

Guía Análisis y Diseño de Algoritmos

Dra. Luz María Sánchez García

Ing. en Inteligencia Artificial / Lic. en Ciencia de Datos

Unidad I	<ul style="list-style-type: none">• Tipos de complejidad<ul style="list-style-type: none">○ Complejidad temporal○ Complejidad espacial• Notación Asintótica<ul style="list-style-type: none">○ Notación θ○ Notación O○ Notación Ω○ Notación o○ Notación ω• Funciones típicas para describir crecimientos asintóticos<ul style="list-style-type: none">○ Lineal○ Constante○ Logarítmica○ Lineal logarítmica○ Cuadrática○ Cúbica○ Exponencial○ Factorial• Algoritmos iterativos (búsqueda y ordenamiento)• Algoritmos iterativos vs recursivos
Unidad II	<ul style="list-style-type: none">• Divide y Vencerás<ul style="list-style-type: none">○ Problema del máximo subarreglo○ El algoritmo de Strassen○ Karatsuba○ Merge Sort y Quick Sort○ Búsqueda dicotómica○ Ecuaciones de recurrencia<ul style="list-style-type: none">▪ Método de Sustitución▪ Método de Iteraciones▪ El teorema Maestro y su demostración• Programación dinámica<ul style="list-style-type: none">○ El problema de corte de varilla○ El problema de multiplicación de una cadena de matrices○ Mochila 0,1• Greedy<ul style="list-style-type: none">○ El problema de la selección de actividades○ Códigos de Huffman○ El problema del cambio

Unidad III y IV	<ul style="list-style-type: none">• Análisis probabilístico y Algoritmos aleatorizados<ul style="list-style-type: none">○ Las Vegas○ Monte Carlo• NP-Compleitud<ul style="list-style-type: none">○ Problema del viajante (TSP)○ Problema de Satisfacibilidad Booleana (SAT)○ Problema del Coloreado de Grafos○ Problema del ciclo hamiltoniano○ Problema de la clique.
------------------------	--

Capítulo 1

LA COMPLEJIDAD DE LOS ALGORITMOS

1.1 INTRODUCCIÓN

En un sentido amplio, dado un problema y un dispositivo donde resolverlo, es necesario proporcionar un método preciso que lo resuelva, adecuado al dispositivo. A tal método lo denominamos *algoritmo*.

En el presente texto nos vamos a centrar en dos aspectos muy importantes de los algoritmos, como son su diseño y el estudio de su eficiencia.

El primero se refiere a la búsqueda de métodos o procedimientos, secuencias finitas de instrucciones adecuadas al dispositivo que disponemos, que permitan resolver el problema. Por otra parte, el segundo nos permite medir de alguna forma el coste (en tiempo y recursos) que consume un algoritmo para encontrar la solución y nos ofrece la posibilidad de comparar distintos algoritmos que resuelven un mismo problema.

Este capítulo está dedicado al segundo de estos aspectos: la eficiencia. En cuanto a las técnicas de diseño, que corresponden a los patrones fundamentales sobre los que se construyen los algoritmos que resuelven un gran número de problemas, se estudiarán en los siguientes capítulos.

1.2 EFICIENCIA Y COMPLEJIDAD

Una vez dispongamos de un algoritmo que funciona correctamente, es necesario definir criterios para medir su rendimiento o comportamiento. Estos criterios se centran principalmente en su simplicidad y en el uso eficiente de los recursos.

A menudo se piensa que un algoritmo sencillo no es muy eficiente. Sin embargo, la sencillez es una característica muy interesante a la hora de diseñar un algoritmo, pues facilita su verificación, el estudio de su eficiencia y su mantenimiento. De ahí que muchas veces prime la simplicidad y legibilidad del código frente a alternativas más crípticas y eficientes del algoritmo. Este hecho se pondrá de manifiesto en varios de los ejemplos mostrados a lo largo de este libro, en donde profundizaremos más en este compromiso.

Respecto al uso eficiente de los recursos, éste suele medirse en función de dos parámetros: el *espacio*, es decir, memoria que utiliza, y el *tiempo*, lo que tarda en ejecutarse. Ambos representan los costes que supone encontrar la solución al problema planteado mediante un algoritmo. Dichos parámetros van a servir además para comparar algoritmos entre sí, permitiendo determinar el más adecuado de

entre varios que solucionan un mismo problema. En este capítulo nos centraremos solamente en la eficiencia temporal.

El tiempo de ejecución de un algoritmo va a depender de diversos factores como son: los datos de entrada que le suministremos, la calidad del código generado por el compilador para crear el programa objeto, la naturaleza y rapidez de las instrucciones máquina del procesador concreto que ejecute el programa, y la complejidad intrínseca del algoritmo. Hay dos estudios posibles sobre el tiempo:

1. Uno que proporciona una medida *teórica* (a priori), que consiste en obtener una función que acote (por arriba o por abajo) el tiempo de ejecución del algoritmo para unos valores de entrada dados.
2. Y otro que ofrece una medida *real* (a posteriori), consistente en medir el tiempo de ejecución del algoritmo para unos valores de entrada dados y en un ordenador concreto.

Ambas medidas son importantes puesto que, si bien la primera nos ofrece estimaciones del comportamiento de los algoritmos de forma independiente del ordenador en donde serán implementados y sin necesidad de ejecutarlos, la segunda representa las medidas reales del comportamiento del algoritmo. Estas medidas son *funciones temporales* de los datos de entrada.

Entendemos por *tamaño de la entrada* el número de componentes sobre los que se va a ejecutar el algoritmo. Por ejemplo, la dimensión del vector a ordenar o el tamaño de las matrices a multiplicar.

La unidad de tiempo a la que debe hacer referencia estas medidas de eficiencia no puede ser expresada en segundos o en otra unidad de tiempo concreta, pues no existe un ordenador estándar al que puedan hacer referencia todas las medidas. Denotaremos por $T(n)$ el tiempo de ejecución de un algoritmo para una entrada de tamaño n .

Teóricamente $T(n)$ debe indicar el número de instrucciones ejecutadas por un ordenador idealizado. Debemos buscar por tanto medidas simples y abstractas, independientes del ordenador a utilizar. Para ello es necesario acotar de alguna forma la diferencia que se puede producir entre distintas implementaciones de un mismo algoritmo, ya sea del mismo código ejecutado por dos máquinas de distinta velocidad, como de dos códigos que implementen el mismo método. Esta diferencia es la que acota el siguiente principio:

Principio de Invarianza

Dado un algoritmo y dos implementaciones suyas I_1 e I_2 , que tardan $T_1(n)$ y $T_2(n)$ segundos respectivamente, el *Principio de Invarianza* afirma que existe una constante real $c > 0$ y un número natural n_0 tales que para todo $n \geq n_0$ se verifica que $T_1(n) \leq cT_2(n)$.

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.

Con esto podemos definir sin problemas que un algoritmo tarda un tiempo *del orden de* $T(n)$ si existen una constante real $c > 0$ y una implementación I del algoritmo que tarda menos que $cT(n)$, para todo n tamaño de la entrada.

Dos factores a tener muy en cuenta son la constante multiplicativa y el n_0 para los que se verifican las condiciones, pues si bien a priori un algoritmo de orden cuadrático es mejor que uno de orden cúbico, en el caso de tener dos algoritmos cuyos tiempos de ejecución son $10^6 n^2$ y $5n^3$ el primero sólo será mejor que el segundo para tamaños de la entrada superiores a 200.000.

También es importante hacer notar que el comportamiento de un algoritmo puede cambiar notablemente para diferentes entradas (por ejemplo, lo ordenados que se encuentren ya los datos a ordenar). De hecho, para muchos programas el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ésta. Así suelen estudiarse tres casos para un mismo algoritmo: *caso peor*, *caso mejor* y *caso medio*.

El caso mejor corresponde a la traza (secuencia de sentencias) del algoritmo que realiza menos instrucciones. Análogamente, el caso peor corresponde a la traza del algoritmo que realiza más instrucciones. Respecto al caso medio, corresponde a la traza del algoritmo que realiza un número de instrucciones igual a la esperanza matemática de la variable aleatoria definida por todas las posibles trazas del algoritmo para un tamaño de la entrada dado, con las probabilidades de que éstas ocurran para esa entrada.

Es muy importante destacar que esos casos corresponden a un tamaño de la entrada dado, puesto que es un error común confundir el caso mejor con el que menos instrucciones realiza en cualquier caso, y por lo tanto contabilizar las instrucciones que hace para $n = 1$.

A la hora de medir el tiempo, siempre lo haremos en función del *número de operaciones elementales* que realiza dicho algoritmo, entendiendo por operaciones elementales (en adelante OE) aquellas que el ordenador realiza en tiempo acotado por una constante. Así, consideraremos OE las operaciones aritméticas básicas, asignaciones a variables de tipo predefinido por el compilador, los saltos (llamadas a funciones y procedimientos, retorno desde ellos, etc.), las comparaciones lógicas y el acceso a estructuras indexadas básicas, como son los vectores y matrices. Cada una de ellas contabilizará como 1 OE.

Resumiendo, el tiempo de ejecución de un algoritmo va a ser una función que mide el número de operaciones elementales que realiza el algoritmo para un tamaño de entrada dado.

En general, es posible realizar el estudio de la complejidad de un algoritmo sólo en base a un conjunto reducido de sentencias, aquellas que caracterizan que el algoritmo sea lento o rápido en el sentido que nos interesa. También es posible distinguir entre los tiempos de ejecución de las diferentes operaciones elementales, lo cual es necesario a veces por las características específicas del ordenador (por ejemplo, se podría considerar que las operaciones $+$ y \div presentan complejidades diferentes debido a su implementación). Sin embargo, en este texto tendremos en cuenta, a menos que se indique lo contrario, todas las operaciones elementales del lenguaje, y supondremos que sus tiempos de ejecución son todos iguales.

Para hacer un estudio del tiempo de ejecución de un algoritmo para los tres casos citados comenzaremos con un ejemplo concreto. Supongamos entonces que disponemos de la definición de los siguientes tipos y constantes:

```
CONST n =...; (* num. maximo de elementos de un vector *);
TYPE vector = ARRAY [1..n] OF INTEGER;
```

y de un algoritmo cuya implementación en Modula-2 es:

```
PROCEDURE Buscar(VAR a:vector;c:INTEGER):CARDINAL;
  VAR j:CARDINAL;
BEGIN
  j:=1;                                (* 1 *)
  WHILE (a[j]<c) AND (j<n) DO          (* 2 *)
    j:=j+1                             (* 3 *)
  END;                                  (* 4 *)
  IF a[j]=c THEN                       (* 5 *)
    RETURN j                           (* 6 *)
  ELSE RETURN 0                        (* 7 *)
  END                                  (* 8 *)
END Buscar;
```

Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecuta 1 OE (una asignación).
- En la línea (2) se efectúa la condición del bucle, con un total de 4 OE (dos comparaciones, un acceso al vector, y un *AND*).
- La línea (3) está compuesta por un incremento y una asignación (2 OE).
- La línea (5) está formada por una condición y un acceso al vector (2 OE).
- La línea (6) contiene un *RETURN* (1 OE) si la condición se cumple.
- La línea (7) contiene un *RETURN* (1 OE), cuando la condición del *IF* anterior es falsa.

Obsérvese cómo no se contabiliza la copia del vector a la pila de ejecución del programa, pues se pasa por referencia y no por valor (está declarado como un argumento *VAR*, aunque no se modifique dentro de la función). En caso de pasarlo por valor, necesitaríamos tener en cuenta el coste que esto supone (un incremento de n OE). Con esto:

- En el *caso mejor* para el algoritmo, se efectuará la línea (1) y de la línea (2) sólo la primera mitad de la condición, que supone 2 OE (suponemos que las expresiones se evalúan de izquierda a derecha, y con “cortocircuito”, es decir, una expresión lógica deja de ser evaluada en el momento que se conoce su valor, aunque no hayan sido evaluados todos sus términos). Tras ellas la función acaba ejecutando las líneas (5) a (7). En consecuencia, $T(n)=1+2+3=6$.
- En el *caso peor*, se efectúa la línea (1), el bucle se repite $n-1$ veces hasta que se cumple la segunda condición, después se efectúa la condición de la línea (5) y la función acaba al ejecutarse la línea (7). Cada iteración del bucle está compuesta por las líneas (2) y (3), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. Por tanto

$$T(n) = 1 + \left(\left(\sum_{i=1}^{n-1} (4+2) \right) + 4 \right) + 2 + 1 = 6n + 2.$$

- En el *caso medio*, el bucle se ejecutará un número de veces entre 0 y $n-1$, y vamos a suponer que cada una de ellas tiene la misma probabilidad de suceder. Como existen n posibilidades (puede que el número buscado no esté) suponemos a priori que son equiprobables y por tanto cada una tendrá una probabilidad asociada de $1/n$. Con esto, el número medio de veces que se efectuará el bucle es de

$$\sum_{i=0}^{n-1} i \frac{1}{n} = \frac{n-1}{2}.$$

Tenemos pues que

$$T(n) = 1 + \left(\left(\sum_{i=1}^{(n-1)/2} (4+2) \right) + 2 \right) + 2 + 1 = 3n + 3.$$

Es importante observar que no es necesario conocer el propósito del algoritmo para analizar su tiempo de ejecución y determinar sus casos mejor, peor y medio, sino que basta con estudiar su código. Suele ser un error muy frecuente el determinar tales casos basándose sólo en la funcionalidad para la que el algoritmo fue concebido, olvidando que es el código implementado el que los determina.

En este caso, un examen más detallado de la función (¡y no de su nombre!) nos muestra que tras su ejecución, la función devuelve la posición de un entero dado c dentro de un vector ordenado de enteros, devolviendo 0 si el elemento no está en el vector. Lo que acabamos de probar es que su caso mejor se da cuando el elemento está en la primera posición del vector. El caso peor se produce cuando el elemento no está en el vector, y el caso medio ocurre cuando consideramos equiprobables cada una de las posiciones en las que puede encontrarse el elemento dentro del vector (incluyendo la posición especial 0, que indica que el elemento a buscar no se encuentra en el vector).

1.2.1 Reglas generales para el cálculo del número de OE

La siguiente lista presenta un conjunto de reglas generales para el cálculo del número de OE, siempre considerando el peor caso. Estas reglas definen el número de OE de cada estructura básica del lenguaje, por lo que el número de OE de un algoritmo puede hacerse por inducción sobre ellas.

- Vamos a considerar que el tiempo de una OE es, por definición, de orden 1. La constante c que menciona el Principio de Invarianza dependerá de la implementación particular, pero nosotros supondremos que vale 1.
- El tiempo de ejecución de una secuencia consecutiva de instrucciones se calcula sumando los tiempos de ejecución de cada una de las instrucciones.

- El tiempo de ejecución de la sentencia “CASE C OF $v_1:S_1|v_2:S_2|\dots|v_n:S_n$ END;” es $T = T(C) + \max\{T(S_1), T(S_2), \dots, T(S_n)\}$. Obsérvese que $T(C)$ incluye el tiempo de comparación con v_1, v_2, \dots, v_n .
- El tiempo de ejecución de la sentencia “IF C THEN S1 ELSE S2 END;” es $T = T(C) + \max\{T(S_1), T(S_2)\}$.
- El tiempo de ejecución de un bucle de sentencias “WHILE C DO S END;” es $T = T(C) + (n^\circ \text{ iteraciones}) * (T(S) + T(C))$. Obsérvese que tanto $T(C)$ como $T(S)$ pueden variar en cada iteración, y por tanto habrá que tenerlo en cuenta para su cálculo.
- Para calcular el tiempo de ejecución del resto de sentencias iterativas (*FOR*, *REPEAT*, *LOOP*) basta expresarlas como un bucle *WHILE*. A modo de ejemplo, el tiempo de ejecución del bucle:

```
FOR i:=1 TO n DO
  S
END;
```

puede ser calculado a partir del bucle equivalente:

```
i:=1;
WHILE i<=n DO
  S; INC(i)
END;
```

- El tiempo de ejecución de una llamada a un procedimiento o función $F(P_1, P_2, \dots, P_n)$ es 1 (por la llamada), más el tiempo de evaluación de los parámetros P_1, P_2, \dots, P_n , más el tiempo que tarde en ejecutarse F , esto es, $T = 1 + T(P_1) + T(P_2) + \dots + T(P_n) + T(F)$. No contabilizamos la copia de los argumentos a la pila de ejecución, salvo que se trate de estructuras complejas (registros o vectores) que se pasan por valor. En este caso contabilizaremos tantas OE como valores simples contenga la estructura. El paso de parámetros por referencia, por tratarse simplemente de punteros, no contabiliza tampoco.
- El tiempo de ejecución de las llamadas a procedimientos recursivos va a dar lugar a ecuaciones en recurrencia, que veremos posteriormente.
- También es necesario tener en cuenta, cuando el compilador las incorpore, las optimizaciones del código y la forma de evaluación de las expresiones, que pueden ocasionar “cortocircuitos” o realizarse de forma “perezosa” (*lazy*). En el presente trabajo supondremos que no se realizan optimizaciones, que existe el cortocircuito y que no existe evaluación perezosa.

1.3 COTAS DE COMPLEJIDAD. MEDIDAS ASINTÓTICAS

Una vez vista la forma de calcular el tiempo de ejecución T de un algoritmo, nuestro propósito es intentar clasificar dichas funciones de forma que podamos compararlas. Para ello, vamos a definir clases de equivalencia, correspondientes a las funciones que “crecen de la misma forma”.

En las siguientes definiciones \mathbf{N} denotará el conjunto de los números naturales y \mathbf{R} el de los reales.

1.3.1 Cota Superior. Notación O

Dada una función f , queremos estudiar aquellas funciones g que a lo sumo crecen tan deprisa como f . Al conjunto de tales funciones se le llama cota superior de f y lo denominamos $O(f)$. Conociendo la cota superior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden superior al de la cota.

Definición 1.1

Sea $f: \mathbf{N} \rightarrow [0, \infty)$. Se define el conjunto de funciones de orden O (Omicron) de f como:

$$O(f) = \{g: \mathbf{N} \rightarrow [0, \infty) \mid \exists c \in \mathbf{R}, c > 0, \exists n_0 \in \mathbf{N} \bullet g(n) \leq cf(n) \quad \forall n \geq n_0\}.$$

Diremos que una función $t: \mathbf{N} \rightarrow [0, \infty)$ es de orden O de f si $t \in O(f)$.

Intuitivamente, $t \in O(f)$ indica que t está acotada superiormente por algún múltiplo de f . Normalmente estaremos interesados en la menor función f tal que t pertenezca a $O(f)$.

En el ejemplo del algoritmo *Buscar* analizado anteriormente obtenemos que su tiempo de ejecución en el mejor caso es $O(1)$, mientras que sus tiempos de ejecución para los casos peor y medio son $O(n)$.

Propiedades de O

Veamos las propiedades de la cota superior. La demostración de todas ellas se obtiene aplicando la definición 1.1.

1. Para cualquier función f se tiene que $f \in O(f)$.
2. $f \in O(g) \Rightarrow O(f) \subset O(g)$.
3. $O(f) = O(g) \Leftrightarrow f \in O(g)$ y $g \in O(f)$.
4. Si $f \in O(g)$ y $g \in O(h) \Rightarrow f \in O(h)$.
5. Si $f \in O(g)$ y $f \in O(h) \Rightarrow f \in O(\min(g, h))$.
6. Regla de la suma: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 + f_2 \in O(\max(g, h))$.
7. Regla del producto: Si $f_1 \in O(g)$ y $f_2 \in O(h) \Rightarrow f_1 \cdot f_2 \in O(g \cdot h)$.
8. Si existe $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
 - a) Si $k \neq 0$ y $k < \infty$ entonces $O(f) = O(g)$.
 - b) Si $k = 0$ entonces $f \in O(g)$, es decir, $O(f) \subset O(g)$, pero sin embargo se verifica que $g \notin O(f)$.

Obsérvese la importancia que tiene el que exista tal límite, pues si no existiese (o fuera infinito) no podría realizarse tal afirmación, como veremos en la resolución de los problemas de este capítulo.

De las propiedades anteriores se deduce que la relación \sim_O , definida por $f \sim_O g$ si y sólo si $O(f) = O(g)$, es una relación de equivalencia. Siempre escogeremos el representante más sencillo para cada clase; así los órdenes de complejidad constante serán expresados por $O(1)$, los lineales por $O(n)$, etc.

1.3.2 Cota Inferior. Notación Ω

Dada una función f , queremos estudiar aquellas funciones g que a lo sumo crecen tan lentamente como f . Al conjunto de tales funciones se le llama cota inferior de f y lo denominamos $\Omega(f)$. Conociendo la cota inferior de un algoritmo podemos asegurar que, en ningún caso, el tiempo empleado será de un orden inferior al de la cota.

Definición 1.2

Sea $f: \mathbf{N} \rightarrow [0, \infty)$. Se define el conjunto de funciones de orden Ω (Omega) de f como:

$$\Omega(f) = \{g: \mathbf{N} \rightarrow [0, \infty) \mid \exists c \in \mathbf{R}, c > 0, \exists n_0 \in \mathbf{N} \bullet g(n) \geq cf(n) \quad \forall n \geq n_0\}.$$

Diremos que una función $t: \mathbf{N} \rightarrow [0, \infty)$ es de orden Ω de f si $t \in \Omega(f)$.

Intuitivamente, $t \in \Omega(f)$ indica que t está acotada inferiormente por algún múltiplo de f . Normalmente estaremos interesados en la mayor función f tal que t pertenezca a $\Omega(f)$, a la que denominaremos su cota inferior.

Obtener buenas cotas inferiores es en general muy difícil, aunque siempre existe una cota inferior trivial para cualquier algoritmo: al menos hay que leer los datos y luego escribirlos, de forma que ésa sería una primera cota inferior. Así, para ordenar n números una cota inferior sería n , y para multiplicar dos matrices de orden n sería n^2 ; sin embargo, los mejores algoritmos conocidos son de órdenes $n \log n$ y $n^{2.8}$ respectivamente.

