada una un coste diferente a la *edición* de la consulta:

- *Intercambio* de dos caracteres consecutivos: coste de edición igual a 1
- *Sustitución* de un carácter por otro cualquiera: coste de edición igual a 2
- *Inserción* de un carácter cualquiera en una posición cualquiera: coste de edición igual a 3

Implementar una función que escriba la secuencia de operaciones con menor coste total que hay que realizar sobre el vector *consulta* para que coincida exactamente con el vector *texto*. La

función devolverá el coste total de estas operaciones, es decir, la suma de los costes asociados a cada operación realizada.

1. Elección razonada del esquema algorítmico

La solución es resultado de una secuencia de pasos o decisiones pero no se puede establecer un criterio óptimo de selección de cada decisión por lo que no puede ser un esquema voraz. Por tanto hay que realizar una exploración de las posibles soluciones y buscar la óptima. Debido a que hay un criterio de optimalidad, es decir, existe una función que decide si un nodo puede llevar a una solución mejor que la encontrada hasta el momento, el esquema adecuado es Ramificación y Poda.

2. Descripción del esquema

```
Función RamificaciónPoda (nodo_raíz) dev nodo
        inicializarSolución(Solución,valor_sol_actual);
        Montículo:=montículoVacío();
        cota:=acotar(nodo_raíz);
        poner((cota,nodo_raíz),Montículo);
        mientras no vacío(Montículo) hacer
                (cota,nodo):=quitarPrimero(Montículo);
                si es_mejor(cota, valor_sol_actual) entonces
                        si válido(nodo) entonces
                                /*cota es el valor real de la mejor solución hasta ahora*/
                                valor_sol_actual:=cota;
                                Solución:=nodo; fsi;
                        si no para cada hijo en compleciones(nodo) hacer
                                cota:=acotar(hijo);
                                poner((cota,hijo),Montículo); fpara; fsi;
                sino devolver Solución fsi;     /*termina el bucle, montículo no tiene mejores*/
        fmientras
        devolver Solución;
```

3. Estructuras de datos

- *Vector de texto*
- *Vector de consulta*
- *Montículo de mínimos en el que cada componente almacene una solución parcial (nodo) con su cota correspondiente.*
- *nodo=tupla*
  - *acciones: lista de Strings;*
  - *último_coincidente: cardinal;*
  - *long: cardinal;*
  - *coste: cardinal;*

4. Algoritmo completo

inicializarSolución

*La solución inicial debe tener un coste asociado mayor o igual que la solución final. De acuerdo con el problema, basta con realizar sustituciones de los caracteres de la consulta (coste=2*long_consulta) y realizar la inserción de los caracteres que faltan (coste=3*(long_texto-long_consulta)). Además tenemos que construir la solución inicial: la secuencia de sustituciones e inserciones.*

```
Función inicializarSolución(long_consulta, texto, long_texto, Solución, valor_sol_actual);
    valor_sol_actual:= 2*long_consulta+3*(long_texto-long_consulta);
    Solución:=listaVacia();
    para i desde 1 hasta long_consulta hacer
        añadir(Solución, "sustituir i por texto[i]");
    fpara;
    para i desde long_consulta+1 hasta long_texto hacer
        añadir(Solución, "insertar en i, texto[i]");
    fpara;
```

válido(nodo)

Un nodo será válido (como solución) si se han hecho coincidir los long_texto caracteres de texto. Para ello, vamos a ir haciendo coincidir la solución desde el principio del vector hasta el final. por tanto, un nodo será válido (aunque no sea la mejor solución todavía) si:

```
nodo.último_coincidente==long_texto
```

es_mejor(cota, valor_sol_actual)

Como se trata de encontrar la solución de menor coste, una cota será mejor que el valor de la solución actual si:

```
cota<valor_sol_actual
```

acotar(nodo)

Se trata de realizar una estimación que sea mejor o igual que la mejor solución que se puede alcanzar desde ese nodo. De esta manera sabremos que si, aún así, la estimación es peor que la solución actual, por esa rama no encontraremos nada mejor y se puede podar. La mejor estimación será sumar al coste ya acumulado, el menor coste que podría tener completar la solución: que el resto de caracteres de consulta coincidieran (coste=0) y realizar tantas inserciones como caracteres nos falten (coste=2(long_texto-nodo.long)). Es decir, acotar(nodo) es:*

```
nodo.coste+2*(long_texto-nodo.long)
```

compleciones(nodo)