Propiedades de Ω

Veamos las propiedades de la cota inferior Ω . La demostración de todas ellas se obtiene de forma simple aplicando la definición 1.2.

1. Para cualquier función f se tiene que $f \in \Omega(f)$.
2. $f \in \Omega(g) \Rightarrow \Omega(f) \subset \Omega(g)$.
3. $\Omega(f) = \Omega(g) \Leftrightarrow f \in \Omega(g)$ y $g \in \Omega(f)$.
4. Si $f \in \Omega(g)$ y $g \in \Omega(h) \Rightarrow f \in \Omega(h)$.
5. Si $f \in \Omega(g)$ y $f \in \Omega(h) \Rightarrow f \in \Omega(\max(g, h))$.
6. Regla de la suma: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 + f_2 \in \Omega(g + h)$.

7. Regla del producto: Si $f_1 \in \Omega(g)$ y $f_2 \in \Omega(h) \Rightarrow f_1 \cdot f_2 \in \Omega(g \cdot h)$.
8. Si existe $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
- Si $k \neq 0$ y $k < \infty$ entonces $\Omega(f) = \Omega(g)$.
 - Si $k = 0$ entonces $g \in \Omega(f)$, es decir, $\Omega(g) \subset \Omega(f)$, pero sin embargo se verifica que $f \notin \Omega(g)$.

De las propiedades anteriores se deduce que la relación \sim_Ω , definida por $f \sim_\Omega g$ si y sólo si $\Omega(f) = \Omega(g)$, es una relación de equivalencia. Al igual que hacíamos para el caso de la cota superior O , siempre escogeremos el representante más sencillo para cada clase. Así los órdenes de complejidad Ω constante serán expresados por $\Omega(1)$, los lineales por $\Omega(n)$, etc.

1.3.3 Orden Exacto. Notación Θ

Como última cota asintótica, definiremos los conjuntos de funciones que crecen asintóticamente de la misma forma.

Definición 1.3

Sea $f: \mathbf{N} \rightarrow [0, \infty)$. Se define el conjunto de funciones de orden Θ (Theta) de f como:

$$\Theta(f) = O(f) \cap \Omega(f)$$

o, lo que es igual:

$$\Theta(f) = \{g: \mathbf{N} \rightarrow [0, \infty) \mid \exists c, d \in \mathbf{R}, c, d > 0, \exists n_0 \in \mathbf{N} \cdot cf(n) \leq g(n) \leq df(n) \forall n \geq n_0\}.$$

Diremos que una función $t: \mathbf{N} \rightarrow [0, \infty)$ es de orden Θ de f si $t \in \Theta(f)$.

Intuitivamente, $t \in \Theta(f)$ indica que t está acotada tanto superior como inferiormente por múltiplos de f , es decir, que t y f crecen de la misma forma.

Propiedades de Θ

Veamos las propiedades de la cota exacta. La demostración de todas ellas se obtiene también de forma simple aplicando la definición 1.3 y las propiedades de O y Ω .

- Para cualquier función f se tiene que $f \in \Theta(f)$.
- $f \in \Theta(g) \Rightarrow \Theta(f) = \Theta(g)$.
- $\Theta(f) = \Theta(g) \Leftrightarrow f \in \Theta(g)$ y $g \in \Theta(f)$.
- Si $f \in \Theta(g)$ y $g \in \Theta(h) \Rightarrow f \in \Theta(h)$.
- Regla de la suma: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 + f_2 \in \Theta(\max(g, h))$.

6. Regla del producto: Si $f_1 \in \Theta(g)$ y $f_2 \in \Theta(h) \Rightarrow f_1 \cdot f_2 \in \Theta(g \cdot h)$.
7. Si existe $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = k$, dependiendo de los valores que tome k obtenemos:
 - a) Si $k \neq 0$ y $k < \infty$ entonces $\Theta(f) = \Theta(g)$.
 - b) Si $k = 0$ los órdenes exactos de f y g son distintos.

1.3.4 Observaciones sobre las cotas asintóticas

1. La utilización de las cotas asintóticas para comparar funciones de tiempo de ejecución se basa en la hipótesis de que son suficientes para decidir el mejor algoritmo, prescindiendo de las constantes de proporcionalidad. Sin embargo, esta hipótesis puede no ser cierta cuando el tamaño de la entrada es pequeño.
2. Para un algoritmo dado se pueden obtener tres funciones que miden su tiempo de ejecución, que corresponden a sus casos mejor, medio y peor, y que denominaremos respectivamente $T_m(n)$, $T_{1/2}(n)$ y $T_p(n)$. Para cada una de ellas podemos dar tres cotas asintóticas de crecimiento, por lo que se obtiene un total de nueve cotas para el algoritmo.
3. Para simplificar, dado un algoritmo diremos que su orden de complejidad es $O(f)$ si su tiempo de ejecución para el peor caso es de orden O de f , es decir, $T_p(n)$ es de orden $O(f)$. De forma análoga diremos que su orden de complejidad para el mejor caso es $\Omega(g)$ si su tiempo de ejecución para el mejor caso es de orden Ω de g , es decir, $T_m(n)$ es de orden $\Omega(g)$.
4. Por último, diremos que un algoritmo es de orden exacto $\Theta(f)$ si su tiempo de ejecución en el caso medio $T_{1/2}(n)$ es de este orden.

1.4 RESOLUCIÓN DE ECUACIONES EN RECURRENCIA

En las secciones anteriores hemos descrito cómo determinar el tiempo de ejecución de un algoritmo a partir del cómputo de sus operaciones elementales (OE). En general, este cómputo se reduce a un mero ejercicio de cálculo. Sin embargo, para los algoritmos recursivos nos vamos a encontrar con una dificultad añadida, pues la función que establece su tiempo de ejecución viene dada por una ecuación en recurrencia, es decir, $T(n) = E(n)$, en donde en la expresión E aparece la propia función T .

Resolver tal tipo de ecuaciones consiste en encontrar una expresión no recursiva de T , y por lo general no es una labor fácil. Lo que veremos en esta sección es cómo se pueden resolver algunos tipos concretos de ecuaciones en recurrencia, que son las que se dan con más frecuencia al estudiar el tiempo de ejecución de los algoritmos desarrollados según las técnicas aquí presentadas.

1.4.1 Recurrencias homogéneas

Son de la forma:

$$a_0 T(n) + a_1 T(n-1) + a_2 T(n-2) + \dots + a_k T(n-k) = 0$$

donde los coeficientes a_i son números reales, y k es un número natural entre 1 y n . Para resolverlas vamos a buscar soluciones que sean combinaciones de funciones exponenciales de la forma:

$$T(n) = c_1 p_1(n) r_1^n + c_2 p_2(n) r_2^n + \dots + c_k p_k(n) r_k^n = \sum_{i=1}^k c_i p_i(n) r_i^n,$$

donde los valores c_1, c_2, \dots, c_n y r_1, r_2, \dots, r_n son números reales, y $p_1(n), \dots, p_k(n)$ son polinomios en n con coeficientes reales. Si bien es cierto que estas ecuaciones podrían tener soluciones más complejas que éstas, se conjetura que serían del mismo orden y por tanto no nos ocuparemos de ellas.

Para resolverlas haremos el cambio $x^n = T(n)$, con lo cual obtenemos la *ecuación característica* asociada:

$$a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_k = 0.$$

Llamemos r_1, r_2, \dots, r_k a sus raíces, ya sean reales o complejas. Dependiendo del orden de multiplicidad de tales raíces, pueden darse los dos siguientes casos.

Caso 1: Raíces distintas

Si todas las raíces de la ecuación característica son distintas, esto es, $r_i \neq r_j$, si $i \neq j$, entonces la solución de la ecuación en recurrencia viene dada por la expresión:

$$T(n) = c_1 r_1^n + c_2 r_2^n + \dots + c_k r_k^n = \sum_{i=1}^k c_i r_i^n$$

donde los coeficientes c_i se determinan a partir de las condiciones iniciales.

Como ejemplo, veamos lo que ocurre para la ecuación en recurrencia definida para la sucesión de Fibonacci:

$$T(n) = T(n-1) + T(n-2), \quad n \geq 2$$

con las condiciones iniciales $T(0) = 0$, $T(1) = 1$. Haciendo el cambio $x^2 = T(n)$ obtenemos su ecuación característica $x^2 = x + 1$, o lo que es igual, $x^2 - x - 1 = 0$, cuyas raíces son:

$$r_1 = \frac{1 + \sqrt{5}}{2}, \quad r_2 = \frac{1 - \sqrt{5}}{2}$$

y por tanto

$$T(n) = c_1 \left(\frac{1 + \sqrt{5}}{2} \right)^n + c_2 \left(\frac{1 - \sqrt{5}}{2} \right)^n.$$

Para calcular las constantes c_1 y c_2 necesitamos utilizar las condiciones iniciales de la ecuación original, obteniendo:

$$\left. \begin{aligned} T(0) &= c_1 \left(\frac{1+\sqrt{5}}{2} \right)^0 + c_2 \left(\frac{1-\sqrt{5}}{2} \right)^0 = c_1 + c_2 = 0 \\ T(1) &= c_1 \left(\frac{1+\sqrt{5}}{2} \right)^1 + c_2 \left(\frac{1-\sqrt{5}}{2} \right)^1 = 1 \end{aligned} \right\} \Rightarrow c_1 = -c_2 = \frac{1}{\sqrt{5}}.$$

Sustituyendo entonces en la ecuación anterior, obtenemos

$$T(n) = \frac{1}{\sqrt{5}} \left(\frac{1+\sqrt{5}}{2} \right)^n - \frac{1}{\sqrt{5}} \left(\frac{1-\sqrt{5}}{2} \right)^n \in O(\varphi^n).$$

Caso 2: Raíces con multiplicidad mayor que 1

Supongamos que alguna de las raíces (p.e. r_1) tiene multiplicidad $m > 1$. Entonces la ecuación característica puede ser escrita en la forma

$$(x - r_1)^m (x - r_2) \dots (x - r_{k-m+1})$$

en cuyo caso la solución de la ecuación en recurrencia viene dada por la expresión:

$$T(n) = \sum_{i=1}^m c_i n^{i-1} r_1^n + \sum_{i=m+1}^k c_i r_{i-m+1}^n$$

donde los coeficientes c_i se determinan a partir de las condiciones iniciales.

Veamos un ejemplo en el que la ecuación en recurrencia es:

$$T(n) = 5T(n-1) - 8T(n-2) + 4T(n-3), \quad n \geq 2$$

con las condiciones iniciales $T(k) = k$ para $k = 0, 1, 2$. La ecuación característica que se obtiene es $x^3 - 5x^2 + 8x - 4 = 0$, o lo que es igual $(x-2)^2(x-1) = 0$ y por tanto,

$$T(n) = c_1 2^n + c_2 n 2^n + c_3 1^n.$$

De las condiciones iniciales obtenemos $c_1 = 2$, $c_2 = -1/2$ y $c_3 = -2$, por lo que

$$T(n) = 2^{n+1} - n 2^{n-1} - 2 \in \Theta(n 2^n).$$

Este caso puede ser generalizado de la siguiente forma. Si r_1, r_2, \dots, r_k son las raíces de la ecuación característica de una ecuación en recurrencia homogénea, cada una de multiplicidad m_i , esto es, si la ecuación característica puede expresarse como:

$$(x - r_1)^{m_1} (x - r_2)^{m_2} \dots (x - r_k)^{m_k} = 0,$$

entonces la solución a la ecuación en recurrencia viene dada por la expresión:

$$T(n) = \sum_{i=1}^{m_1} c_{1i} n^{i-1} r_1^n + \sum_{i=1}^{m_2} c_{2i} n^{i-1} r_2^n + \dots + \sum_{i=1}^{m_k} c_{ki} n^{i-1} r_k^n.$$

1.4.2 Recurrencias no homogéneas

Consideremos una ecuación de la forma:

$$a_0 T(n) + a_1 T(n-1) + \dots + a_k T(n-k) = b^n p(n)$$

donde los coeficientes a_i y b son números reales, y $p(n)$ es un polinomio en n de grado d . Una primera idea para resolver la ecuación es manipularla para convertirla en homogénea, como muestra el siguiente ejemplo.

Sea la ecuación $T(n) - 2T(n-1) = 3^n$ para $n \geq 2$, con las condiciones iniciales $T(0) = 0$ y $T(1) = 1$. En este caso $b = 3$ y $p(n) = 1$, polinomio en n de grado 0.

Podemos escribir la ecuación de dos formas distintas. En primer lugar, para $n+1$ tenemos que

$$T(n+1) - 2T(n) = 3^{n+1}.$$

Pero si multiplicamos por 3 la ecuación original obtenemos:

$$3T(n) - 6T(n-1) = 3^{n+1}$$

Restando ambas ecuaciones, conseguimos

$$T(n+1) - 5T(n) + 6T(n-1) = 0,$$

que resulta ser una ecuación homogénea cuya solución, aplicando lo visto anteriormente, es

$$T(n) = 3^n - 2^n \in \Theta(3^n).$$

Estos cambios son, en general, difíciles de ver. Afortunadamente, para este tipo de ecuaciones también existe una fórmula general para resolverlas, buscando sus soluciones entre las funciones que son combinaciones lineales de exponenciales, en donde se demuestra que la ecuación característica es de la forma:

$$(a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_k)(x - b)^{d+1} = 0,$$

lo que permite resolver el problema de forma similar a los casos anteriores.

Como ejemplo, veamos cómo se resuelve la ecuación en recurrencia que plantea el algoritmo de las torres de Hanoi:

$$T(n) = 2T(n-1) + n.$$

Su ecuación característica es entonces $(x-2)(x-1)^2 = 0$, y por tanto

$$T(n) = c_1 2^n + c_2 1^n + c_3 n 1^n \in \Theta(2^n).$$

Generalizando este proceso, supongamos ahora una ecuación de la forma:

$$a_0 T(n) + a_1 T(n-1) + \dots + a_k T(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + \dots + b_s^n p_s(n)$$

donde como en el caso anterior, los coeficientes a_i y b_i son números reales y $p_i(n)$ son polinomios en n de grado d_i . En este caso también existe una forma general de la solución, en donde se demuestra que la ecuación característica es:

$$(a_0 x^k + a_1 x^{k-1} + a_2 x^{k-2} + \dots + a_k)(x - b_1)^{d_1+1} (x - b_2)^{d_2+1} \dots (x - b_s)^{d_s+1} = 0.$$

Como ejemplo, supongamos la ecuación

$$T(n) = 2T(n-1) + n + 2^n, n \geq 1,$$

con la condición inicial $T(0) = 1$. En este caso tenemos que $b_1 = 1$, $p_1(n) = n$, $b_2 = 2$ y $p_2(n) = 1$, por lo que su ecuación característica es $(x-2)^2(x-1)^2 = 0$, lo que da lugar a la expresión final de $T(n)$:

$$T(n) = -2 - n + 2^{n+1} + n2^n \in \Theta(n2^n).$$

1.4.3 Cambio de Variable

Esta técnica se aplica cuando n es potencia de un número real a , esto es, $n = a^k$. Sea por ejemplo, para el caso $a = 2$, la ecuación $T(n) = 4T(n/2) + n$, donde n es una potencia de 2 ($n > 3$), $T(1) = 1$, y $T(2) = 6$.

Si $n = 2^k$ podemos escribir la ecuación como:

$$T(2^k) = 4T(2^{k-1}) + 2^k.$$

Haciendo el cambio de variable $t_k = T(2^k)$ obtenemos la ecuación

$$t_k = 4t_{k-1} + 2^k$$

que corresponde a una de las ecuaciones estudiadas anteriormente, cuya solución viene dada por la expresión

$$t_k = c_1(2^k)^2 + c_2 2^k.$$

Deshaciendo el cambio que realizamos al principio obtenemos que

$$T(n) = c_1 n^2 + c_2 n.$$

Calculando entonces las constantes a partir de las condiciones iniciales:

$$T(n) = 2n^2 - n \in \Theta(n^2).$$

1.4.4 Recurrencias No Lineales

En este caso, la ecuación que relaciona $T(n)$ con el resto de los términos no es lineal. Para resolverla intentaremos convertirla en una ecuación lineal como las que hemos estudiado hasta el momento.

Por ejemplo, sea la ecuación $T(n) = nT^2(n/2)$ para n potencia de 2, $n > 1$, con la condición inicial $T(1) = 1/3$. Llamando $t_k = T(2^k)$, la ecuación queda como

$$t_k = T(2^k) = 2^k T^2(2^{k-1}) = 2^k t_{k-1}^2,$$

que no corresponde a ninguno de los tipos estudiados. Necesitamos hacer un cambio más para transformar la ecuación. Tomando logaritmos a ambos lados y haciendo el cambio $u_k = \log t_k$ obtenemos

$$u_k - 2u_{k-1} = k,$$

ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-2)(x-1)^2 = 0$. Por tanto,

$$u_k = c_1 2^k + c_2 + c_3 k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $u_k = \log t_k$

$$t_k = 2^{c_1 2^k + c_2 + c_3 k}$$

y después $t_k = T(2^k)$. En consecuencia

$$T(n) = 2^{c_1 n + c_2 + c_3 \log n}.$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de una y tenemos tres incógnitas, usamos la ecuación en recurrencia original para obtener las restantes:

$$T(2) = 2T^2(1) = 2/9.$$

$$T(4) = 4T^2(2) = 16/81.$$

Con esto llegamos a que $c_1 = \log(4/3) = 2 - \log 3$, $c_2 = -2$, $c_3 = -1$ y por consiguiente:

$$T(n) = \frac{2^{2n}}{4n3^n}.$$

1.5 PROBLEMAS PROPUESTOS

1.1. De las siguientes afirmaciones, indicar cuales son ciertas y cuales no:

- | | |
|---|--|
| (i) $n^2 \in O(n^3)$ | (ix) $n^2 \in \Omega(n^3)$ |
| (ii) $n^3 \in O(n^2)$ | (x) $n^3 \in \Omega(n^2)$ |
| (iii) $2^{n+1} \in O(2^n)$ | (xi) $2^{n+1} \in \Omega(2^n)$ |
| (iv) $(n+1)! \in O(n!)$ | (xii) $(n+1)! \in \Omega(n!)$ |
| (v) $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$ | (xiii) $f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$ |
| (vi) $3^n \in O(2^n)$ | (xiv) $3^n \in \Omega(2^n)$ |
| (vii) $\log n \in O(n^{1/2})$ | (xv) $\log n \in \Omega(n^{1/2})$ |
| (viii) $n^{1/2} \in O(\log n)$ | (xvi) $n^{1/2} \in \Omega(\log n)$ |

1.2. Sea a una constante real, $0 < a < 1$. Usar las relaciones \subset y $=$ para ordenar los órdenes de complejidad de las siguientes funciones: $n \log n$, $n^2 \log n$, n^8 , n^{1+a} , $(1+a)^n$, $(n^2 + 8n + \log^3 n)^4$, $n^2 / \log n$, 2^n .

1.3. La siguiente ecuación recurrente representa un caso típico de un algoritmo recursivo:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \leq n \leq b \\ aT(n-b) + cn^k & \text{si } n > b \end{cases}$$

donde a, c, k son números reales, n, b son números naturales, y $a > 0$, $c > 0$, $k \geq 0$. En general, la constante a representa el número de llamadas recursivas que se realizan para un problema de tamaño n en cada ejecución del algoritmo; $n-b$ es el tamaño de los subproblemas generados; y cn^k representa el coste de las instrucciones del algoritmo que no son llamadas recursivas.

$$\text{Demostrar que } T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < 1 \\ \Theta(n^{k+1}) & \text{si } a = 1 \\ \Theta(a^{n \div b}) & \text{si } a > 1 \end{cases}$$

1.4. La siguiente ecuación recurrente representa un caso típico de *Divide y Vencerás*:

$$T(n) = \begin{cases} cn^k & \text{si } 1 \leq n < b \\ aT(n/b) + cn^k & \text{si } n \geq b \end{cases}$$

donde a, c, k son números reales, n, b son números naturales, y $a > 0$, $c > 0$, $k \geq 0$, $b > 1$. La expresión cn^k representa en general el coste de descomponer el problema inicial en a subproblemas y el de componer las soluciones para producir la solución del problema original.

1.5. Supongamos que disponemos de la siguiente definición de tipo:

Consideramos entonces los procedimientos y funciones siguientes:

```
PROCEDURE Algoritmo1(VAR a:vector);  
    VAR i,j:CARDINAL;  
        temp:INTEGER;  
BEGIN  
    FOR i:=1 TO n-1 DO (* 1 *)  
        FOR j:=n TO i+1 BY -1 DO (* 2 *)  
            IF a[j-1]>a[j] THEN (* 3 *)  
                temp:=a[j-1]; (* 4 *)  
                a[j-1]:=a[j]; (* 5 *)  
                a[j]:=temp (* 6 *)  
            END (* 7 *)  
        END (* 8 *)  
    END (* 9 *)  
END Algoritmo1;
```

```
PROCEDURE Algoritmo2(VAR a:vector;c:INTEGER):CARDINAL;  
    VAR inf,sup,i:CARDINAL;  
BEGIN  
    inf:=1; sup:=n; (* 1 *)  
    WHILE (sup>=inf) DO (* 2 *)  
        i:=(inf+sup) DIV 2; (* 3 *)  
        IF a[i]=c THEN RETURN i (* 4 *)  
        ELSIF c<a[i] THEN sup:=i-1 (* 5 *)  
        ELSE inf:=i+1 (* 6 *)  
        END (* 7 *)  
    END; (* 8 *)  
    RETURN 0; (* 9 *)  
END Algoritmo2;
```

```

PROCEDURE Euclides(m,n:CARDINAL):CARDINAL;
  VAR temp:CARDINAL;
BEGIN
  WHILE m>0 DO
    temp:=m;
    m:=n MOD m;
    n:=temp
  END;
  RETURN n
END Euclides;

```

```

PROCEDURE Misterio(n:CARDINAL);
  VAR i,j,k,s:INTEGER;
BEGIN
  s:=0;
  FOR i:=1 TO n-1 DO
    FOR j:=i+1 TO n DO
      FOR k:=1 TO j DO
        s:=s+2
      END
    END
  END
END Misterio;

```

- a) Calcular sus tiempos de ejecución en el mejor, peor, y caso medio.
 b) Dar cotas asintóticas O , Ω y Θ para las funciones anteriores.

- 1.6. Demostrar las siguientes inclusiones estrictas: $O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(n^k) \subset O(2^n) \subset O(n!)$.
- 1.7. a) Demostrar que $f \in O(g) \Leftrightarrow g \in \Omega(f)$.
 b) Dar un ejemplo de funciones f y g tales que $f \in O(g)$ pero que $f \notin \Omega(g)$.
 c) Demostrar que $\forall a, b > 1$ se tiene que $\log_a n \in \Theta(\log_b n)$.

- 1.8. Considérense las siguientes funciones de n :

$$f_1(n) = n^2; \quad f_2(n) = n^2 + 1000n;$$

$$f_3(n) = \begin{cases} n, & \text{si } n \text{ impar} \\ n^3, & \text{si } n \text{ par} \end{cases}; \quad f_4(n) = \begin{cases} n, & \text{si } n \leq 100 \\ n^3, & \text{si } n > 100 \end{cases}$$

Para cada posible valor de i, j indicar si $f_i \in O(f_j)$ y si $f_i \in \Omega(f_j)$.

1.9. Resolver las siguientes ecuaciones y dar su orden de complejidad:

- a) $T(n)=3T(n-1)+4T(n-2)$ si $n>1$; $T(0)=0$; $T(1)=1$.
- b) $T(n)=2T(n-1)-(n+5)3^n$ si $n>0$; $T(0)=0$.
- c) $T(n)=4T(n/2)+n^2$ si $n>4$, n potencia de 2; $T(1)=1$; $T(2)=8$.
- d) $T(n)=2T(n/2)+n\log n$ si $n>1$, n potencia de 2.
- e) $T(n)=3T(n/2)+5n+3$ si $n>1$, n potencia de 2.
- f) $T(n)=2T(n/2)+\log n$ si $n>1$, n potencia de 2.
- g) $T(n)=2T(n^{1/2})+\log n$ con $n=2^{2^k}$; $T(2)=1$.
- h) $T(n)=5T(n/2)+(n\log n)^2$ si $n>1$, n potencia de 2; $T(1)=1$.
- i) $T(n)=T(n-1)+2T(n-2)-2T(n-3)$ si $n>2$; $T(n)=9n^2-15n+106$ si $n=0,1,2$.
- j) $T(n)=(3/2)T(n/2)-(1/2)T(n/4)-(1/n)$ si $n>2$; $T(1)=1$; $T(2)=3/2$.
- k) $T(n)=2T(n/4)+n^{1/2}$ si $n>4$, n potencia de 4.
- l) $T(n)=4T(n/3)+n^2$ si $n>3$, n potencia de 3.

1.10. Suponiendo que $T_1 \in O(f)$ y que $T_2 \in O(f)$, indicar cuáles de las siguientes afirmaciones son ciertas:

- a) $T_1 + T_2 \in O(f)$.
- b) $T_1 - T_2 \in O(f)$.
- c) $T_1 / T_2 \in O(1)$.
- d) $T_1 \in O(T_2)$.

1.11. Encontrar dos funciones $f(n)$ y $g(n)$ tales que $f \notin O(g)$ y $g \notin O(f)$.

1.12. Demostrar que para cualquier constante k se verifica que $\log^k n \in O(n)$.

1.13. Consideremos los siguientes procedimientos y funciones sobre árboles. Calcular sus tiempos de ejecución y sus órdenes de complejidad.

```

PROCEDURE Inorden(t:arbol);      (* recorrido en inorden de t *)
BEGIN
  IF NOT Esvacio(t) THEN        (* 1
*)
    Inorden(Izq(t));             (* 2 *)
    Opera(Raiz(t));              (* 3 *)
    Inorden(Der(t));             (* 4 *)
  END;                           (* 5 *)
END Inorden;
```

```

PROCEDURE Altura(t:arbol):CARDINAL; (* altura de t *)
BEGIN
  IF Esvacio(t) THEN (* 1 *)
    RETURN 0 (* 2 *)
  ELSE (* 3 *)
    RETURN 1+Max2(Altura(Izq(t)),Altura(Der(t))) (* 4 *)
  END (* 5 *)
END Altura;

```

```

PROCEDURE Mezcla(t1,t2:arbol):arbol;
(* devuelve un arbol binario de busqueda con los elementos de
  los dos arboles binarios de busqueda t1 y t2. La funcion Ins
  inserta un elemento en un arbol binario de busqueda *)
BEGIN
  IF Esvacio(t1) THEN (* 1 *)
    RETURN t2 (* 2 *)
  ELSIF Esvacio(t2) THEN (* 3 *)
    RETURN t1 (* 4 *)
  ELSE (* 5 *)
    RETURN Mezcla(Mezcla(Ins(t1,Raiz(t2)),Izq(t2)),
                  Der(t2)) (* 6 *)
  END (* 7 *)
END Mezcla;

```

Supondremos que las operaciones básicas del tipo abstracto de datos *arbol* (*Raiz*, *Izq*, *Der*, *Esvacio*) son $O(1)$, así como las operaciones *Opera* (que no es relevante lo que hace) y *Max2* (que calcula el máximo de dos números). Por otro lado, supondremos que la complejidad de la función *Ins* es $O(\log n)$.

- 1.14.** Ordenar las siguientes funciones de acuerdo a su velocidad de crecimiento:
 n , \sqrt{n} , $\log n$, $\log \log n$, $\log^2 n$, $n/\log n$, $\sqrt{n} \log^2 n$, $(1/3)^n$, $(3/2)^n$, 17 , n^2 .

- 1.15.** Resolver la ecuación $T(n) = \frac{1}{n} \left(\sum_{i=0}^{n-1} T(i) \right) + cn$, siendo $T(0) = 0$.

- 1.16.** Consideremos las siguientes funciones:

```

CONST n = ...;
TYPE vector = ARRAY[1..n] OF INTEGER;

```

```

PROCEDURE BuscBin(VAR a:vector;
                  prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
  VAR mitad:CARDINAL;
BEGIN
  IF (prim>=ult) THEN RETURN a[ult]=x           (* 1 *)
  ELSE                                           (* 2 *)
    mitad:=(prim+ult)DIV 2;                     (* 3 *)
    IF x=a[mitad] THEN RETURN TRUE              (* 4 *)
    ELSIF (x<a[mitad]) THEN                     (* 5 *)
      RETURN BuscBin(a,prim,mitad-1,x)          (* 6 *)
    ELSE                                        (* 7 *)
      RETURN BuscBin(a,mitad+1,ult,x)           (* 8 *)
    END                                         (* 9 *)
  END                                         (* 10 *)
END BuscBin;

```

```

PROCEDURE Sumadigitos(num:CARDINAL):CARDINAL;
BEGIN
  IF num<10 THEN RETURN num                    (* 1 *)
  ELSE RETURN (num MOD 10)+Sumadigitos(num DIV 10) (* 2 *)
  END                                           (* 3 *)
END Sumadigitos;

```

- a) Calcular sus tiempos de ejecución y sus órdenes de complejidad.
- b) Modificar los algoritmos eliminando la recursión.
- c) Calcular la complejidad de los algoritmos modificados y justificar para qué casos es más conveniente usar uno u otro.

1.17. Consideremos la siguiente función:

```

PROCEDURE Raro(VAR a:vector;prim,ult:CARDINAL):INTEGER;
  VAR mitad,terc:CARDINAL;
BEGIN
  IF (prim>=ult) THEN RETURN a[ult] END;
  mitad:=(prim+ult)DIV 2;      (* posicion central *)
  terc :=(ult-prim)DIV 3;      (* num. elementos DIV 3 *)
  RETURN a[mitad]+Raro(a,prim,prim+terc)+Raro(a,ult-terc,ult)
END Raro;

```

- a) Calcular el tiempo de ejecución de la llamada a la función $Raro(a, 1, n)$, suponiendo que n es potencia de 3.
- b) Dar una cota de complejidad para dicho tiempo de ejecución.

1.6 SOLUCIÓN A LOS PROBLEMAS PROPUESTOS

Antes de comenzar con la resolución de los problemas es necesario hacer una aclaración sobre la notación utilizada para las funciones logarítmicas. A partir de ahora y a menos que se exprese explícitamente otra base, la función “log” hará referencia a logaritmos en base dos.

Solución al Problema 1.1

(☺/☹)

- (i) $n^2 \in O(n^3)$ es cierto pues $\lim_{n \rightarrow \infty} (n^2/n^3) = 0$.
- (ii) $n^3 \in O(n^2)$ es falso pues $\lim_{n \rightarrow \infty} (n^2/n^3) = 0$.
- (iii) $2^{n+1} \in O(2^n)$ es cierto pues $\lim_{n \rightarrow \infty} (2^{n+1}/2^n) = 2$.
- (iv) $(n+1)! \in O(n!)$ es falso pues $\lim_{n \rightarrow \infty} (n!/(n+1)!) = 0$.
- (v) $f(n) \in O(n) \Rightarrow 2^{f(n)} \in O(2^n)$ es falso. Por ejemplo, sea $f(n) = 3n$; claramente $f(n) \in O(n)$ pero sin embargo $\lim_{n \rightarrow \infty} (2^n/2^{3n}) = 0$, con lo cual $2^{3n} \notin O(2^n)$. De forma más general, resulta ser falso para cualquier función lineal de la forma $f(n) = \alpha n$ con $\alpha > 1$, y cierto para $f(n) = \beta n$ con $\beta \leq 1$.
- (vi) $3^n \in O(2^n)$ es falso pues $\lim_{n \rightarrow \infty} (2^n/3^n) = 0$.
- (vii) $\log n \in O(n^{1/2})$ es cierto pues $\lim_{n \rightarrow \infty} (\log n/n^{1/2}) = 0$.
- (viii) $n^{1/2} \in O(\log n)$ es falso pues $\lim_{n \rightarrow \infty} (\log n/n^{1/2}) = 0$.
- (ix) $n^2 \in \Omega(n^3)$ es falso pues $\lim_{n \rightarrow \infty} (n^2/n^3) = 0$.
- (x) $n^3 \in \Omega(n^2)$ es cierto pues $\lim_{n \rightarrow \infty} (n^2/n^3) = 0$.
- (xi) $2^{n+1} \in \Omega(2^n)$ es cierto pues $\lim_{n \rightarrow \infty} (2^{n+1}/2^n) = 2$.
- (xii) $(n+1)! \in \Omega(n!)$ es cierto pues $\lim_{n \rightarrow \infty} (n!/(n+1)!) = 0$.
- (xiii) $f(n) \in \Omega(n) \Rightarrow 2^{f(n)} \in \Omega(2^n)$ es falso. Por ejemplo, sea $f(n) = (1/2)n$; claramente $f(n) \in O(n)$ pero sin embargo $\lim_{n \rightarrow \infty} (2^{(1/2)n}/2^n) = 0$, con lo cual $2^{(1/2)n} \notin \Omega(2^n)$. De forma más general, resulta ser falso para cualquier función $f(n) = \alpha n$ con $\alpha < 1$, y cierto para $f(n) = \beta n$ con $\beta \geq 1$.
- (xiv) $3^n \in \Omega(2^n)$ es cierto pues $\lim_{n \rightarrow \infty} (2^n/3^n) = 0$.

(xv) $\log n \in \Omega(n^{1/2})$ es falso pues $\lim_{n \rightarrow \infty} (\log n / n^{1/2}) = 0$.

(xvi) $n^{1/2} \in \Omega(\log n)$ es cierto pues $\lim_{n \rightarrow \infty} (\log n / n^{1/2}) = 0$.

Solución al Problema 1.2

(☺)

- Respecto al orden de complejidad O tenemos que:

$$O(n \log n) \subset O(n^{1+a}) \subset O(n^2 / \log n) \subset O(n^2 \log n) \subset O(n^8) = O((n^2 + 8n + \log^3 n)^4) \subset O((1+a)^n) \subset O(2^n).$$

Puesto que todas las funciones son continuas, para comprobar que $O(f) \subset O(g)$, basta ver que $\lim_{n \rightarrow \infty} (f(n)/g(n)) = 0$, y para comprobar que $O(f) = O(g)$, basta ver que $\lim_{n \rightarrow \infty} (f(n)/g(n))$ es finito y distinto de 0.

- Por otro lado, respecto al orden de complejidad Ω , obtenemos que:

$$\Omega(n \log n) \supset \Omega(n^{1+a}) \supset \Omega(n^2 / \log n) \supset \Omega(n^2 \log n) \supset \Omega(n^8) = \Omega((n^2 + 8n + \log^3 n)^4) \supset \Omega((1+a)^n) \supset \Omega(2^n)$$

Para comprobar que $\Omega(f) \subset \Omega(g)$, basta ver que $\lim_{n \rightarrow \infty} (g(n)/f(n)) = 0$, y para comprobar que $\Omega(f) = \Omega(g)$, basta ver que $\lim_{n \rightarrow \infty} (f(n)/g(n))$ es finito y distinto de 0 puesto que al ser las funciones continuas tenemos garantizada la existencia de los límites.

- Y en lo relativo al orden de complejidad Θ , al definirse como la intersección de los órdenes O y Ω , sólo tenemos asegurado que:

$$\Theta(n^8) = \Theta((n^2 + 8n + \log^3 n)^4),$$

siendo los órdenes Θ del resto de las funciones conjuntos no comparables.

Solución al Problema 1.3

(☹)

La ecuación dada puede ser también escrita como $T(n) - aT(n-b) = cn^k$, ecuación en recurrencia no homogénea cuya ecuación característica es:

$$(x^b - a)(x - 1)^{k+1} = 0.$$

- Para estudiar las raíces de esa ecuación, vamos a suponer primero que $a \neq 1$. En este caso, la ecuación tiene una raíz de multiplicidad $k+1$ (el 1), y b raíces

distintas r_1, r_2, \dots, r_b (las b raíces b -ésimas de a^\dagger). Entonces la solución de la ecuación en recurrencia es de la forma:

$$T(n) = c_1 1^n + c_2 n 1^n + c_3 n^2 1^n + \dots + c_{k+1} n^k 1^n + d_1 r_1^n + d_2 r_2^n + \dots + d_b r_b^n =$$

$$= \left(\sum_{i=1}^{k+1} c_i n^{i-1} \right) + \left(\sum_{i=1}^b d_i r_i^n \right)$$

siendo c_i y d_i coeficientes reales.

- Si $a < 1$, las raíces b -ésimas de a (esto es, las r_i) son menores en módulo que 1, con lo cual el segundo sumatorio tiende a cero cuando n tiende a ∞ , y en consecuencia $T(n) \in \Theta(n^k)$ pues $\lim_{n \rightarrow \infty} \frac{T(n)}{n^k} = c_{k+1}$ es finito y distinto de cero.

Para ver que efectivamente c_{k+1} es distinto de cero independientemente de las condiciones iniciales, sustituimos esta expresión de $T(n)$ en la ecuación original, $T(n) - aT(n-b) = cn^k$, y por tanto:

$$\left(\sum_{i=1}^{k+1} c_i n^{i-1} + \sum_{i=1}^b d_i r_i^n \right) - a \left(\sum_{i=1}^{k+1} c_i (n-b)^{i-1} + \sum_{i=1}^b d_i r_i^n \right) = cn^k.$$

Igualando ahora los coeficientes que acompañan a n^k obtenemos que $c_{k+1} - ac_{k+1} = c$, o lo que es igual, $(1-a)c_{k+1} = c$. Ahora bien, como sabemos que $a < 1$ y $c > 0$, entonces c_{k+1} no puede ser cero.

- Si $a > 1$, las funciones del segundo sumatorio son exponenciales, mientras que las primeras se mantienen dentro de un orden polinomial, por lo que en este caso el orden de complejidad del algoritmo es exponencial. Ahora bien, como todas las raíces b -ésimas de a tienen el mismo módulo, todas crecen de la misma forma y por tanto todas son del mismo orden de complejidad, obteniendo que

$$\Theta(r_1^n) = \Theta(r_2^n) = \dots = \Theta(r_b^n).$$

Como $\lim_{n \rightarrow \infty} \frac{T(n)}{r_1^n} = d_1$ es distinto de cero y finito, podemos concluir que:

$$T(n) \in \Theta(r_1^n) = \Theta\left(\left(\sqrt[b]{a}\right)^n\right) = \Theta(a^{n \div b}).$$

Hemos supuesto que $d_1 \neq 0$. Esto no tiene por qué ser necesariamente cierto para todas las condiciones iniciales, aunque sin embargo sí es cierto que al menos uno de los coeficientes d_i ha de ser distinto de cero. Basta tomar ese sumando para demostrar lo anterior.

[†] Recordemos que dados dos números reales a y b , la solución de la ecuación $x^b - a = 0$ tiene b raíces distintas, que pueden ser expresadas como $a^{1/b} e^{2\pi i k / b}$, para $k=0, 1, 2, \dots, b-1$.

- Supongamos ahora que $a = 1$. En este caso la multiplicidad de la raíz 1 es $k+2$, con lo cual

$$\begin{aligned} T(n) &= c_1 1^n + c_2 n 1^n + c_3 n^2 1^n + \dots + c_{k+2} n^{k+1} 1^n + d_2 r_2^n + \dots + d_b r_b^n = \\ &= \left(\sum_{i=1}^{k+2} c_i n^{i-1} \right) + \left(\sum_{i=2}^b d_i r_i^n \right) \end{aligned}$$

Pero las raíces r_2, r_3, \dots, r_b son todas de módulo 1 (obsérvese que $r_1=1$), y por tanto el segundo sumando de $T(n)$ es de complejidad $\Theta(1)$.

Así, el crecimiento de $T(n)$ coincide con el del primer sumando, que es un polinomio de grado $k+1$ con lo cual $T(n) \in \Theta(n^{k+1})$.

Solución al Problema 1.4

(\mathcal{G})

Haciendo el cambio $n = b^m$, o lo que es igual, $m = \log_b n$, obtenemos que

$$T(b^m) = aT(b^{m-1}) + cb^{mk}.$$

Llamando $t_m = T(b^m)$, la ecuación queda como

$$t_m - at_{m-1} = c(b^k)^m,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-a)(x-b^k) = 0$.

Para resolver esta ecuación, supongamos primero que $a = b^k$. Entonces, la ecuación característica es $(x-b^k)^2 = 0$ y por tanto

$$t_m = c_1 b^{km} + c_2 m b^{km}.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_m = T(b^m)$ con lo que

$$T(b^m) = c_1 b^{km} + c_2 m b^{km} = (c_1 + c_2 m) b^{km},$$

y después $n = b^m$, obteniendo finalmente que

$$T(n) = (c_1 + c_2 \log_b n) n^k \in \Theta(n^k \log n).^\dagger$$

Supongamos ahora el caso contrario, $a \neq b^k$. Entonces la ecuación característica tiene dos raíces distintas, y por tanto

$$t_m = c_1 a^m + c_2 b^{km}.$$

Necesitamos deshacer los cambios hechos. Primero $t_m = T(b^m)$, con lo que

$$T(b^m) = c_1 a^m + c_2 b^{km},$$

y después $n = b^m$, obteniendo finalmente que

$$T(n) = c_1 a^{\log_b n} + c_2 n^k = c_1 n^{\log_b a} + c_2 n^k.$$

[†] Obsérvese que se hace uso de que $\log_b n \in \Theta(\log n)$, lo que se demuestra en el problema 1.7.

En consecuencia, si $\log_b a > k$ (si y sólo si $a > b^k$) entonces $T(n) \in \Theta(n^{\log_b a})$. Si no, es decir, $a < b^k$, entonces $T(n) \in \Theta(n^k)$.

Solución al Problema 1.5

Procedimiento *Algoritmo1* (☺)

a) Para obtener el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan 3 OE (una asignación, una resta y una comparación) en cada una de las iteraciones del bucle más otras 3 al final, cuando se efectúa la salida del *FOR*.
- Igual ocurre con la línea (2), también con 3 OE (una asignación, una suma y una comparación) por iteración, más otras 3 al final del bucle.
- En la línea (3) se efectúa una condición, con un total de 4 OE (una diferencia, dos accesos a un vector, y una comparación).
- Las líneas (4) a (6) sólo se ejecutan si se cumple la condición de la línea (3), y realizan un total de 9 OE: 3, 4 y 2 respectivamente.

Con esto:

En el *caso mejor* para el algoritmo la condición de la línea (3) será siempre falsa, y no se ejecutarán nunca las líneas (4), (5) y (6). Así, el bucle más interno realizará $(n-i)$ iteraciones, cada una de ellas con 4 OE (línea 3), más las 3 OE de la línea (2). Por tanto, el bucle más interno realiza un total de

$$\left(\sum_{j=i+1}^n (4 + 3) \right) + 3 = 7 \left(\sum_{j=i+1}^n 1 \right) + 3 = 7(n-i) + 3$$

OE, siendo el 3 adicional por la condición de salida del bucle.

A su vez, el bucle externo repetirá esas $7(n-i)+3$ OE en cada iteración, lo que hace que el número de OE que se realizan en el algoritmo sea:

$$T(n) = \left(\sum_{i=1}^{n-1} (7(n-i) + 3) + 3 \right) + 3 = \frac{7}{2}n^2 + \frac{5}{2}n - 3.$$

- En el *caso peor*, la condición de la línea (3) será siempre verdadera, y las líneas (4), (5) y (6) se ejecutarán en todas las iteraciones. Por tanto, el bucle más interno realiza

$$\left(\sum_{j=i+1}^n (4 + 9 + 3) \right) + 3 = 16(n-i) + 3$$

OE. El bucle externo realiza aquí el mismo número de iteraciones que en el caso anterior, por lo que el número de OE en este caso es:

$$T(n) = \left(\sum_{i=1}^{n-1} (16(n-i) + 3) + 3 \right) + 3 = 8n^2 - 2n - 3.$$

- En el *caso medio*, la condición de la línea (3) será verdadera con probabilidad 1/2. Así, las líneas (4), (5) y (6) se ejecutarán en la mitad de las iteraciones del bucle más interno, y por tanto realiza

$$\left(\sum_{j=i+1}^n \left(4 + \frac{1}{2} 9 \right) + 3 \right) + 3 = \frac{23}{2} (n-i) + 3$$

OE. El bucle externo realiza aquí el mismo número de iteraciones que en el caso anterior, por lo que el número de OE en este caso es:

$$T(n) = \left(\sum_{i=1}^{n-1} \left(\frac{23}{2} (n-i) + 3 \right) + 3 \right) + 3 = \frac{23}{4} n^2 + \frac{1}{4} n - 3.$$

b) Como los tiempos de ejecución en los tres casos son polinomios de grado 2, la complejidad del algoritmo es cuadrática, independientemente de qué caso se trate.