Sabiendo que hasta nodo.último_coincidente todos los caracteres coinciden, se trata de considerar las posibilidades para que el siguiente carácter de la consulta coincida con el texto:

- *No hacer nada si ya coinciden de antemano.*
- *Intercambiarlo por el siguiente si éste coincide con el texto.*
- *Sustituirlo por el correspondiente del texto*
- *Insertar el correspondiente del texto y correr el resto del vector, siempre que la consulta no haya alcanzado el tamaño del texto.*

Si se da el primer caso, no es necesario generar el resto de compleciones porque no se puede encontrar una solución mejor con ninguna de las otras alternativas. Aún así, no pasaría nada si se incluyen. Sin embargo, para el resto de alternativas, no hay garantías

de que una permita encontrar mejor solución que otra, aunque el coste de la acción puntual sea menor. Por tanto, hay que incluir todas las alternativas.

```
Función compleciones(nodo, texto, long_texto, consulta) dev lista_nodos
    lista:=crearLista();
    si consulta[hijo.último_coincidente]==texto[hijo.último_coincidente] entonces
        hijo:=crearNodo();
        hijo.último_coincidente:=nodo.último_coincidente+1;
        hijo.acciones=nodo.acciones;
        hijo.coste=nodo.coste;
        hijo.long=nodo.long;
        añadir(lista, hijo);
    si no
        /* intercambio */
        si consulta[hijo.último_coincidente+1]==texto[hijo.último_coincidente]
        entonces
            intercambiar(consulta[hijo.último_coincidente],
                         consulta[hijo.último_coincidente+1]);
            hijo:=crearNodo();
            hijo.último_coincidente:=nodo.último_coincidente+1;
            hijo.acciones=nodo.acciones;
            añadir(hijo.acciones,"intercambiar consulta[hijo.último_coincidente]
                                  por consulta[hijo.último_coincidente+1]");
            hijo.coste=nodo.coste+1;
            hijo.long=nodo.long;
            añadir(lista, hijo);
        fsi;
        /* sustitución */
        hijo:=crearNodo();
        hijo.último_coincidente:=nodo.último_coincidente+1;
        hijo.acciones=nodo.acciones;
        insertar(hijo.acciones,"sustituir hijo.último_coincidente por
                                texto[hijo.último_coincidente]");
        hijo.coste=nodo.coste+2;
        hijo.long=nodo.long;
        añadir(lista, hijo);
        /* inserción */
        si (nodo.long<long_texto) entonces
            hijo:=crearNodo();
            hijo.último_coincidente:=nodo.último_coincidente+1;
            hijo.acciones=nodo.acciones;
            insertar(hijo.acciones,"insertar en hijo.último_coincidente,
                                    texto[hijo.último_coincidente]");
            hijo.coste=nodo.coste+3;
            hijo.long=nodo.long+1;
            añadir(lista, hijo);
        fsi;
    fsi;
    devolver lista;
```

La función principal quedaría entonces como sigue:

```
Función ajustar(consulta, long_consulta, texto, long_texto) dev coste
        inicializarSolución(long_consulta, texto, long_texto, Solución, valor_sol_actual);
        Montículo:=montículoVacío();
        nodo.acciones=listaVacia();
        nodo.último_coincidente=0;
        nodo.coste=0;
        nodo.long=long_consulta;
        cota:= nodo.coste+2*(long_texto-nodo.long);
        poner((cota,nodo),Montículo);
        mientras ¬vacío(Montículo) hacer
                (cota,nodo):=quitarPrimero(Montículo);
                si cota<valor_sol_actual entonces
                        si nodo.último_coincidente==long_texto entonces
                                valor_sol_actual:=cota;
                                solución:=nodo.acciones;
                        si no
                                para cada hijo en compleciones(nodo, texto, long_texto, consulta)
                                hacer
                                        cota:= nodo.coste+2*(long_texto-nodo.long);
                                        poner((cota,hijo),Montículo);
                                fpara;
                        fsi;
                sino
                        /* Ya no puede haber una solución mejor */
                        escribir(solución);
                        devolver valor_sol_actual;
                fsi;
        fmientras
        escribir(solucion);
        devolver valor_sol_actual;
```

5. Estudio del coste

En este caso únicamente podemos hallar una cota superior del coste del algoritmo por descripción del espacio de búsqueda. En el caso peor se generan 3 hijos por nodo hasta llegar a una profundidad de long_texto. Por tanto, el espacio a recorrer siempre será menor que 3^{long_texto}.