Obsérvese cómo hemos analizado el tiempo de ejecución del algoritmo sólo en función de su código y no respecto a lo que hace, puesto que en muchos casos esto nos llevaría a conclusiones erróneas. Debe ser a posteriori cuando se analice el objetivo para el que fue diseñado el algoritmo.

En el caso que nos ocupa, un examen más detallado del código del procedimiento nos muestra que el algoritmo está diseñado para ordenar de forma creciente el vector que se le pasa como parámetro, siguiendo el método de la Burbuja. Lo que acabamos de ver es que sus casos mejor, peor y medio se producen respectivamente cuando el vector está inicialmente ordenado de forma creciente, decreciente y aleatoria.

Función *Algoritmo2*

(S)

a) Para calcular el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan 2 OE (dos asignaciones).
- En la línea (2) se efectúa la condición del bucle, que supone 1 OE (la comparación).
- Las líneas (3) a (6) componen el cuerpo del bucle, y contabilizan 3, 2+1, 2+2 y 2 OE respectivamente. Es importante hacer notar que el bucle también puede finalizar si se verifica la condición de la línea (4).
- Por último, la línea (9) supone 1 OE. A ella se llega cuando la condición del bucle *WHILE* deja de ser cierta.

Con esto:

- En el *caso mejor* se efectuarán solamente la líneas (1), (2), (3) y (4). En consecuencia, $T(n) = 2+1+3+3 = 9$.
- En el *caso peor* se efectúa la línea (1), y después se repite el bucle hasta que su condición sea falsa, acabando la función al ejecutarse la línea (9). Cada iteración del bucle está compuesta por las líneas (2) a (8), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. En cada iteración se reducen a la mitad los elementos a considerar, por lo que el bucle se repite $\log n$ veces. Por tanto,

$$T(n) = 2 + \left(\left(\sum_{i=1}^{\log n} (1 + 3 + 2 + 2 + 2) \right) + 1 \right) + 1 = 10 \log n + 4.$$

- En el *caso medio*, necesitamos calcular el número medio de veces que se repite el bucle, y para esto veamos cuántas veces puede repetirse, y qué probabilidad tiene cada una de suceder.

Por un lado, el bucle puede repetirse desde una vez hasta $\log n$ veces, puesto que en cada iteración se divide por dos el número de elementos considerados. Si se repitiese una sola vez, es que el elemento ocuparía la posición $n/2$, lo que ocurre con una probabilidad $1/(n+1)$. Si el bucle se repitiese dos veces es que el elemento ocuparía alguna de las posiciones $n/4$ ó $3n/4$, lo cual ocurre con probabilidad $1/(n+1) + 1/(n+1) = 2/(n+1)$. En general, si se repitiese i veces es que el elemento ocuparía alguna de las posiciones $nk/2^i$, con k impar y $1 \leq k < 2^i$.

Es decir, el bucle se repite i veces con probabilidad $2^{i-1}/(n+1)$. Por tanto, el número medio de veces que se repite el ciclo vendrá dado por la expresión:

$$\sum_{i=1}^{\log n} i \frac{2^{i-1}}{n+1} = \frac{n \log n - n + 1}{n+1}.$$

Con esto, la función ejecuta la línea (1) y después el bucle se repite ese número medio de veces, saliendo por la instrucción *RETURN* en la línea (4). Por consiguiente,

$$T(n) = 2 + \left(\frac{n \log n - n + 1}{n+1} \right) (1 + 3 + 2 + 2) + (1 + 3 + 3) = 9 + 8 \frac{n \log n - n + 1}{n+1}$$

- c) En el caso mejor el tiempo de ejecución es una constante. Para los casos peor y medio, la complejidad resultante es de orden $\Theta(\log n)$ puesto que

$$\lim_{n \rightarrow \infty} \frac{T(n)}{\log n}$$

es una constante finita y distinta de cero en ambos casos (10 y 8 respectivamente).

Función Euclides

(G)

a) En este caso el análisis del tiempo de ejecución y la complejidad de la función sigue un proceso distinto al estudiado en los casos anteriores.

Lo primero es resaltar algunas características del algoritmo, siguiendo una línea de razonamiento similar a la de [BRA97]:

- [1] Para cualquier par de enteros no negativos m y n tales que $n \geq m$, se verifica que $n \bmod m < n/2$. Veámoslo:
 - a) Si $m > n/2$ entonces $1 \leq n/m < 2$ y por tanto $n \text{ DIV } m = 1$, lo que implica que $n \bmod m = n - m(n \text{ DIV } m) = n - m < n - n/2 = n/2$.
 - b) Por otro lado, si $m \leq n/2$ entonces $n \bmod m < m \leq n/2$.
- [2] Podemos suponer sin pérdida de generalidad que $n \geq m$. Si no, la primera iteración del bucle intercambia n con m ya que $n \bmod m = n$ cuando $n < m$. Además, la condición $n \geq m$ se conserva siempre (es decir, es un invariante del bucle) pues $n \bmod m$ nunca es mayor que m .
- [3] El cuerpo del bucle efectúa 4 OE, con lo cual el tiempo del algoritmo es del orden exacto del número de iteraciones que realiza el bucle. Por consiguiente, para determinar la complejidad del algoritmo es suficiente acotar este número.
- [4] Una propiedad curiosa de este algoritmo es que no se produce un avance notable con cada iteración del bucle, sino que esto ocurre cada dos iteraciones. Consideremos lo que les ocurre a m y n cuando el ciclo se repite dos veces, suponiendo que no acaba antes. Sean m_0 y n_0 los valores originales de los parámetros, que podemos suponer $n_0 \geq m_0$ por [2]. Después de la primera iteración, m vale $n_0 \bmod m_0$. Después de la segunda iteración, n toma ese valor, y por tanto ya es menor que $n_0/2$ (por [1]). En consecuencia, n vale menos de la mitad de lo que valía tras dos iteraciones del bucle. Como se sigue manteniendo que $n \geq m$, el mismo razonamiento se puede repetir para las siguientes dos iteraciones, y así sucesivamente.

El hecho de que n valga menos de la mitad cada dos iteraciones del bucle es el que nos permite intuir que el bucle se va a repetir del orden de $2 \log n$ veces. Vamos a demostrar esto formalmente.

Para ello, vamos a tratar el bucle como si fuera un algoritmo recursivo. Sea $T(l)$ el número máximo de veces que se repite el bucle para valores iniciales m y n cuando $m \leq n \leq l$. En este caso l representa el tamaño de la entrada.

- Si $n \leq 2$ el bucle no se repite (si $m = 0$) o se hace una sola vez (si m es 1 ó 2).
- Si $n > 2$ y $m=1$ o bien m divide a n , el bucle se repite una sola vez.
- En otro caso ($n > 2$ y m no divide a n) el bucle se ejecuta dos veces, y por lo visto en [4], n vale a lo sumo la mitad de lo que valía inicialmente. En consecuencia $n \leq (l/2)$, y además m se sigue manteniendo por debajo de n .

Esto nos lleva a la ecuación en recurrencia $T(l) \leq 2 + T(l/2)$ si $l > 2$, $T(l) \leq 1$ si $l \leq 2$, lo que implica que el algoritmo de Euclides es de complejidad logarítmica respecto al tamaño de la entrada (l).

Nos preguntaremos la razón de usar $T(l)$ para acotar el número de iteraciones que realiza el algoritmo en vez de definir T directamente como una función de n , el mayor de los dos operandos, lo cual sería mucho más intuitivo.

El problema es que si definimos $T(n)$ como el número de iteraciones que realiza el algoritmo para los valores $m \leq n$, no podríamos concluir que $T(n) \leq 2 + T(n/2)$ del hecho de que n valga la mitad de su valor tras cada dos iteraciones del bucle.

Por ejemplo, para *Euclides*(8,13), obtenemos que $T(13) = 5$ en el peor caso, mientras que $T(13/2) = T(6) = 2$. Esto ocurre porque tras dos iteraciones del bucle n no vale 6, sino 5 (y $m = 3$), y con esto sí es cierto que $T(13) \leq 2 + T(5)$ ya que $T(5) = 3$.

La raíz de este problema es que esta nueva definición más intuitiva de T no lleva a una función monótona no decreciente ($T(5) > T(6)$) y por tanto la existencia de algún $n' \leq n/2$ tal que $T(n) \leq 2 + T(n')$ no implica necesariamente que $T(n) \leq 2 + T(n/2)$.

En vez de esto, solamente podríamos afirmar que $T(n) \leq 2 + \max\{T(n') | n' \leq n/2\}$, que es una ecuación en recurrencia bastante difícil de resolver. Esa es la razón de que escogiésemos nuestra función T de forma que fuera no decreciente y que expresara una cota superior del número de iteraciones.

Para acabar, es interesante hacer notar una característica curiosa de este algoritmo: se demuestra que su caso peor ocurre cuando m y n son dos términos consecutivos de la sucesión de Fibonacci.

b) $T(l) \in \Theta(\log l)$ como se deduce de la ecuación en recurrencia que define el tiempo de ejecución del algoritmo.

Procedimiento *Misterio*

(☺)

a) En este caso son tres bucles anidados los que se ejecutan, independientemente de los valores de la entrada, es decir, no existe peor, medio o mejor caso, sino un único caso.

Para calcular el tiempo de ejecución, veamos el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecuta 1 OE (una asignación).
- En la línea (2) se ejecutarán 3 OE (una asignación, una resta y una comparación) en cada una de las iteraciones del bucle más otras 3 al final, cuando se efectúa la salida del *FOR*.
- Igual ocurre con la línea (3), también con 3 OE (una asignación, una suma y una comparación) por iteración, más otras 3 al final del bucle.
- Y también en la línea (4), esta vez con 2 OE (asignación y comparación) más las 2 adicionales de terminación del bucle.
- Por último, la línea (5) supone 2 OE (un incremento y una asignación).

Con esto, el bucle interno se ejecutará j veces, el medio $(n-i)$ veces, y el bucle exterior $(n-1)$ veces, lo que conlleva un tiempo de ejecución de:

$$\begin{aligned}
T(n) &= 1 + \left(\sum_{i=1}^{n-1} \left(3 + \left(\sum_{j=i+1}^n \left(3 + \left(\sum_{k=1}^j (2+2) \right) + 2 \right) + 3 \right) \right) + 3 = \\
&= 1 + \left(\sum_{i=1}^{n-1} \left(3 + \left(\sum_{j=i+1}^n (3 + (4j) + 2) \right) + 3 \right) \right) + 3 = 1 + \left(\sum_{i=1}^{n-1} \left(3 + \left(\sum_{j=i+1}^n (4j + 5) \right) + 3 \right) \right) + 3 = \\
&= 1 + \left(\sum_{i=1}^{n-1} (3 + (2(n+i) + 7)(n-i) + 3) \right) + 3 = \\
&= 1 + \frac{8n^3 + 15n^2 + 13n - 36}{6} + 3 = \frac{4}{3}n^3 + \frac{15}{6}n^2 + \frac{13}{6}n - 2.
\end{aligned}$$

b) Como el tiempo de ejecución es un polinomio de grado 3, la complejidad del algoritmo es de orden $\Theta(n^3)$.

Solución al Problema 1.6

(☺)

Para comprobar que $O(f) \subset O(g)$ en cada caso y que esa inclusión es estricta, basta ver que $\lim_{n \rightarrow \infty} (f(n)/g(n)) = 0$, pues todas las funciones son continuas y por tanto los límites existen. Por consiguiente,

$$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(n^k) \subset O(2^n) \subset O(n!).$$

Solución al Problema 1.7

(☺)

a) Por la definición de O , sabemos que $f \in O(g)$ si y sólo si existen $c_1 > 0$ y n_1 tales que $f(n) \leq c_1 g(n)$ para todo $n \geq n_1$.

Análogamente, por la definición de Ω tenemos que $g \in \Omega(f)$ si y sólo si existen $c_2 > 0$ y n_2 tales que $g(n) \geq c_2 f(n)$ para todo $n \geq n_2$. Por consiguiente,

\Rightarrow) Si $f \in O(g)$ basta tomar $c_2 = 1/c_1$ y $n_2 = n_1$ para ver que $g(n) \in \Omega(f)$.

\Leftarrow) Recíprocamente, si $g \in \Omega(f)$ basta tomar $c_1 = 1/c_2$ y $n_1 = n_2$ para que $f \in O(g)$.

Obsérvese que esto es posible pues c_1 y c_2 son ambos estrictamente mayores que cero, y por tanto poseen inverso.

$$b) \text{ Sean } f(n) = \begin{cases} n^2 & \text{si } n \text{ es par.} \\ 1 & \text{si } n \text{ es impar.} \end{cases} \text{ y } g(n) = n^2.$$

Entonces $\Theta(g) = \Theta(n^2)$, y por otro lado $O(f) = O(n^2)$, con lo cual $f \in O(n^2) = O(g)$. Sin embargo, si n es impar no puede existir $c > 0$ tal que $f(n) = 1 \geq cn^2 = cg(n)$, y por consiguiente $f \notin \Omega(g)$.

Intuitivamente, lo que buscamos es una función f cuyo crecimiento asintótico estuviera acotado superiormente por g (es decir, que f no creciera “más deprisa” que g) y que sin embargo f no estuviera acotado inferiormente por g .

c) Veamos que $\log_a n \in \Theta(\log_b n)$.

Sabemos por las propiedades de los logaritmos que si a y b son números reales mayores que 1 se cumple que

$$\log_b n = \frac{\log_a n}{\log_a b}.$$

Con esto,

$$\lim_{n \rightarrow \infty} \frac{\log_a n}{\log_b n} = \lim_{n \rightarrow \infty} \log_a b = \log_a b,$$

que es una constante real finita distinta de cero (pues $a, b > 1$), y por tanto $\Theta(\log_a n) = \Theta(\log_b n)$.

Solución al Problema 1.8

(☺)

Para justificar estas afirmaciones nos apoyaremos en la definición de O y Ω , y trataremos de encontrar la constante real c y el número natural n_0 que caracterizan las inecuaciones que definen a ambas cotas.

- $f_1 \in O(f_2)$ pues $n^2 \leq n^2 + 1000n$ para todo n (podemos tomar $c = 1$, $n_0 = 1$).
- $f_1 \notin O(f_3)$ pues si n es impar no existen c y n_0 tales que $n^2 \leq cn$.
- $f_1 \in O(f_4)$ pues $n^2 \leq n^3$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
- $f_2 \in O(f_1)$ pues basta tomar c y n_0 tales que $c > 1 + 1000/n$ para todo $n \geq n_0$ que sabemos que existen pues $(1000/n)$ tiende a cero.
- $f_2 \notin O(f_3)$ pues si n es impar no existen c y n_0 tales que $n^2 + 1000n \leq cn$.
- $f_2 \in O(f_4)$ pues $n^2 + 1000n \leq n^3$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
- $f_3 \notin O(f_1)$ pues si n es par no existen c y n_0 tales que $n^3 \leq cn^2$.
- $f_3 \notin O(f_2)$ pues si n es par no existen c y n_0 tales que $n^3 \leq c(n^2 + 1000n)$.
- $f_3 \in O(f_4)$ pues $f_3(n) \leq n^3 = f_4(n)$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
- $f_4 \notin O(f_1)$ pues si $n > 100$ no existen c y n_0 tales que $n^3 \leq cn^2$.
- $f_4 \notin O(f_2)$ pues si $n > 100$ no existen c y n_0 tales que $n^3 \leq c(n^2 + 1000n)$.
- $f_4 \notin O(f_3)$ pues si $n > 100$, n impar, no existen c y n_0 tales que $n^3 \leq cn$.
- $f_1 \in \Omega(f_2)$ pues $n^2 \geq c(n^2 + 1000n)$ para c y n_0 tales que $c > 1 + 1000/n$ para todo $n \geq n_0$ que sabemos que existen pues $(1000/n)$ tiende a cero.
- $f_1 \notin \Omega(f_3)$ pues si n es par no existen c y n_0 tales que $n^2 \geq cn^3$.

- $f_1 \notin \Omega(f_4)$ pues si $n > 100$ no existen c y n_0 tales que $n^2 \geq cn^3$.
- $f_2 \in \Omega(f_1)$ pues $n^2 + 100n \geq n^2$ para todo n (podemos tomar $c = 1$, $n_0 = 1$).
- $f_2 \notin \Omega(f_3)$ pues si n es par no existen c y n_0 tales que $n^2 + 1000n \geq cn^3$.
- $f_2 \notin \Omega(f_4)$ pues si $n > 100$ no existen c y n_0 tales que $n^2 + 1000n \geq cn^3$.
- $f_3 \notin \Omega(f_1)$ pues si n es impar no existen c y n_0 tales que $n \geq cn^2$.
- $f_3 \notin \Omega(f_2)$ pues si n es impar no existen c y n_0 tales que $n \geq c(n^2 + 1000n)$.
- $f_3 \notin \Omega(f_4)$ pues si n es impar no existen c y n_0 tales que $n \geq cn^3$.
- $f_4 \in \Omega(f_1)$ pues $n^3 \geq n^2$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
- $f_4 \in \Omega(f_2)$ pues $n^3 \geq n^2 + 1000n$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).
- $f_4 \in \Omega(f_3)$ pues $n^3 \geq f_3(n)$ si $n > 100$ (podemos tomar $c = 1$, $n_0 = 101$).

Obsérvese que $\Theta(f_1) = \Theta(f_2) = \Theta(n^2)$, $\Theta(f_4) = \Theta(n^3)$, $\Omega(f_3) = \Omega(n)$, y $O(f_3) = O(n^3)$.

Solución al Problema 1.9

(☺/☺)

Para resolver estas ecuaciones seguiremos generalmente el mismo método, basado en los resultados expuestos al comienzo del capítulo. Primero intentaremos transformar la ecuación en recurrencia en una ecuación de la forma:

$$a_0 T(n) + a_1 T(n-1) + \dots + a_k T(n-k) = b_1^n p_1(n) + b_2^n p_2(n) + \dots + b_s^n p_s(n),$$

para después resolver su ecuación característica asociada. Con las raíces de esta ecuación es fácil ya obtener el término general de la función buscada.

a) $T(n) = 3T(n-1) + 4T(n-2)$ si $n > 1$; $T(0) = 0$; $T(1) = 1$.

Escribiendo la ecuación de otra forma:

$$T(n) - 3T(n-1) - 4T(n-2) = 0,$$

ecuación en recurrencia homogénea con ecuación característica $x^2 - 3x - 4 = 0$. Resolviendo esta ecuación, sus raíces son 4 y -1, con lo cual:

$$T(n) = c_1 4^n + c_2 (-1)^n.$$

Para calcular las constantes necesitamos las condiciones iniciales:

$$\left. \begin{array}{l} 0 = T(0) = c_1 4^0 + c_2 (-1)^0 = c_1 + c_2 \\ 1 = T(1) = c_1 4^1 + c_2 (-1)^1 = 4c_1 - c_2 \end{array} \right\} \Rightarrow \left. \begin{array}{l} c_1 = 1/5 \\ c_2 = -1/5 \end{array} \right\}$$

Sustituyendo entonces en la ecuación anterior, obtenemos

$$T(n) = \frac{1}{5}(4^n - (-1)^n) \in O(4^n).$$

b) $T(n) = 2T(n-1) - (n+5)3^n$ si $n > 0$; $T(0) = 0$.

Reescribiendo la ecuación obtenemos:

$$T(n) - 2T(n-1) = -(n+5)3^n,$$

que es una ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-2)(x-3)^2 = 0$, de raíces 2 y 3 (esta última con grado de multiplicidad dos), con lo cual

$$T(n) = c_1 2^n + c_2 3^n + c_3 n 3^n.$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de una y tenemos tres incógnitas, usamos la ecuación en recurrencia para obtener las otras dos:

$$T(1) = 2T(0) - 6 \cdot 3 = -18$$

$$T(2) = 2T(1) - 7 \cdot 9 = -99$$

Con esto:

$$\left. \begin{aligned} 0 &= T(0) = c_1 2^0 + c_2 3^0 + c_3 0 \cdot 3^0 = c_1 + c_2 \\ -18 &= T(1) = c_1 2^1 + c_2 3^1 + c_3 1 \cdot 3^1 = 2c_1 + 3c_2 + 3c_3 \\ -99 &= T(2) = c_1 2^2 + c_2 3^2 + c_3 2 \cdot 3^2 = 4c_1 + 9c_2 + 18c_3 \end{aligned} \right\} \Rightarrow \begin{aligned} c_1 &= 9 \\ c_2 &= -9 \\ c_3 &= -3 \end{aligned}$$

Sustituyendo estos valores en la ecuación anterior, obtenemos

$$T(n) = 9 \cdot 2^n - 9 \cdot 3^n - 3n 3^n \in \Theta(n 3^n).$$

Existe otra forma de resolver este tipo de problemas, que se basa en manipular la ecuación original hasta convertirla en homogénea. Partiendo de la ecuación $T(n) - 2T(n-1) = (n+5)3^n$, necesitamos escribir un sistema de ecuaciones basado en ella que permita anular el término no dependiente de $T(n)$. Para ello:

- primero escribimos la recurrencia original,
- la segunda ecuación se obtiene reemplazando n por $n-1$ y multiplicando por -6 ,
- y la tercera se obtiene reemplazando n por $n-2$ y multiplicando por 9 .

De esta forma obtenemos:

$$\begin{aligned} T(n) - 2T(n-1) &= (n+5)3^n \\ -6T(n-1) + 12T(n-2) &= -6(n+4)3^{n-1} \\ 9T(n-2) - 18T(n-3) &= 9(n+3)3^{n-2} \end{aligned}$$

Sumando estas tres ecuaciones conseguimos una ecuación homogénea:

$$T(n) - 8T(n-1) + 21T(n-2) - 18T(n-3) = 0$$

cuya ecuación característica es $x^3 - 8x^2 + 21x - 18 = (x-2)(x-3)^2$. A partir de aquí se puede resolver mediante el proceso descrito anteriormente.

Como puede observarse, este método es más intuitivo pero menos metódico y ordenado que el que hemos utilizado para solucionar ecuaciones en recurrencia. Además, no hay una única forma de plantear estas ecuaciones.

c) $T(n) = 4T(n/2) + n^2$ si $n > 4$, n potencia de 2; $T(1) = 1$; $T(2) = 8$.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 4T(2^{k-1}) + 2^{2k}.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 4t_{k-1} + 4^k,$$

ecuación no homogénea con ecuación característica $(x-4)^2 = 0$. Por tanto,

$$t_k = c_1 4^k + c_2 k 4^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 4^k + c_2 k 4^k = c_1 2^{2k} + c_2 k 2^{2k}$$

y después $n = 2^k$, obteniendo finalmente

$$T(n) = c_1 n^2 + c_2 n^2 \log n.$$

Para calcular las constantes necesitamos las condiciones iniciales:

$$\left. \begin{array}{l} 1 = T(1) = c_1 1^2 + c_2 1^2 \cdot 0 = c_1 \\ 8 = T(2) = c_1 2^2 + c_2 2^2 \cdot 1 = 4c_1 + 4c_2 \end{array} \right\} \Rightarrow \begin{array}{l} c_1 = 1 \\ c_2 = 1 \end{array}$$

Sustituyendo estos valores en la ecuación anterior, obtenemos

$$T(n) = n^2 + n^2 \log n \in \Theta(n^2 \log n).$$

Existe otra forma de resolver este tipo de problemas, mediante el desarrollo “telescópico” de la ecuación en recurrencia. Escribiremos la ecuación hasta llegar a una expresión en donde sólo aparezcan las condiciones iniciales:

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{2}\right) + n^2 = 4\left(4T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2\right) + n^2 = \\ &= 4^2 T\left(\frac{n}{4}\right) + 2n^2 = 4^2\left(4T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2\right) + 2n^2 = \\ &= 4^3 T\left(\frac{n}{8}\right) + 3n^2 = \dots = 4^x T(1) + xn^2. \end{aligned}$$

De esta forma hemos ido desarrollando los términos de esta sucesión, cada uno en función de términos anteriores. Sólo nos queda por calcular el número de términos (x) que hemos tenido que desarrollar.

Pero ese número x coincide con el número de términos de la sucesión $n/2, n/4, n/8, \dots, 4, 2, 1$, que es $\log n$ pues n es una potencia de 2. En consecuencia,

$$T(n) = 4^{\log n} T(1) + \log n \cdot n^2 = n^{\log 4} \cdot 1 + \log n \cdot n^2 = n^2 + \log n \cdot n^2 \in \Theta(n^2 \log n).$$

d) $T(n) = 2T(n/2) + n \log n$ si $n > 1$, n potencia de 2.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 2T(2^{k-1}) + k2^k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 2t_{k-1} + k2^k,$$

ecuación en recurrencia no homogénea con ecuación característica $(x-2)^3 = 0$. Por tanto,

$$t_k = c_1 2^k + c_2 k 2^k + c_3 k^2 2^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 2^k + c_2 k 2^k + c_3 k^2 2^k,$$

y después $n = 2^k$ ($k = \log n$), por lo cual

$$T(n) = c_1 n + c_2 n \log n + c_3 n \log^2 n.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación original:

$$n \log n = T(n) - 2T(n/2) = (c_3 - c_2)n + 2c_3 n \log n,$$

por lo que $c_3 = c_2$ y $2c_3 = 1$, de donde

$$T(n) = c_1 n + 1/2 n \log n + 1/2 n \log^2 n.$$

En consecuencia $T(n) \in \Theta(n \log^2 n)$ independientemente de las condiciones iniciales.

e) $T(n) = 3T(n/2) + 5n + 3$ si $n > 1$, n potencia de 2.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 3T(2^{k-1}) + 5 \cdot 2^k + 3.$$

Llamando $t_k = T(2^k)$, la ecuación final es:

$$t_k = 3t_{k-1} + 5 \cdot 2^k + 3,$$

ecuación en recurrencia no homogénea cuya ecuación característica asociada es $(x-3)(x-2)(x-1) = 0$. Por tanto,

$$t_k = c_1 3^k + c_2 2^k + c_3.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 3^k + c_2 2^k + c_3$$

y después $n = 2^k$ ($k = \log n$), por lo cual

$$T(n) = c_1 3^{\log n} + c_2 n + c_3 = c_1 n^{\log 3} + c_2 n + c_3.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación en recurrencia original, y obtenemos:

$$c_1 n^{\log 3} + c_2 n + c_3 = 3(c_1 (n^{\log 3}/3) + c_2 n/2 + c_3) + 5n + 3.$$

Igualando los coeficientes de $n^{\log 3}$, n y los términos independientes obtenemos $c_3 = -3/2$ y $c_2 = -10$, de donde

$$T(n) = c_1 n^{\log 3} - 10n - 3/2.$$

Como $\log 3 > 1$, $T(n)$ será de complejidad $\Theta(n^{\log 3})$ si c_1 es distinto de cero, o bien $T(n) \in \Theta(n)$ si $c_1 = 0$.

Para ver cuándo c_1 vale cero estudiaremos los valores de las condiciones iniciales que le hacen tomar ese valor, en este caso $T(1)$. Por un lado, utilizando la ecuación original, tenemos que para $n = 2$:

$$T(2) = 3T(1) + 10 + 3.$$

Por otro lado, basándonos en la ecuación que hemos obtenido,

$$T(2) = 3c_1 - 20 - 3/2.$$

Igualando ambas ecuaciones, obtenemos que $c_1 = T(1) + 23/2$. Por tanto,

$$T(n) \in \begin{cases} \Theta(n^{\log 3}) & \text{si } T(1) \neq -23/2 \\ \Theta(n) & \text{si } T(1) = -23/2 \end{cases}$$

f) $T(n) = 2T(n/2) + \log n$ si $n > 1$, n potencia de 2.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 2T(2^{k-1}) + k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 2t_{k-1} + k,$$

ecuación en recurrencia no homogénea que puede ser expresada como

$$t_k - 2t_{k-1} = k$$

y cuya ecuación característica asociada es $(x-2)(x-1)^2 = 0$. Por tanto,

$$t_k = c_1 2^k + c_2 + c_3 k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 2^k + c_2 + c_3 k$$

y después $n = 2^k$ ($k = \log n$), y por tanto

$$T(n) = c_1 n + c_2 + c_3 \log n.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación en recurrencia original, y obtenemos:

$$c_1 n + c_2 + c_3 \log n = 2(c_1 n/2 + c_2 + c_3 \log n - c_3) + \log n.$$

Igualando los coeficientes de $\log n$ y los términos independientes obtenemos que $c_3 = -1$ y $c_2 = -2$, de donde

$$T(n) = c_1 n - 2 - \log n.$$

Esta función será de orden de complejidad $\Theta(n)$ si c_1 es distinto de cero, o bien $T(n) \in \Theta(\log n)$ si $c_1 = 0$.

Para ver cuándo c_1 vale cero estudiaremos los valores de las condiciones iniciales que le hacen tomar ese valor, en este caso $T(1)$. Por un lado, utilizando la ecuación original, tenemos que para $n = 2$:

$$T(2) = 2T(1) + 1.$$

Por otro lado, basándonos en la ecuación que hemos obtenido

$$T(2) = 2c_1 - 2 - 1.$$

Igualando ambas ecuaciones, obtenemos que $c_1 = T(1) + 2$. Por tanto,

$$T(n) \in \begin{cases} \Theta(\log n) & \text{si } T(1) = -2 \\ \Theta(n) & \text{si } T(1) \neq -2 \end{cases}$$

g) $T(n) = 2T(n^{1/2}) + \log n$ con $n = 2^{2^k}$; $T(2) = 1$.

Haciendo el cambio $n = 2^{2^k}$ ($k = \log \log n$) obtenemos la ecuación

$$T(2^{2^k}) = 2T(2^{2^{k-1}}) + \log 2^{2^k}.$$

Llamando $t_k = T(2^{2^k})$, la ecuación final es

$$t_k = 2t_{k-1} + 2^k,$$

ecuación en recurrencia no homogénea cuya ecuación característica es $(x-2)^2 = 0$. Por tanto,

$$t_k = c_1 2^k + c_2 k 2^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^{2^k})$, con lo que

$$T(2^{2^k}) = c_1 2^k + c_2 k 2^k$$

y después $n = 2^{2^k}$ ($k = \log \log n$, o bien $\log n = 2^k$), por lo cual tenemos que

$$T(n) = c_1 \log n + c_2 \log n \cdot \log \log n.$$

Para calcular las constantes necesitamos las condiciones iniciales. Como disponemos de sólo una y tenemos dos incógnitas, usamos la ecuación original para obtener la otra:

$$T(4) = 2T(2) + \log 4 = 4.$$

Con esto:

$$\left. \begin{array}{l} 1 = T(2) = c_1 \log 2 + c_2 \log 2 \cdot 0 = c_1 \\ 4 = T(4) = c_1 \log 4 + c_2 \log 4 \cdot \log \log 4 = 2c_1 + 2c_2 \end{array} \right\} \Rightarrow \left. \begin{array}{l} c_1 = 1 \\ c_2 = 1 \end{array} \right\}$$

Sustituyendo estos valores en la ecuación anterior, obtenemos

$$T(n) = \log n + \log n \cdot \log \log n \in \Theta(\log n \cdot \log \log n).$$

h) $T(n) = 5T(n/2) + (n \log n)^2$ si $n > 1$, n potencia de 2; $T(1) = 1$.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 5T(2^{k-1}) + (k 2^k)^2 = 5T(2^{k-1}) + k^2 4^k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 5t_{k-1} + k^2 4^k,$$

ecuación en recurrencia no homogénea que puede ser expresada como

$$t_k - 5 t_{k-1} = k^2 4^k,$$

cuya ecuación característica asociada es $(x-5)(x-4)^3 = 0$. Por tanto,

$$t_k = c_1 5^k + c_2 4^k + c_3 k 4^k + c_4 k^2 4^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 5^k + c_2 4^k + c_3 k 4^k + c_4 k^2 4^k$$

y después $n = 2^k$ ($k = \log n$), por tanto

$$\begin{aligned} T(n) &= c_1 5^{\log n} + c_2 4^{\log n} + c_3 \log n 4^{\log n} + c_4 \log^2 n 4^{\log n} = \\ &= c_1 n^{\log 5} + c_2 n^{\log 4} + c_3 \log n \cdot n^{\log 4} + c_4 \log^2 n \cdot n^{\log 4} = \\ &= c_1 n^{\log 5} + c_2 n^2 + c_3 n^2 \log n + c_4 n^2 \log^2 n. \end{aligned}$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de una y tenemos tres incógnitas, usamos la ecuación en recurrencia para obtener las otras dos:

$$\begin{aligned} T(2) &= 5T(1) + 2^2 = 9; \\ T(4) &= 5T(2) + 8^2 = 109; \\ T(8) &= 5T(4) + 24^2 = 1121. \end{aligned}$$

Con esto:

$$\left. \begin{aligned} 1 &= T(1) = c_1 1 + c_2 1 + c_3 1 \cdot 0 + c_4 1 \cdot 0 = c_1 + c_2 \\ 9 &= T(2) = c_1 5 + c_2 4 + c_3 4 \cdot 1 + c_4 4 \cdot 1 = 5c_1 + 4c_2 + 4c_3 + 4c_4 \\ 109 &= T(4) = c_1 25 + c_2 16 + c_3 16 \cdot 2 + c_4 16 \cdot 4 = 25c_1 + 16c_2 + 32c_3 + 64c_4 \\ 1121 &= T(8) = c_1 125 + c_2 64 + c_3 64 \cdot 3 + c_4 64 \cdot 9 = 125c_1 + 64c_2 + 192c_3 + 576c_4 \end{aligned} \right\}$$

Solucionando el sistema de ecuaciones obtenemos los valores de las constantes:

$$\left. \begin{aligned} c_1 &= 181 \\ c_2 &= -180 \\ c_3 &= -40 \\ c_4 &= -4 \end{aligned} \right\}$$

y sustituyéndolos en la ecuación anterior, obtenemos

$$T(n) = 181n^{\log 5} - 180n^2 - 40n^2 \log n - 4n^2 \log^2 n \in \Theta(n^2 \log^2 n).$$

i) $T(n) = T(n-1) + 2T(n-2) - 2T(n-3)$ si $n > 2$; $T(n) = 9n^2 - 15n + 106$ si $n=0,1,2$.

Reescribiendo la ecuación:

$$T(n) - T(n-1) - 2T(n-2) + 2T(n-3) = 0,$$

ecuación en recurrencia homogénea cuya ecuación característica asociada es

$$x^3 - x^2 - 2x + 2 = 0.$$

Resolviendo esa ecuación, sus raíces son 1, $\sqrt{2}$ y $-\sqrt{2}$, con lo cual

$$T(n) = c_1 + c_2(\sqrt{2})^n + c_3(-\sqrt{2})^n.$$

Para calcular las constantes necesitamos las condiciones iniciales:

$$\left. \begin{array}{l} 106 = T(0) = c_1 + c_2 + c_3 \\ 100 = T(1) = c_1 + c_2\sqrt{2} - c_3\sqrt{2} \\ 112 = T(2) = c_1 + 2c_2 + 2c_3 \end{array} \right\} \Rightarrow \left. \begin{array}{l} c_1 = 100 \\ c_2 = 3 \\ c_3 = 3 \end{array} \right\} \Rightarrow T(n) = 100 + 3\sqrt{2}^n (1 + (-1)^n)$$

En consecuencia, $T(n) \in \Theta(2^{n/2})$.

j) $T(n) = (3/2)T(n/2) - (1/2)T(n/4) - (1/n)$ si $n > 2$, n potencia de 2; $T(1)=1$; $T(2)=3/2$.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = (3/2)T(2^{k-1}) - (1/2)T(2^{k-2}) - (1/2)^k.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = (3/2)t_{k-1} - (1/2)t_{k-2} - (1/2)^k$$

ecuación en recurrencia no homogénea en la forma

$$t_k - (3/2)t_{k-1} + (1/2)t_{k-2} = -(1/2)^k,$$

cuya ecuación característica asociada es $(x-1)(x-1/2)^2 = 0$. Por tanto,

$$t_k = c_1 + c_2 2^{-k} + c_3 k 2^{-k}.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1 + c_2 2^{-k} + c_3 k 2^{-k}$$

después $n = 2^k$ ($k = \log n$), con lo cual

$$T(n) = c_1 + (1/n)c_2 + c_3(\log n/n).$$

Para calcular las constantes necesitamos las condiciones iniciales. Como sólo disponemos de dos y tenemos tres incógnitas, usamos la ecuación en recurrencia para obtener la tercera:

$$T(4) = (3/2)T(2) - (1/2)T(1) - (1/4) = 3/2.$$

Con esto:

$$\left. \begin{array}{l} 1 = T(1) = c_1 + c_2 \\ 3/2 = T(2) = c_1 + c_2(1/2) + c_3(1/2) \\ 3/2 = T(4) = c_1 + c_2(1/4) + c_3(1/2) \end{array} \right\} \Rightarrow \left. \begin{array}{l} c_1 = 1 \\ c_2 = 0 \\ c_3 = 1 \end{array} \right\} \Rightarrow T(n) = 1 + \frac{\log n}{n} \in \Theta(1).$$

k) $T(n) = 2T(n/4) + n^{1/2}$ si $n > 4$, n potencia de 4.

Haciendo el cambio $n = 2^k$ (o, lo que es igual, $k = \log n$) obtenemos

$$T(2^k) = 2T(2^{k-2}) + 2^{k/2}.$$

Llamando $t_k = T(2^k)$, la ecuación final es

$$t_k = 2t_{k-2} + 2^{k/2},$$

ecuación en recurrencia no homogénea de la forma

$$t_k - 2t_{k-2} = (\sqrt{2})^k$$

cuya ecuación característica asociada es $(x^2 - 2)(x - \sqrt{2}) = 0$, o lo que es igual, $(x + \sqrt{2})(x - \sqrt{2})^2 = 0$. Por tanto,

$$t_k = c_1(-\sqrt{2})^k + c_2(\sqrt{2})^k + c_3k(\sqrt{2})^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(2^k)$, con lo que

$$T(2^k) = c_1(-\sqrt{2})^k + c_2(\sqrt{2})^k + c_3k(\sqrt{2})^k$$

después $n = 2^k$ ($k = \log n$), y por tanto obtenemos:

$$T(n) = \sqrt{n} (c_1(-1)^{\log n} + c_2 + c_3 \log n).$$

Si n es múltiplo de 4 entonces $\log n$ es par, y por tanto $(-1)^{\log n}$ vale siempre 1. Esto nos permite afirmar, llamando $c_0 = c_1 + c_2$, que

$$T(n) = \sqrt{n} (c_0 + c_3 \log n).$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación original:

$$\sqrt{n} (c_0 + c_3 \log n) = 2(\sqrt{n}/2 (c_0 + c_3 \log n - 2c_3)) + \sqrt{n}.$$

Igualando los coeficientes de \sqrt{n} , $\sqrt{n} \log n$ y los términos independientes obtenemos $c_3 = 1/2$, de donde

$$T(n) = \sqrt{n} (c_0 + 1/2 \log n) \in \Theta(\sqrt{n} \log n).$$

Este problema también podría haberse solucionado mediante otro cambio, $n = 4^k$ (o, lo que es igual, $k = \log_4 n$) obteniendo la ecuación

$$T(4^k) = 2T(4^{k-1}) + 4^{k/2}.$$

Esta nos lleva, tras llamar $t_k = T(4^k)$, a la ecuación final

$$t_k = 2t_{k-1} + 2^k,$$

cuya resolución conduce a la misma solución que la obtenida mediante el primer cambio.

l) $T(n) = 4T(n/3) + n^2$ si $n > 3$, n potencia de 3.

Haciendo el cambio $n = 3^k$ (o, lo que es igual, $k = \log_3 n$) obtenemos que

$$T(3^k) = 4T(3^{k-1}) + 9^k.$$

Llamando $t_k = T(3^k)$, la ecuación final es

$$t_k = 4t_{k-1} + 9^k,$$

ecuación en recurrencia no homogénea de la forma

$$t_k - 4t_{k-1} = 9^k,$$

cuya ecuación característica asociada es $(x-4)(x-9) = 0$. Por tanto,

$$t_k = c_1 4^k + c_2 9^k.$$

Necesitamos ahora deshacer los cambios hechos. Primero $t_k = T(3^k)$, con lo que

$$T(3^k) = c_1 4^k + c_2 3^{2k}$$

y después $n = 3^k$ ($k = \log_3 n$), y por tanto

$$T(n) = c_1 4^{\log_3 n} + c_2 n^2 = c_1 n^{\log_3 4} + c_2 n^2.$$

De esta ecuación no conocemos condiciones iniciales para calcular todas las constantes, pero sí es posible intentar fijar alguna de ellas. Para eso, basta sustituir la expresión que hemos encontrado para $T(n)$ en la ecuación en recurrencia original, y obtenemos:

$$c_1 n^{\log_3 4} + c_2 n^2 = 4 \left(c_1 \left(\frac{n}{3} \right)^{\log_3 4} + c_2 \left(\frac{n}{3} \right)^2 \right) + n^2 = c_1 n^{\log_3 4} + \left(\frac{4}{9} c_2 + 1 \right) n^2$$

Igualando los coeficientes de $n^{\log_3 4}$ y de n^2 obtenemos $c_2 = 9/5$, de donde

$$T(n) = c_1 n^{\log_3 4} + \frac{9}{5} n^2.$$

Como $\log_3 4 < 2$, entonces $T(n) \in \Theta(n^2)$.

Solución al Problema 1.10

(☺)

a) Ciertamente. Se deduce de la propiedad 6 del apartado 1.3.1, pero veamos una posible demostración directa:

Si $T_1 \in O(f)$, sabemos que existen $c_1 > 0$ y n_1 tales que $T_1(n) \leq c_1 f(n)$ para todo $n \geq n_1$. Análogamente, como $T_2 \in O(f)$, existen $c_2 > 0$ y n_2 tales que $T_2(n) \leq c_2 f(n)$ para $n \geq n_2$. [1.1]

Para comprobar que $T_1 + T_2 \in O(f)$, debemos encontrar una constante real $c > 0$ y un número natural n_0 tales que $T_1(n) + T_2(n) \leq c f(n)$ para todo $n \geq n_0$. [1.2]

Apoyándonos en [1.1], basta tomar $n_0 = \max\{n_1, n_2\}$ y $c = c_1 + c_2$, con las que se verifica la ecuación [1.2] para todo $n \geq n_0$.

Existe otra forma de demostrarlo, utilizando límites en caso de que estos existan, como sucede por ejemplo cuando las funciones son continuas:

Si $T_1 \in O(f)$, entonces $\lim_{n \rightarrow \infty} \frac{T_1(n)}{f(n)} = k_1 < \infty$.

Análogamente, como $T_2 \in O(f)$, $\lim_{n \rightarrow \infty} \frac{T_2(n)}{f(n)} = k_2 < \infty$. [1.3]

Veamos entonces que $\lim_{n \rightarrow \infty} \frac{T_1(n) + T_2(n)}{f(n)} = k < \infty$. [1.4]

Pero [1.4] es cierto pues, como los dos límites en [1.3] son finitos y positivos podemos conmutar la suma con el límite y obtenemos que

$$\lim_{n \rightarrow \infty} \frac{T_1(n) + T_2(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{T_1(n)}{f(n)} + \lim_{n \rightarrow \infty} \frac{T_2(n)}{f(n)} = k_1 + k_2 < \infty.$$

b) Ciertamente.

Análogamente a lo realizado en el apartado anterior, si $T_1 \in O(f)$, entonces $\lim_{n \rightarrow \infty} \frac{T_1(n)}{f(n)} = k_1 < \infty$. Igualmente, como $T_2 \in O(f)$, $\lim_{n \rightarrow \infty} \frac{T_2(n)}{f(n)} = k_2 < \infty$. [1.5]

Veamos entonces que $\lim_{n \rightarrow \infty} \frac{T_1(n) - T_2(n)}{f(n)} = k < \infty$. [1.6]

Pero [1.6] es cierto pues, como los dos límites en [1.5] existen y son finitos y positivos podemos conmutar la resta con el límite y obtenemos que

$$\lim_{n \rightarrow \infty} \frac{T_1(n) - T_2(n)}{f(n)} = \lim_{n \rightarrow \infty} \frac{T_1(n)}{f(n)} - \lim_{n \rightarrow \infty} \frac{T_2(n)}{f(n)} = k_1 - k_2 < \infty.$$

c) Falso.

Consideremos $T_1(n) = n^2$, $T_2(n) = n$, y $f(n) = n^3$. Tenemos por tanto que $T_1 \in O(f)$ y $T_2 \in O(f)$, pero sin embargo $T_1(n)/T_2(n) = n \notin O(1)$.

d) Falso.

Consideremos de nuevo $T_1(n) = n^2$, $T_2(n) = n$, y $f(n) = n^3$. Tenemos por tanto que $T_1 \in O(f)$ y $T_2 \in O(f)$, pero sin embargo $T_1 \notin O(T_2)$ pues $n^2 \notin O(n)$.

Solución al Problema 1.11

(☺)

Sean $f(n) = n$ y $g(n) = \begin{cases} n^2, & \text{si } n \text{ es par.} \\ 1, & \text{si } n \text{ es impar.} \end{cases}$

Si n es impar, no podemos encontrar ninguna constante c tal que

$$f(n) = n \leq cg(n) = c,$$

y por tanto $f \notin O(g)$. Por otro lado, si n es par no podemos encontrar ninguna constante c tal que

$$g(n) = n^2 \leq cf(n) = cn,$$

y por tanto $g \notin O(f)$.

Solución al Problema 1.12

(☺)

Para comprobar que $\log^k n \in O(n)$ basta ver que

$$\lim_{n \rightarrow \infty} \frac{\log^k n}{n} = 0$$

para todo k . Pero eso es cierto siempre. Obsérvese además que por esa misma razón $\log^k n \notin \Omega(n)$ para cualquier $k > 0$.

Solución al Problema 1.13

Vamos a suponer que los tiempos de ejecución de las funciones *Esvacio*, *Izq*, *Der* y *Raiz* es de c operaciones elementales (OE), que el tiempo de ejecución de *Opera* es d OE, y el de *Max2* es 1 OE.

Procedimiento *Inorden*

(☺)

Para calcular el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan $2+c$ OE: la llamada a *Esvacio* (1 OE), el tiempo de ejecución de este procedimiento (c) y una negación.
- En la línea (2) se efectúa la llamada a *Izq* (1 OE), lo que tarda ésta en ejecutarse (c OE) más la llamada a *Inorden* (1 OE) y lo que tarde ésta en ejecutarse, que va a depender del número de elementos del árbol *Izq*(t).
- En la línea (3) se ejecutan $2+c+d$ OE: dos llamadas a procedimientos y sus respectivos tiempos de ejecución.
- El número de OE de la línea (4) se calcula de forma análoga a la línea (2): $2+c$ más lo que tarda *Inorden* en ejecutarse con el número de elementos que hay en *Der*(t).

Para estudiar el tiempo de ejecución, vamos a considerar dos casos extremos: que el árbol sea degenerado (es decir, una lista) y que sea equilibrado. Cualquier árbol se encuentra en una situación intermedia a estos dos casos.

- Si t es degenerado, podemos suponer sin pérdida de generalidad que *Esvacio*(*Izq*(t)) y que para todo a subárbol de t se verifica que *Esvacio*(*Izq*(a)). Por tanto, el número de OE que se realizan en la ejecución de *Inorden*(t) para un árbol t con n elementos es:

$$\begin{aligned} T(n) &= (2+c) + (2+c+T(0)) + (2+c+d) + (2+c+T(n-1)) = \\ &\quad 8+4c+d+T(0)+T(n-1). \\ T(0) &= 2+c. \end{aligned}$$

Con esto, $T(n) = 10 + 5c + d + T(n-1)$, ecuación en recurrencia no homogénea que podemos resolver desarrollándola telescópicamente:

$$\begin{aligned} T(n) &= 10 + 5c + d + T(n-1) = (10 + 5c + d) + (10 + 5c + d) + T(n-2) = \dots = \\ &\quad (10 + 5c + d)n + (2+c) \in \Theta(n) \end{aligned}$$

- Si t es equilibrado sus dos subárboles (izquierdo y derecho) tienen del orden de $n/2$ elementos y son a su vez equilibrados. Por tanto, el número de OE que se realizan en la ejecución de *Inorden*(t) para un árbol t con n elementos es:

$$\begin{aligned} T(n) &= (2+c) + (2+c+T(n/2)) + (2+c+d) + (2+c+T(n/2)) = 8+4c+d+2T(n/2). \\ T(0) &= 2+c. \end{aligned}$$

Para resolver esta ecuación en recurrencia se hace el cambio $t_k = T(2^k)$, con lo que obtenemos

$$t_k - 2t_{k-1} = 8 + 4c + d,$$

ecuación no homogénea con ecuación característica $(x-2)(x-1) = 0$. Por tanto,

$$t_k = c_1 2^k + c_2$$

y, deshaciendo los cambios,

$$T(n) = c_1 n + c_2.$$

Para calcular las constantes, nos apoyamos en la condición inicial $T(0)=2+c$, junto con el valor de $T(1)$, que puede ser calculado basándonos en la expresión de la ecuación en recurrencia: $T(1) = 8 + 4c + d + 2(2 + c)$, obteniendo

$$T(n) = (10 + 5c + d)n + (2+c) \in \Theta(n).$$

Función *Altura*

(☺)

Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan $1+c$ OE: la llamada a *Esvacio* (1 OE) más el tiempo de ejecución de este procedimiento (c OE).
- En la línea (2) se realiza 1 OE.
- En la línea (4) se efectúan:
 - a) la llamada a *Izq* (1 OE), lo que tarda ésta en ejecutarse (c OE) más la llamada a *Altura* (1 OE) y lo que tarde ésta en ejecutarse, que va a depender del número de elementos del árbol *Izq*(t); más
 - b) la llamada a *Der* (1 OE), lo que tarda ésta en ejecutarse (c OE) más la llamada a *Altura* (1 OE) y lo que tarde ésta en ejecutarse, que va a depender del número de elementos del árbol *Der*(t); más
 - c) el cálculo del máximo de ambos números (1 OE), un incremento (1 OE) y el *RETURN* (1 OE).

Para estudiar el tiempo de ejecución de esta función consideraremos los mismos casos que para la función *Inorden*: que el árbol sea degenerado (es decir, una lista) o que sea equilibrado.

- Si t es degenerado, podemos suponer sin pérdida de generalidad que *Esvacio*(*Izq*(t)) y que para todo a subárbol de t se verifica que *Esvacio*(*Izq*(a)). Por tanto, el número de OE que se realizan en la ejecución de *Altura*(t) para un árbol t con n elementos es:

$$T(n) = (1+c) + (1+c+1+T(0)) + 1+c+1+T(n-1) + 3 = 8 + 3c + T(0) + T(n-1). \\ T(0) = (1+c) + 1 = 2+c.$$

Con esto, $T(n)=10+4c+T(n-1)$, ecuación en recurrencia no homogénea que podemos resolver desarrollándola telescópicamente:

$$T(n) = 10 + 4c + T(n-1) = (10 + 4c) + (10 + 4c) + T(n-2) = \dots = \\ (10 + 4c)n + (2 + c) \in \Theta(n)$$

- Si t es equilibrado sus dos subárboles tienen del orden de $n/2$ elementos y son también equilibrados. Por tanto, el número de OE que se realizan en la ejecución de *Altura*(t) para un árbol t con n elementos es:

$$T(n) = (1+c) + (1+c+1+T(n/2)) + 1+c+1+T(n/2) + 3 = 8 + 3c + 2T(n/2). \\ T(0) = 2+c.$$

Para resolver esta ecuación en recurrencia se hace el cambio $t_k = T(2^k)$, con lo que obtenemos

$$t_k - 2t_{k-1} = 8 + 3c,$$

ecuación no homogénea de ecuación característica $(x-2)(x-1) = 0$. Por tanto,

$$t_k = c_1 2^k + c_2.$$

Deshaciendo los cambios,

$$T(n) = c_1 n + c_2.$$

Para calcular las constantes, nos apoyamos en la condición inicial $T(0) = 2 + c$, junto con el valor de $T(1)$, que puede ser calculado basándonos en la expresión de la ecuación en recurrencia: $T(1) = 8 + 3c + 2(2 + c)$. Finalmente obtenemos

$$T(n) = (10 + 4c)n + (2 + c) \in \Theta(n).$$

Función Mezcla

(S)

Para resolver este problema vamos a suponer que el tiempo de ejecución del procedimiento *Ins*, que inserta un elemento en un árbol binario de búsqueda, es $A \log n + B$, siendo A y B dos constantes. Supongamos también que n y m son el número de elementos de $t1$ y $t2$ respectivamente.

Para estudiar el tiempo de ejecución $T(n, m)$ consideraremos, al igual que hicimos para la función anterior, dos casos extremos: que el árbol $t2$ sea degenerado (es decir, una lista) o que sea equilibrado.

- Si $t2$ es degenerado, podemos suponer sin pérdida de generalidad que $Esvacio(Izq(t2))$ y que para todo a subárbol de $t2$ se verifica que $Esvacio(Izq(a))$. Por tanto, vamos a ver el número de OE que se realizan en cada línea de la función en este caso:
 - En la línea (1) se invoca a $Esvacio(t1)$, lo que supone $1 + c$ OE.
 - En la línea (2) se efectúa 1 OE.
 - Análogamente, las líneas (3) y (4) realizan $(1 + c)$ y 1 respectivamente.
 - Para estudiar el número de OE que realiza la línea (6), vamos a dividirla en cuatro partes:
 - a) $a1 := Ins(t1, Raiz(t2))$, siendo $a1$ una variable auxiliar para efectuar los cálculos. Se efectúan $2 + c + A \log n + B$ operaciones elementales: la llamada a $Raiz$ (1), el tiempo que ésta tarda (c), la llamada a Ins (1 OE), y su tiempo de ejecución ($A \log n + B$).
 - b) $a2 := Mezcla(a1, Izq(t2))$, siendo $a2$ una variable auxiliar para efectuar los cálculos. Se efectúan aquí $2 + c + T(n+1, 0)$ operaciones elementales: llamada a Izq (1), el tiempo que ésta tarda (c), la llamada a $Mezcla$ (1 OE), y su tiempo de ejecución, que será $T(n+1, 0)$, pues estamos suponiendo que $Esvacio(Izq(a))$ para todo a subárbol de $t2$.
 - c) $a3 := Mezcla(a2, Der(t2))$, siendo $a3$ una variable auxiliar para efectuar los cálculos. Se efectúan $2 + c + T(n+1, m-1)$ operaciones elementales: la

llamada a *Der* (1), el tiempo que ésta tarda (c), la llamada a *Mezcla* (1 OE), y su tiempo de ejecución, que será $T(n+1, m-1)$, pues estamos suponiendo que $Esvacio(Izq(a))$ para todo a subárbol de $t2$ o, lo que es igual, que el número de elementos de $Der(t)$ es $m-1$.

d) *RETURN a3*, que realiza 1 OE.

Por tanto, la ejecución de *Mezcla*($t1, t2$) en este caso es :

$$T(n, m) = 9 + 5c + B + A \log n + T(n+1, 0) + T(n+1, m-1)$$

con las condiciones iniciales $T(0, m) = 2 + c$ y $T(n, 0) = 3 + 2c$. Para resolver la ecuación en recurrencia podemos expresarla como:

$$T(n, m) = 12 + 7c + B + A \log n + T(n+1, m-1)$$

haciendo uso de la segunda condición inicial. Desarrollando telescópicamente la ecuación:

$$\begin{aligned} T(n, m) &= 12 + 7c + B + A \log n + T(n+1, m-1) = \\ &= (12 + 7c + B + A \log n) + (12 + 7c + B + A \log(n+1)) + T(n+2, m-2) = \\ &\dots\dots\dots \\ &= m(12 + 7c + B) + \left(\sum_{i=0}^{m-1} A \log(n+i) \right) + T(n+m, 0) = \\ &= m(12 + 7c + B) + 2c + 3 + A \left(\sum_{i=0}^{m-1} \log(n+i) \right). \end{aligned}$$

Pero como $\log(n+i) \leq \log(n+m)$ para todo $0 \leq i \leq m$,

$$T(n, m) \leq m(12 + 7c + B) + 2c + 3 + Am \log(n+m) \in O(m \log(n+m))$$

- El segundo caso es que $t2$ sea equilibrado, para el que se demuestra de forma análoga que

$$T(n, m) \in O(m \log(n+m)).$$

Solución al Problema 1.14

(☺)

Para comprobar que $O(f) \subset O(g)$, basta ver que $\lim_{n \rightarrow \infty} (f(n)/g(n)) = 0$ en cada caso pues las funciones son continuas, lo que implica la existencia de los límites. De esta forma se obtiene la siguiente ordenación:

$$\begin{aligned} O((1/3)^n) &\subset O(17) \subset O(\log \log n) \subset O(\log n) \subset O(\log^2 n) \subset O(\sqrt{n}) \subset O(\sqrt{n} \log^2 n) \\ &\subset O(n/\log n) \subset O(n) \subset O(n^2) \subset O((3/2)^n). \end{aligned}$$

Solución al Problema 1.15

(☹)

Para resolver la ecuación

$$T(n) = \frac{1}{n} \left(\sum_{i=0}^{n-1} T(i) \right) + cn,$$

siendo $T(0) = 0$, podemos reescribirla como:

$$nT(n) = \sum_{i=0}^{n-1} T(i) + cn^2 \quad [1.7]$$

Por otro lado, para $n-1$ obtenemos:

$$(n-1)T(n-1) = \sum_{i=0}^{n-2} T(i) + c(n-1)^2 \quad [1.8]$$

Restando [1.7] y [1.8]:

$$nT(n) - nT(n-1) + T(n-1) = T(n-1) + c(2n-1) \Rightarrow nT(n) = nT(n-1) + c(2n-1) \Rightarrow$$

$$T(n) = T(n-1) + c(2-1/n).$$

Desarrollando telescópicamente la ecuación en recurrencia:

$$\begin{aligned} T(n) &= T(n-1) + c(2 - 1/n) = \\ &= T(n-2) + c(2 - 1/(n-1)) + c(2 - 1/n) = \\ &= T(n-3) + c(2 - 1/(n-2)) + c(2 - 1/(n-1)) + c(2 - 1/n) = \\ &\dots\dots\dots \\ &= T(0) + c \sum_{i=1}^n \left(2 - \frac{1}{i} \right) = \\ &= c \sum_{i=1}^n \left(2 - \frac{1}{i} \right) \end{aligned}$$

ya que teníamos que $T(0) = 0$. Veamos cual es el orden de $T(n)$:

- a) Como $(2-1/i) \leq 2$ para todo $i > 0$, $T(n) \leq c \sum_{i=1}^n 2 = 2cn \Rightarrow T(n) \in O(n)$.
- b) Como $(2-1/i) \geq 1$ para todo $i > 0$, $T(n) \geq c \sum_{i=1}^n 1 = cn \Rightarrow T(n) \in \Omega(n)$.

Por tanto, $T(n) \in \Theta(n)$.

Solución al Problema 1.16

Función *BuscBin*

(☺)

a) Para determinar su tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan la comparación del *IF* (1 OE), y un acceso a un vector (1 OE), una comparación (1 OE) y un *RETURN* (1 OE) si la condición es verdadera.
- En la línea (3) se realizan 3 OE (suma, división y asignación).
- En la línea (4) hay un acceso a un vector (1 OE) y una comparación (1 OE), y además 1 OE en caso de que la condición del *IF* sea verdadera.
- En la línea (5) hay un acceso a un vector (1 OE) y una comparación (1 OE).
- Las líneas (6) y (8) efectúan $3 + T(n/2)$ cada una: una operación aritmética (incremento o decremento de 1), una llamada a la función *BuscBin* (lo que supone 1 OE), más lo que tarde en ejecutarse la función con la mitad de los elementos y un *RETURN* (1 OE).

Por tanto obtenemos la ecuación en recurrencia $T(n) = 11 + T(n/2)$, con la condición inicial $T(1) = 4$. Para resolverla, haciendo el cambio $t_k = T(2^k)$ obtenemos

$$t_k - t_{k-1} = 11,$$

ecuación no homogénea cuya ecuación característica es $(x-1)^2 = 0$. Por tanto,

$$t_k = c_1 k + c_2$$

y, deshaciendo los cambios,

$$T(n) = c_1 \log n + c_2.$$

Para calcular las constantes, nos basaremos en la condición inicial $T(1) = 4$, junto con el valor de $T(2)$, que podemos calcular apoyándonos en la expresión de la ecuación en recurrencia: $T(2) = 11 + 4 = 15$. Finalmente obtenemos

$$T(n) = 11 \log n + 4 \in \Theta(\log n)$$

b) La recursión de este programa, por tratarse de un caso de recursión de cola, puede ser eliminada mediante un bucle que simule las llamadas recursivas a la función. La condición de terminación del bucle puede ser tomada del caso base de la función recursiva y el cuerpo de dicho bucle consiste en una preparación de los argumentos de la función recursiva y el cálculo que ésta realiza:

```

PROCEDURE BuscBit(a:vector;prim,ult:CARDINAL;x:INTEGER):BOOLEAN;
  VAR mitad:CARDINAL;
BEGIN
  WHILE (prim<ult) DO
    mitad:=(prim+ult)DIV 2;
    IF x=a[mitad] THEN RETURN TRUE
    ELSIF (x<a[mitad]) THEN
      ult:=mitad-1
    ELSE

```

```

(* 1 *)
(* 2 *)
(* 3 *)
(* 4 *)
(* 5 *)
(* 6 *)

```

```

        prim:=mitad+1          (* 7 *)
    END                        (* 8 *)
END;                          (* 9 *)
RETURN x=a[ult]               (* 10 *)
END BuscBit;

```

c) Para el cálculo del tiempo de ejecución y la complejidad de la función no recursiva podemos seguir un proceso análogo al que seguimos para la función *Algoritmo2* (en el problema 1.5). Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan en cada una de las líneas:

- En la línea (1) se efectúa la condición del bucle, que supone 1 OE (la comparación).
- Las líneas (2) a (9) componen el cuerpo del bucle, y contabilizan 3, 2+1, 2, 2, 0, 2, 0 y 0 OE respectivamente.
- Por último, la línea (10) supone 3 OE. A ella se llega cuando la condición del bucle deja de verificarse.

El bucle se repite hasta que su condición sea falsa, acabando la función al ejecutarse la línea (10). Cada iteración del bucle está compuesta por las líneas (1) a (9), junto con una ejecución adicional de la línea (1) que es la que ocasiona la salida del bucle. En cada iteración se reduce a la mitad los elementos a considerar, por lo que el bucle se repite $\log n$ veces. Por tanto, en el peor caso,

$$T(n) = \left(\left(\sum_{i=1}^{\log n} (1 + 3 + 2 + 2 + 2) \right) + 1 \right) + 3 = 10 \log n + 4 \in \Theta(\log n).$$

Como puede verse, el tiempo de ejecución de ambas funciones es prácticamente igual, lo que a priori implica que cualquiera de las dos pueden usarse indistintamente. Sin embargo, hay que tener en cuenta la mayor complejidad espacial que siempre suponen los procedimientos recursivos por la utilización de la pila, lo que hace que ante una igualdad de tiempos de ejecución, los procedimientos iterativos sean preferibles frente a los recursivos. Pero no sólo la complejidad ha de ser tenida en cuenta para la elección del algoritmo. La claridad y sencillez del código es un factor también a considerar, pues ello va a implicar una mejor legibilidad y una depuración del programa y mantenimiento más fácil, aspectos todos ellos muy importantes.

Función *Sumadigitos*

(☺)

a) Para calcular el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan una comparación (1 OE) y un *RETURN* (1 OE) si la condición es verdadera.
- En la línea (2) se efectúa una división (1 OE), una llamada a la función *Sumadigitos* (1 OE), más lo que tarda ésta con un décimo del tamaño de su entrada, una suma (1 OE), un resto (1 OE), y un *RETURN* (1 OE).

Llamando n al parámetro *num* de la función, obtenemos la ecuación en recurrencia $T(n) = 6 + T(n/10)$, con la condición inicial $T(1) = 2$.

Para resolverla hacemos los cambios $n = 10^k$ (o, lo que es igual, $k = \log_{10} n$) y $t_k = T(10^k)$ y obtenemos

$$t_k - t_{k-1} = 6,$$

ecuación no homogénea cuya ecuación característica es $(x-1)^2 = 0$. Por tanto,

$$t_k = c_1 + c_2 k.$$

Deshaciendo los cambios,

$$T(n) = c_1 + c_2 \log_{10} n.$$

Para calcular las constantes, nos apoyamos en la condición inicial $T(1) = 2$, junto con el valor de $T(10)$, que puede ser calculado apoyándonos en la expresión de la ecuación en recurrencia: $T(10) = 6 + 2 = 8$. Finalmente obtenemos

$$T(n) = 6 \log_{10} n + 2 \in \Theta(\log n)$$

Como vemos, en este caso la complejidad de la función depende del logaritmo en base 10 de su parámetro *num* (esto es, de su número de dígitos).

- b) La recursión de este algoritmo puede ser eliminada mediante un bucle que simule las llamadas recursivas a la función, cuya condición de terminación puede ser tomada del caso base de la función recursiva, y cuyo cuerpo consiste en los cálculos que ésta realiza, junto con una preparación de los argumentos de la siguiente llamada. En concreto, el algoritmo que implementa el algoritmo no recursivo es el siguiente:

```

PROCEDURE Sumadigitos_it(num: CARDINAL): CARDINAL;
  VAR s: CARDINAL;
BEGIN
  s := num MOD 10;                                (* 1 *)
  WHILE num >= 10 DO                                (* 2 *)
    num := num DIV 10;                              (* 3 *)
    s := s + (num MOD 10)                          (* 4 *)
  END;                                              (* 5 *)
  RETURN s                                         (* 6 *)
END Sumadigitos_it;

```

- c) Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se ejecutan 2 OE (un resto y una asignación).
- En la línea (2) se efectúa la condición del bucle, que supone 1 OE.
- Las líneas (3) y (4) componen el cuerpo del bucle, y contabilizan 2 y 3 OE respectivamente.
- Por último, la línea (6) supone 1 OE. A ella se llega cuando la condición del bucle deja de verificarse.

Con esto, se efectúa la línea (1), y después se repite el bucle hasta que su condición sea falsa, acabando la función al ejecutarse la línea (6). Cada iteración del bucle está compuesta por las líneas (2) a (4), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. En cada iteración se diezman los elementos a considerar, por lo que el bucle se repite $\log_{10} n$ veces.

$$\text{Por tanto, } T(n) = 2 + \left(\left(\sum_{i=1}^{\log_{10} n} (1 + 2 + 3) \right) + 1 \right) + 1 = 6 \log_{10} n + 4 \in \Theta(\log n).$$

Llegados a este punto vemos que ocurre lo mismo que en el algoritmo anterior. Los tiempos de ejecución de las funciones recursiva e iterativa son similares. Es por tanto una decisión del usuario decantarse por el diseño generalmente más robusto ofrecido por los algoritmos recursivos, frente a la menor complejidad espacial que presentan los iterativos al no utilizar la pila de recursión.

Solución al Problema 1.17

(☺)

a) Para determinar el tiempo de ejecución del algoritmo, calcularemos primero el número de operaciones elementales (OE) que se realizan:

- En la línea (1) se puede ejecutar o sólo la condición (si ésta es falsa) con un total de 1 OE, o bien la sentencia entera, con un total de 3 OE.
- Las líneas (2) y (3) están compuestas por operaciones aritméticas y asignaciones, y se realizan un total de 3 OE en cada una.
- La línea (4) tiene tres partes: un acceso a $a[\text{mitad}]$, que supone 1 OE; la llamada a $Raro(a, \text{prim}, \text{prim} + \text{terc})$, que supone $2 + T(n/3)$ (1 de la suma de prim y terc , 1 OE de la llamada a $Raro$, y luego el tiempo de ejecución de $Raro$ para un tercio del número de elementos con los que fue invocada la función original); y la llamada a $Raro(a, \text{ult} - \text{terc}, \text{ult})$, que supone $2 + T(n/3)$ OE por la misma razón. El resultado de las tres partes ha de sumarse (lo que supone 2 OE) y luego hacer un $RETURN$ (1 OE). En resumen, en la línea (4) se ejecutan un total de $1 + 2 + T(n/3) + 2 + T(n/3) + 2 + 1 = 8 + 2T(n/3)$ OE.

Con esto:

1. Si $n = 1$ (caso base), se ejecuta sólo la línea (1) y por tanto $T(1) = 3$.
2. Si $n = 3^k$, con $k > 0$, se ejecuta sólo la condición del IF (1) y luego el resto de las líneas (2) a (4), con lo cual $T(n) = 15 + 2T(n/3)$.

Tenemos por tanto el tiempo de ejecución del algoritmo definido mediante una ecuación en recurrencia. Para resolverla, haciendo el cambio $n = 3^k$ queda

$$T(3^k) = 2T(3^{k-1}) + 15$$

y llamando t_k a $T(3^k)$ obtenemos

$$t_k = t_{k-1} + 15,$$

ecuación en recurrencia no homogénea de ecuación característica $(x-2)(x-1) = 0$ y consecuentemente

$$t_k = c_1 2^k + c_2 1^k = c_1 2^k + c_2.$$

Cambiando entonces t_k por $T(3^k)$ queda

$$T(3^k) = c_1 2^k + c_2,$$

y deshaciendo el cambio $n = 3^k$ (o, lo que es igual, $k = \log_3 n$), obtenemos finalmente

$$T(n) = c_1 2^{\log_3 n} + c_2 = c_1 n^{\log_3 2} + c_2.$$

Para calcular las constantes necesitamos resolver un sistema de dos ecuaciones con dos incógnitas (c_1 y c_2) basándonos en dos condiciones iniciales de la ecuación en recurrencia. Como de partida sólo disponemos de una ($T(1) = 3$), necesitamos obtener otra. Para ello, apoyándonos en la definición recursiva de T y para $n = 3$, obtenemos

$$T(3) = 15 + 2T(n/3) = 15 + 2T(1) = 21.$$

Por tanto

$$\left. \begin{array}{l} 3 = T(1) = c_1 + c_2 \\ 21 = T(3) = 2c_1 + c_2 \end{array} \right\} \Rightarrow \left. \begin{array}{l} c_1 = 18 \\ c_2 = -15 \end{array} \right\}$$

Sustituyendo estos valores en la ecuación, obtenemos finalmente

$$T(n) = 18n^{\log_3 2} - 15.$$

b) $T(n) \in \Theta(n^{\log_3 2})$.

Para justificarlo, basándonos en las propiedades de Θ , basta ver que

$$\lim_{n \rightarrow \infty} \frac{T(n)}{n^{\log_3 2}}$$

existe, es acotado y distinto de cero. Pero eso es cierto ya que

$$\lim_{n \rightarrow \infty} \frac{T(n)}{n^{\log_3 2}} = \lim_{n \rightarrow \infty} \frac{18n^{\log_3 2} - 15}{n^{\log_3 2}} = 18.$$

66 problemas resueltos de Análisis y Diseño de Algoritmos

Rosa Arruabarrena
Jesús Bermúdez

Informe interno: UPV/EHU /LSI / TR 8-99

REVISADO

15 de Octubre del 2000

PARTE I: Problemas

1. Supuesto que $\forall n \geq n_0 \ f(n) \geq g(n) \geq 0$ y que $f(n), g(n) \in \Theta(h(n))$, ¿qué puedes decir del orden de $f(n)-g(n)$?
2. Demuestra que $\forall a, b \ (a, b > 1 \Rightarrow \lg_a n \in \Theta(\lg_b n))$.
Si $a \neq b$, ¿es cierto $2^{\lg_a n} \in \Theta(2^{\lg_b n})$?
3. Justifica si son ciertas o falsas las afirmaciones siguientes, siendo $f(n)$ y $h(n)$ funciones de coste resultantes de analizar algoritmos:
 - (a) $O(f(n)) = O(h(n)) \Rightarrow O(\lg f(n)) = O(\lg h(n))$
 - (b) $O(\lg f(n)) = O(\lg h(n)) \Rightarrow O(f(n)) = O(h(n))$
4. Sea $t(n)$ el número de líneas generadas por una realización del procedimiento $G(n)$.

```
procedure G ( x: in INTEGER) is
begin
  for I in 1..x loop
    for J in 1..I loop
      PUT_LINE(I, J, x);
    end loop;
  end loop;
  if x > 0 then
    for I in 1..4 loop
      G(x div 2);
    end loop;
  end if;
end G;
```

Calcula el orden exacto de la función $t(n)$.

5. Un natural $n \geq 1$ es triangular si es la suma de una sucesión ascendente no nula de naturales consecutivos que comienza en 1. (Por tanto, los cinco primeros números triangulares son **1**, **3**=1+2, **6**=1+2+3, **10**=1+2+3+4, **15**=1+2+3+4+5.)
 - (a) Escribe un programa que, dado un entero positivo $n \geq 1$, decida si éste es un número triangular con eficiencia incluida en $O(n)$ y empleando un espacio extra de memoria constante.
 - (b) Analiza tu programa.

6. Supongamos que cada noche disponemos de una hora de CPU para ejecutar cierto programa y que con esa hora tenemos suficiente tiempo para ejecutar un programa con una entrada, a lo sumo, de tamaño $n = 1\,000\,000$. Pero el centro de cálculo tras una reasignación de tiempos decide asignarnos 3 horas diarias de CPU. Ahora, ¿cuál es el mayor tamaño de entrada que podrá gestionar nuestro programa, si su complejidad $T(n)$ fuera (para alguna constante k_i)?

- (a) $k_1 n$
- (b) $k_2 n^2$
- (c) $k_3 10^n$

7. Supongamos que cada noche disponemos de una hora de CPU para ejecutar cierto programa y que con esa hora tenemos suficiente tiempo para ejecutar un programa con una entrada, a lo sumo, de tamaño $n = 1\,000\,000$. En esta situación nuestro jefe compra una máquina 100 veces más rápida que la vieja. Ahora ¿cuál es el mayor tamaño de entrada que podrá gestionar nuestro programa en una hora, si su complejidad $T(n)$ fuera (para alguna constante k_i)?

- (a) $k_1 n$
- (b) $k_2 n^2$
- (c) $k_3 10^n$

8. Escribe un algoritmo que calcule los valores máximo y mínimo de un vector con n valores realizando para ello menos de $(3n/2)$ comparaciones entre dichos valores. Demuéstrese que la solución propuesta realiza menos comparaciones que las mencionadas.

9. Resuélvase la siguiente ecuación de recurrencia. ¿De qué orden es?

$$T(n) = \begin{cases} a & n = 1 \\ 2T\left(\frac{n}{4}\right) + \lg n & n > 1 \end{cases}$$

10. Calcula el orden temporal de los siguientes programas:

- (a)

```
function total(n:positivo)
  if n=1 then 1 else total(n-1) + 2 * parcial(n-1)

  siendo
    function parcial (m:positivo)
      if m=1 then 1 else 2 * parcial(m-1)
```
- (b)

```
function total(n,m:positivo)
  if n=1 then m else m + total (n-1, 2 * m)
```

11. El siguiente algoritmo es utilizado por muchos editores de texto. Busca la primera aparición de un string (esto es, de un array de caracteres $B(1..m)$ en el string $A(1..n)$), devolviendo el índice de A donde comienza B, si procede. El valor $Límite = n - m + 1$ es la posición más a la derecha en A donde podría comenzar B.

```

procedure   StringSearch  (A,B: in String; Hallado: out boolean;
                           Comienzo: out Indice) is
  N:= A'Length;      Encontrado:= false;      I, J : Indice;
  M:= B'Length;      Limite:= n-m+1;          Com:= A'First;
begin
  while not Encontrado and (Com ≤ Limite) loop
    I:= Com; J:= B'First;
    while J/= M+1 and then (A(i)=B(j)) loop
      I:= I+1; J:=J+1;
    end loop;
    Encontrado:= (J=M+1);
    if not Encontrado then Com:= Com+1; end if;
  end loop;
  Hallado:= Encontrado;
  Comienzo:= Com;
end StringSearch;

```

¿Cuántas veces se ejecuta la comparación $A(i)=B(j)$ en el peor caso? ¿Qué entradas dan lugar al peor caso? Obsérvese que, usando el lenguaje de programación Ada, el test sólo se comprueba una vez que es cierta la condición $J \neq M+1$.

12. Para ordenar el vector de n elementos $\langle e_1, e_2, \dots, e_n \rangle$ se utiliza una estrategia análoga al Mergesort pero dividiendo el vector en $n/2$ trozos de tamaño 2 y generalizando el Merge a $n/2$ secuencias.

- escribase la función de eficiencia temporal justificando cada uno de los sumandos de la misma, y
- determínese el orden.

13. Dado el algoritmo siguiente, que determina si una cadena C es palíndromo:

```

función PAL (C, i, j) devuelve booleano
  if i≥j then devuelve cierto
  elsif C(i)≠C(j) then devuelve falso
  else devuelve PAL(C, i+1, j-1)

```

Analiza la evaluación de $PAL(C, 1, n)$ en el caso peor y en el caso medio, suponiendo equiprobabilidad de todas las entradas y siendo $\{a, b\}$ el alfabeto que forma las cadenas.

14. Para resolver cierto problema se dispone de un algoritmo trivial cuyo tiempo de ejecución $t(n)$ -para problemas de tamaño n - es cuadrático (i.e. $t(n) \in \Theta(n^2)$). Se ha

encontrado una estrategia *Divide y Vencerás* para resolver el mismo problema; dicha estrategia realiza $D(n) = n \log n$ operaciones para dividir el problema en dos subproblemas de tamaño mitad y $C(n) = n \log n$ operaciones para componer una solución del original con la solución de dichos subproblemas. ¿Es la estrategia *Divide y Vencerás* más eficiente que la empleada en el algoritmo trivial?

15. Dado el siguiente algoritmo para calcular el máximo y el mínimo de un vector de enteros, determínese el N° DE COMPARACIONES entre componentes del vector que se realizan en el caso peor.

```

proc MAX_MIN (T[i..j], Max, Min)
  -- i ≤ j
  si T[i] • T[j] entonces Max:= T[j]; Min:= T[i];
  sino Max:= T[i]; Min:= T[j];
  fin si;
  si i+1 • j-1 entonces
    MAX_MIN (T[i+1..j-1], Aux_Max, Aux_Min);
    si Max < Aux_Max entonces Max:= Aux_Max; fin si;
    si Aux_Min < Min entonces Min:= Aux_Min; fin si;
  fin si;
fin proc;

```

16. Teniendo una secuencia de n claves distintas a ordenar, considérese la siguiente variación del algoritmo de Ordenación por Inserción:

Para $2 \leq i \leq n$: insertar $S(i)$ entre $S(1) \leq S(2) \leq \dots \leq S(i-1)$ empleando la búsqueda dicotómica para determinar la posición correcta donde debe ser insertado $S(i)$.

Es decir, el algoritmo de inserción para *determinar la posición* donde debe insertarse $S(i)$ hará uso del siguiente algoritmo:

```

Alg BUSQ_DICOT_POSICION(Sec, Valor) devuelve Posición
-- Valor ∉ Hacer_Cjto(Sec)
si Sec.Length=1 ent
  si Sec(Sec.First) < Valor ent devolver Sec.First+1
  else devolver Sec.First
sino si Sec((Sec.length)/2) < Valor ent
  devolver BUSQ_DICOT_POSICION(Sec((Sec.length)/2)+1
  ..Sec.Last),
  Valor)
sino devolver BUSQ_DICOT_POSICION(Sec(Sec.First..
  ((Sec.length)/2)-1)), Valor)

```

Ejemplos:

S								
2	5	8	18	20	30	Valor	...	S(n)
1	2	3	4	5	6	i	...	n

BUSQ_DICOT_POSICION(S(1..6), 1) → Posición: 1
 BUSQ_DICOT_POSICION(S(1..6), 3) → Posición: 2
 BUSQ_DICOT_POSICION(S(1..6), 9) → Posición: 4
 BUSQ_DICOT_POSICION(S(1..6), 37) → Posición: 7

Analizar lo siguiente de esta variante del algoritmo de Ordenación por Inserción.

- a) Comparaciones entre claves:
 - ¿Cuál es un caso peor?
 - ¿Cuál es el orden del número de comparaciones entre claves que se harán en ese peor caso?
- b) Movimientos de claves:
 - ¿Cuál es el caso peor?
 - ¿Cuántos movimientos de claves en total se harán en el peor caso?
- c) ¿Cuál es el orden del tiempo de ejecución del algoritmo en el peor caso?

17. Hemos diseñado la siguiente versión del algoritmo Mergesort con intención de evitar el uso de espacio extra de orden lineal. Analice su eficiencia temporal.

```

procedure MERGESORT_SOBRE_VECTOR (V: in out VECTOR,
                                   I,J: in INDICE) is
  K: INDICE := ⌊I+J/2⌋;
begin
  if ES_PEQUEÑO(I,J) then SIMPLE_SORT(V,I,J);
  else
    MERGESORT_SOBRE_VECTOR(V,I,K);
    MERGESORT_SOBRE_VECTOR(V,K+1,J);
    MERGE_SOBRE_VECTOR(V,I,K,J);
  end if;
end MERGESORT_SOBRE_VECTOR;

```

siendo MERGE_SOBRE_VECTOR(V,I,X,J) un procedimiento que ordena el segmento V(I..J) del vector V cuando recibe dos segmentos ordenados que ocupan posiciones consecutivas V(I..X) y V(X+1..J)

```

procedure MERGE_SOBRE_VECTOR (V: in out VECTOR, I,X,J: in INDICE) is
  Izq: INDICE := I; Der: INDICE := X+1;
  Aux: ELEM_VECTOR;
begin
  while Izq <= Der loop
    if V(Izq) > V(Der) then
      Aux := V(Izq);
      V(Izq) := V(Der);
      Insertar Aux en V(Der..J) dejándolo ordenado;
    end if;
    Izq := Izq+1;
  end loop;
end MERGESORT_SOBRE_VECTOR;

```

18. Suponga que vamos a utilizar el siguiente algoritmo para encontrar las k mayores claves de una lista de n claves.

```

function LAS_K_MAYORES_CLAVES (Vec: in VECTOR; K: in INTEGER)
  return VECTOR is
  M: VECTOR;
  Claves: VECTOR(1..K);
begin
  Crear montículo M con las n claves;
  for J in 1 .. K loop

```



```

        Claves(J) := M(1);
        M(1) := M(n-J+1);           -- trasladar el último a la raíz
        HUNDIR_RAIZ(M(1..n-J));
    end loop;
    return Claves;
end LAS_K_MAYORES_CLAVES;

```

¿ De qué orden (en función de n) puede ser k para que este algoritmo sea de $O(n)$?

19. El siguiente algoritmo puede emplearse para buscar un nombre en un listín telefónico (Obviamente el listado está ordenado alfabéticamente):

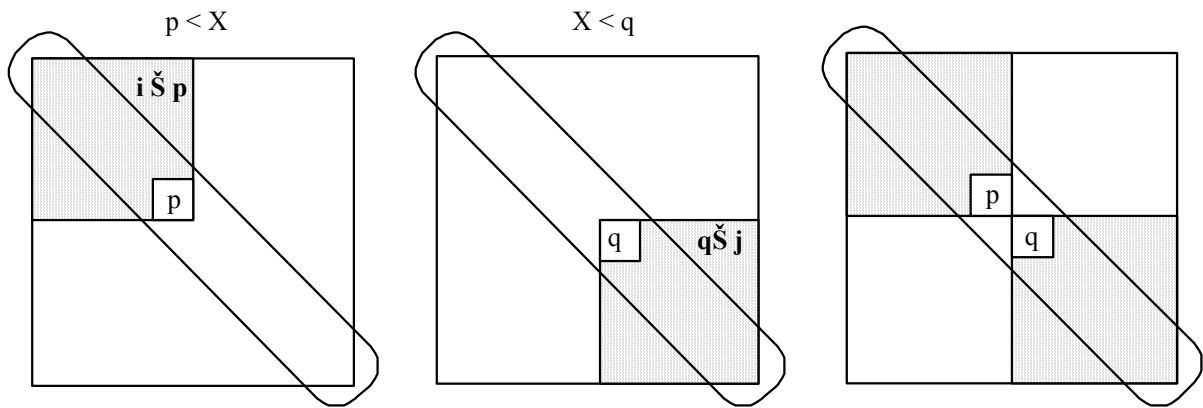
```

function BUSCO      (Lis: in Listado; Apellido: in String;
                    Salto: in Positive) return Integer is
-- Se supone que Apellido está en Lis.
-- Devuelve el índice donde se encuentra Apellido en Lis.
begin
    Indice:= Lis'First+Salto-1;
    while Lis'Last>Indice and then Apellido>Lis(Indice) loop
        Indice:= Indice+Salto;
    end loop;
    if Lis'Last>Indice
    then Hasta:= Indice;
    else Hasta:= Lis'Last;
    end if;
    return (BUSQUEDA_LINEAL(Lis(Indice-Salto+1..Hasta),Apellido));
end BUSCO;

```

- a) ¿Cuántas veces se ejecuta la comparación de Apellido con una componente de Lis en el peor caso? (Sirve la misma observación del ejercicio anterior.)
Sabemos que BUSQUEDA_LINEAL(Lis(X..Y),Apellido) produce $Y-X$ comparaciones.
- b) ¿Cambiaría el orden del algoritmo si en vez de BUSQUEDA_LINEAL invocásemos a BUSQUEDA_DICOTOMICA(Lis(X..Y),Apellido) que produce $1+\lfloor \log(Y-X) \rfloor$ comparaciones?

20. Para determinar si un ítem X se halla presente en una matriz cuadrada de ítems $T(1..n,1..n)$ cuyas filas y columnas están ordenadas crecientemente (de izquierda a derecha y de arriba a abajo respectivamente), se utiliza el siguiente algoritmo: (Sea benevolente con el enunciado, admitiendo que siempre tenga sentido eso de “la diagonal”)
- Con búsqueda dicotómica se busca X en la “diagonal”.
 - Si encontramos X , terminamos.
 - Si no, (encontramos p y q tales que $p < X < q$) descartamos los dos trozos de la matriz donde no puede estar X e invocamos recursivamente el mismo procedimiento sobre los dos trozos restantes.



¿De qué **orden** es este algoritmo en el **caso peor**?

21. Un array T contiene n números naturales distintos. Queremos conocer los menores m números de T . Sabemos que m es mucho menor que n . ¿Cómo lo harías para que resultara más eficiente?:

- (a) Ordenar T y tomar los m primeros.
- (b) Invocar SELECCIONAR(T, k) para $k=1, 2, \dots, m$.
- (c) Usar algún otro método.

Justifica tu respuesta contrastando en tu estudio los análisis de caso peor y caso medio conocidos.

22. Una persona piensa un número entero positivo W . Escribe un algoritmo para que otra persona lo adivine realizándole preguntas con la relaciones de orden: $<$, $>$, $=$. El número de preguntas debe ser $o(W)$.

23.

- (a) Analiza el algoritmo siguiente, que transforma el array $T(1..n)$ en un montículo.

```

proc crear_montículo ( $T(1..n)$ )
para  $i := \lfloor n/2 \rfloor$  disminuyendo hasta 1 hacer hundir( $T(1..n)$ ,  $i$ )

```

- (b) Con el análisis del apartado (a) habrás descubierto que crear_montículo($T(1..n)$) es de $O(n)$. Entonces, ¿por qué es incorrecto el razonamiento siguiente para analizar el algoritmo HeapSort?:

```

proc HeapSort ( $T(1..n)$ )
  crear_montículo ( $T(1..n)$ )
  para  $i := n$  disminuyendo hasta 2 hacer
    intercambiar  $T(1)$  y  $T(i)$ 
    hundir( $T(1..i-1)$ , 1)

```

Yo veo que `crear_montículo (T(1..n))` realiza $\lfloor n/2 \rfloor$ veces el procedimiento hundir, y que el bucle de $n-1$ iteraciones de `HeapSort(T(1..n))` puede verse como 2 veces $\lfloor n/2 \rfloor$ realizaciones del procedimiento hundir (más el intercambio que es de $O(1)$). Por tanto `HeapSort(T(1..n))` realiza consecutivamente tres fragmentos de programa de $O(n)$ cada uno y concluyo que `HeapSort(T(1..n))` es de $O(n)$.

24. Considérese el algoritmo de ordenación siguiente.

```

procedure TerciosSort (B: in out Vector; I,J: in Integer) is
    K: Integer :=  $\left\lfloor \frac{J-I+1}{3} \right\rfloor$ ;
begin
    if B(I) > B(J) then Intercambiar(B(I),B(J)); end if;
    if J-I+1 <= 2 then return; end if;
    TerciosSort (B, I, J-K);
    TerciosSort (B, I+K, J);
    TerciosSort (B, I, J-K);
end TerciosSort;

```

(a) Analiza su tiempo de ejecución.

(b) Compáralo con la eficiencia de *InsertionSort* y *HeapSort*.

Observaciones:

- Puedes considerar valores de n de la forma $(3/2)^k$.
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $\log_3 2 = 0.63092$

25. Completa el siguiente algoritmo para que calcule el k -ésimo mayor elemento de los n números enteros que el procedimiento **get-number** (X: **out** integer) irá generando. Nos interesa minimizar el espacio extra necesario, por tanto busca soluciones que no requieran siempre espacio extra de $\Omega(n)$.

```

for i := 1 to n loop
    get-number (X)
    ....
    ....
end loop
return K_ESIMO

```

26. Un mínimo local de un array $A(a-1...b+1)$ es un elemento $A(k)$ que satisface $A(k-1) \geq A(k) \leq A(k+1)$. Suponemos que $a \leq b$ y $A(a-1) \geq A(a)$ y $A(b) \leq A(b+1)$; estas condiciones garantizan la existencia de algún mínimo local. Diseña un algoritmo que encuentre algún mínimo local de $A(a-1...b+1)$ y que sea substancialmente más rápido que el evidente de $O(n)$ en el peor caso. ¿De qué orden es tu algoritmo?

27.

a) Considera el algoritmo recursivo de búsqueda dicotómica en un vector ordenado:

(Si X no estuviera en el vector devuelve el índice 0):

```

proc BUSQ_DICO (T[i..j], X, Ind)
  si i < j entonces
    k := (i+j) div 2;
    si T[k] = X entonces devolver Ind := k;
    sino
      si T[k] < X entonces BUSQ_DICO(T[k+1..j], X, Ind);
      sino BUSQ_DICO(T[i..k-1], X, Ind);
    fin si;
  fin si;
  sino si i=j y también T[i] = X
    entonces devolver Ind := i;
    sino devolver Ind := 0;
  fin si;
fin si;
fin proc;

```

Considérense dos implementaciones distintas del mismo algoritmo, en las cuales en una el paso de parámetros se realiza por referencia y en la otra por copia (o valor) ¿Habría alguna diferencia en la eficacia temporal entre esas dos implementaciones?

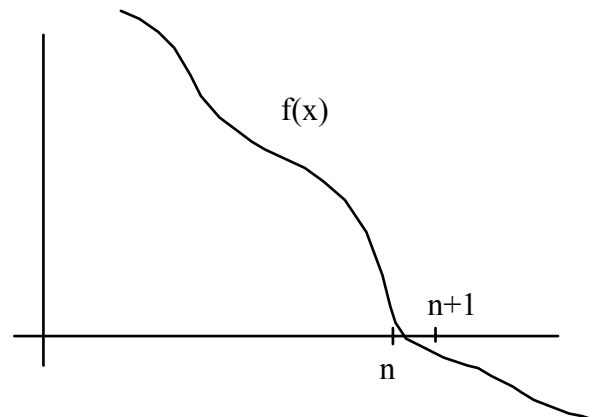
- b) En el Mergesort, ¿habría alguna diferencia en la eficacia temporal entre las dos posibilidades de paso de parámetro antes mencionadas?

28. Sea $F(x)$ una función monótona decreciente y sea N el mayor entero que cumple $F(N) \geq 0$. Asumiendo que N existe, un algoritmo para determinar dicho N es el siguiente:

```

I := 0;
while F(I) > 0 loop
  I := I + 1;
end loop
N = I-1;

```

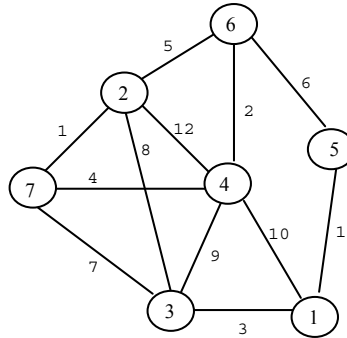


No obstante, esta solución tiene complejidad $O(N)$. Escribese un algoritmo cuyo comportamiento asintótico sea mejor en función de N .

29. Sea una red internacional de n ordenadores. Cada ordenador X puede comunicarse con cada uno de sus vecinos Y al precio de 1 doblón por comunicación. A través de las conexiones de Y y subsiguientes, X puede comunicarse con el resto de ordenadores de la red pagando a cada ordenador que utilice para la conexión el doblón correspondiente.

Describe un algoritmo que **calcule la tabla PRECIO(1..n)** tal que PRECIO(i) es el número mínimo de doblones que le cuesta al ordenador 1 establecer una conexión con el ordenador i

30. Dado el siguiente grafo:



- (a) Escribe, en orden de selección, las aristas seleccionadas por el algoritmo de *Prim* sobre ese grafo.
- (b) Lo mismo que el apartado (a) pero con el algoritmo de *Kruskal*.

31. Supóngase que un grafo tiene exactamente 2 aristas que tienen el mismo peso.

- (a) ¿Construye el algoritmo de *Prim* el mismo árbol de expansión mínimo, independientemente de cuál de esas aristas sea seleccionada antes?
- (b) Lo mismo que el apartado (a) pero con el algoritmo de *Kruskal*.

32. Estúdiese el siguiente algoritmo para calcular las componentes conexas de un grafo no dirigido $G=(\text{VERTICES}, \text{ARISTAS})$ usando la estructura de partición:

```

proc COMPONENTES_CONEXAS (G)
  -- Inicialización de la partición
  para cada vértice  $v \in \text{VERTICES}$  hacer Crear_Parte( $v$ ) fin para;
  para cada arista  $(x,y) \in \text{ARISTAS}$  hacer
    si BUSCAR( $x$ )  $\neq$  BUSCAR( $y$ )
      entonces UNIR(ETIQUETA( $x$ ), ETIQUETA( $y$ ))
    fin si;
  fin para;
fin proc;

```

Si G tiene K componentes conexas, ¿cuántas operaciones BUSCAR se realizan? ¿Cuántas operaciones UNIR se realizan? Exprésese el resultado en términos de $|\text{VERTICES}|$, $|\text{ARISTAS}|$ y K .

33. Dado un grafo G no dirigido, el algoritmo de Kruskal determina el árbol de expansión mínimo de G empleando para ello la estructura partición UNION-FIND. El método comienza ordenando el conjunto de aristas del grafo en orden creciente según sus pesos, para pasar a continuación a tratar una a una estas aristas. Analícese una nueva versión de dicho algoritmo en la cual se emplea un montículo (heap) para almacenar las aristas, y de donde se irán cogiendo una a una para ser tratadas de la misma forma que en la versión original. Escribese el algoritmo y calcúlese su orden en el caso peor.
34. Considérese la información registrada en una agencia de viajes referente a los vuelos existentes entre todos los aeropuertos del mundo. Supongamos que toda conexión aérea es de ida y vuelta. Queremos resolver el siguiente problema: Dados los aeropuertos A y B ¿Cuál es el mínimo número de transbordos en un viaje aéreo de A a B ?
- (a) De los algoritmos de recorrido de grafos conocidos, cuál elegirías para resolver este problema y por qué. Indica cual sería la modificación pertinente del algoritmo elegido.
 - (b) ¿Cuál es la complejidad temporal de cada uno de los algoritmos sobre grafos que conoces que resuelvan este problema?
 - (c) En conclusión, cuál de ellos elegirías y por qué.
- 35.- Supongamos que tenemos un algoritmo $CAM(G,v)$ que, dado un grafo G con pesos no negativos asociados a las aristas y un vértice v del mismo, nos devuelve las distancias mínimas para ir de v al resto de vértices del grafo G . ¿Es cierto $CAM(G,v) = CAM(KRUSKAL(G),v)$? Recuerdese que $KRUSKAL(G)$ es un árbol de expansión mínimo de G .
36. Tras haber visto el algoritmo de Dijkstra, el cual dado un grafo dirigido determina los costes de los caminos mínimos desde un vértice al resto, se pide modificar dicho algoritmo preservando el orden de manera que:
- (a) calcule además el NUMERO de caminos mínimos y
 - (b) calcule lo suficiente para reconstruir esos caminos mínimos.

Justificar las respuestas.

[Pista:

Para que el algoritmo de Dijkstra original, además de determinar los costes de los caminos mínimos, calcule los propios caminos (secuencias de vértices), basta con añadirle otra tabla $CAMINOS(1..n)$ que acumule dichos caminos de la siguiente forma: En $CAMINOS(x)$ se registra la lista de vértices que preceden inmediatamente al vértice x en caminos mínimos de 1 a x . Por ejemplo si $CAMINOS(x) = [x_1, x_2, x_3]$

esto significa que (x_1, x) , (x_2, x) y (x_3, x) son últimas aristas respectivas de caminos mínimos distintos de 1 a x .]

37. El problema de encontrar un subconjunto T de aristas de un grafo conexo G de manera que todos los vértices del grafo queden conectados empleando tan sólo las aristas de T , y la suma de los pesos de las aristas de T sea la menor posible sigue teniendo sentido aún cuando el grafo G tenga aristas con pesos negativos. Sin embargo, la solución puede que ya no sea un árbol. Adáptese el algoritmo de Kruskal para que trabaje con grafos cuyas aristas pueden tener pesos negativos.
38. Sea un grafo dirigido acíclico con pesos no negativos en los arcos. Queremos un algoritmo que calcule la máxima distancia de un vértice origen a cada uno de los demás vértices. ¿Sería correcto un algoritmo voraz con la selección voraz siguiente?: Sea S el conjunto de vértices seleccionados (inicialmente $S = \{\text{Origen}\}$). Seleccionar el vértice $v \notin S$ cuyo *camino especial* sea de longitud máxima.
(La noción de *camino especial* es la análoga a la usada en el algoritmo de Dijkstra de cálculo de caminos mínimos.)
39. Sea un grafo dirigido con pesos mayores o iguales que cero en los arcos. En el algoritmo de Dijkstra, sea w un nodo fuera del conjunto de seleccionados S tras haber añadido un nuevo nodo v a S . ¿Es posible que algún camino especial mínimo del origen a w pase por v y después por algún otro nodo de S ? La respuesta debe quedar justificada matemáticamente.
40. Escriba y analice un algoritmo que, dado un árbol T , determine la profundidad K con el máximo número de nodos de T . Si hubiera varias de esas profundidades, determínese la mayor.

41.- Sea la función recursiva

$$f(n, m) = \begin{cases} 0 & \text{si } m = 0 \\ n & \text{si } m = 1 \\ f\left(n, \left\lfloor \frac{m}{2} \right\rfloor\right) + f\left(n, \left\lceil \frac{m}{2} \right\rceil\right) & \text{si } m > 1 \end{cases}$$

- (a) Es evidente que un programa recursivo ingenuo que calcule $f(n, m)$ repetirá cálculos. Al efecto de evitar estos cálculos redundantes, escribe el programa que calcula dicha función empleando para su diseño la técnica de programación dinámica.
- Utilizando funciones con memoria.

- Utilizando una solución iterativa.

(b) Analiza la complejidad de las soluciones del apartado anterior. ¿Encuentras alguna diferencia en el número de sumas a realizar?

42. ¿Qué técnica usarías para escribir un programa eficiente que calcule $f(n) = n + \frac{2}{n} \sum_{i=1}^{n-1} f(i)$ siendo $f(1)=1$? Justifica brevemente de qué orden sería ese programa.

43.- Sea A: array (1..n) of integer con $n>0$, y sea la función

```
function SUMA?(A(1..i)) return boolean is
-- i>0
  if i=1 ^ A(1)≠0 then return false;
  elseif A(i)=suma(A(1..i-1)) then return true;
  else return SUMA?(A(1..i-1));
```

siendo

```
function suma (A(1..k)) return integer is
  s:integer:=0;
  for j:=1 to k loop s:=s+A(j) end loop
  return s;
```

La evaluación de SUMA?(A(1..n)) devuelve un valor booleano que indica si alguno de los elementos del vector A(1..n) coincide con la suma de todos los elementos que le preceden. Analice la eficiencia de esta evaluación.

Utilice otra técnica para escribir un programa que resuelva el problema de un modo más eficiente. Analice la eficiencia de su propuesta.

44.- Diseña y analiza un algoritmo que calcule la **clausura transitiva** \mathfrak{R} de una relación binaria R. Dada una relación binaria R, $x\mathfrak{R}y \Leftrightarrow xRy \vee (\exists z. x\mathfrak{R}z \wedge z\mathfrak{R}y)$

45. Describe un algoritmo de **programación dinámica** para el problema de selección de actividades: Subconjunto de máxima cardinalidad de los intervalos $[s_i, f_i]$ $1 \leq i \leq n$, y sin solapamientos dos a dos. Se basará en el cálculo **iterativo** de NUM_i para $i=1,2,...,n$ donde NUM_i es el máximo número de actividades mutuamente compatibles del conjunto de actividades $\{1,2,...,i\}$. Supóngase que la entrada viene ordenada ascendentemente según los tiempos de finalización: $f_1 \leq f_2 \leq \dots \leq f_n$

Compara el tiempo que necesita este algoritmo frente al de la solución voraz: Selecciona el que antes termine y no se solape con los ya seleccionados.

46. Tenemos n objetos de pesos p_1, \dots, p_n y valores v_1, \dots, v_n y una mochila de capacidad C . Escriba y analice un algoritmo que determine un subconjunto de objetos cuyo peso total no exceda la capacidad de la mochila y cuyo valor total sea maximal. Todas las cantidades C , v_i y p_i para $i=1, \dots, n$ son números naturales.

(No interesan soluciones que empleen el esquema de Backtracking.)

47. Dadas n clases de monedas v_1, v_2, \dots, v_n y sabiendo que de cada clase se disponen de tantas monedas como se precisen, escribase un algoritmo que determine cuantas combinaciones **distintas** de monedas hay para devolver una cierta cantidad C .

Ej.: Si las clases de monedas fueran: $v_1 = 25$, $v_2 = 50$, $v_3 = 75$ y $v_4 = 100$

Cantidad	Combinaciones distintas
100	5 formas: $25+75$, $2 \cdot 50$, 100 , $4 \cdot 25$, $50 + 2 \cdot 25$
116	0, no hay ninguna combinación

Observación: La combinación $25+75$ es la misma que $75+25$ y por ello es una única combinación a considerar.

48. Una *subsecuencia* de una secuencia S se obtiene retirando de S cualquier número de elementos de cualesquiera posiciones. Por ejemplo: $acxb$ es una subsecuencia de $bbacabxxb$.

La función recursiva f siguiente define (por casos) una *subsecuencia común de longitud maximal* de dos secuencias:

$$f(R, \epsilon) = \epsilon$$

$$f(\epsilon, S) = \epsilon$$

$$f(a \cdot R, a \cdot S) = a \cdot f(R, S)$$

$$f(a \cdot R, b \cdot S) = \text{la de mayor longitud entre } f(R, b \cdot S) \text{ y } f(a \cdot R, S), \quad (a \neq b)$$

Por ejemplo:

$$f(\text{promotor}, \text{prometedor}) = \text{promtor},$$

$$f(\text{aturdido}, \text{tranquilo}) = \text{trio},$$

$$f(\text{si}, \text{no}) = \epsilon.$$

Diseña un algoritmo, utilizando la técnica de programación dinámica (sin usar funciones con memoria), que calcule la longitud de la subsecuencia $f(R, S)$. ¿De qué orden es el tiempo y espacio empleado por tu solución?

49. Sea $G=(V,A)$ un grafo orientado con pesos no negativos en los arcos y $|V|=n$. Sea L la matriz de adyacencia que lo representa. Queremos calcular la distancia mínima entre cada par de vértices $i,j \in V$, usando la siguiente fórmula:

D_{ij}^p = distancia mínima de i a j cuando podemos dar, a lo sumo, p pasos.

Tenemos los valores básicos $D_{ij}^1 = L_{ij}$ y para todos los i y j buscamos D_{ij}^{p-1} sabiendo que D_{ij}^p se define recursivamente: $D_{ij}^p = \min_k \text{adyacente a } i \{D_{ij}^{p-1}, L_{ik} + D_{kj}^{p-1}\}$

Resuelve el problema expuesto empleando la técnica de la programación dinámica (sin funciones con memoria), la cual deberá recoger la definición recursiva arriba expuesta; es decir, se pide:

- determinar la estructura de almacenamiento de las soluciones parciales que se va a emplear: vector o matriz, cada posición de la estructura qué recoge o representa, cuál es la inicialización de la estructura, cómo se irá rellenando, dónde está la solución.
- algoritmo iterativo que calcula la definición recursiva sobre la estructura escogida y descrita en el apartado (a).
- análisis del orden del coste temporal y espacial: $O(T(n))$ y $O(MEE(n))$.

- 50.- Sea el siguiente programa, con X e Y números positivos:

```
function PRINCIPAL (Num: positive) return real is
  Tabla: array (integer range 1.. Num)
                                of real := (1=>X; 2=>Y; others=>0);
  function AUX (Num: natural) return real is
  begin
    if Tabla(Num-1)=0
    then Tabla(Num-1) := AUX(Num-1);
    end if;
    if Tabla(Num-2)=0
    then Tabla(Num-2) := AUX(Num-2);
    end if;
    Tabla(Num) := (Tabla(Num-1)+Tabla(Num-2)) / 2
  end AUX;
begin
  if Num<=2 then return Tabla(Num);
  else return AUX(Num);
  end if;
end PRINCIPAL;
```

- Define la función que calcula $PRINCIPAL(n)$.
- Indica qué técnica se ha usado en el diseño del programa anterior.
- Analiza el tiempo de ejecución de $PRINCIPAL(n)$.
- ¿Usarías este algoritmo si como datos de entrada tuviéramos $X=1$ e $Y=1$?
¿Por qué?

51.- Escribe un programa que decida si un natural dado N es producto de tres naturales consecutivos y cuyo orden temporal sea $o(n)$. Calcula su orden temporal.

52. Es conocido que la versión clásica del algoritmo $QuickSort(T(1..n))$ es de $\Theta(n^2)$ en el caso peor. Usando los siguientes algoritmos, ambos de $\Theta(n)$ en el caso peor: $Mediana(T(1..n), M)$ que calcula la mediana M del vector de enteros $T(1..n)$ y $Clasificar(T(1..n), P)$ que permuta los elementos de $T(1..n)$ de manera que, al final:

si $T(i) < P$ y $T(j) = P$ entonces $i < j$
 si $T(j) = P$ y $T(k) > P$ entonces $j < k$

escribe y analiza otra versión de $QuickSort(T(1..n))$ cuya complejidad temporal sea $o(n^2)$ en el caso peor.

53. El algoritmo de Floyd calcula la distancia mínima entre cada par de nodos de un grafo orientado con pesos no negativos asociados a los arcos. Modifícalo para que además calcule el número de caminos con distancia mínima que hay entre cada par de nodos.

54. Imagina un robot situado sobre unos raíles sin fin, con posibilidad de movimiento de un paso a derecha o izquierda con $actualizar(situacion, sentido)$ y con posibilidad de advertir la presencia de un objeto buscado con $esta_en(situacion)$. El siguiente algoritmo mueve “pendularmente” al robot en busca del objeto citado, partiendo del origen 0.

```

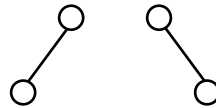
limite:= 1
sentido:= derecha
loop
  situacion:= 0
  while situacion<limite loop
    actualizar(situacion, sentido)
    if esta_en(situacion)
      then return (situacion, sentido)
    end if
  end loop
  sentido:= cambiar(sentido)
  for i in 1..limite loop
    actualizar(situacion, sentido)
  end loop
  limite:= 2*limite
end loop

```

Sabiendo que el objeto será encontrado, analiza el número de veces que se realizará la operación $actualizar(situacion, sentido)$ en función de la distancia del objeto al origen.

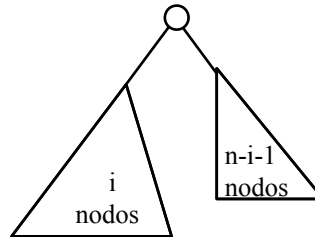
55.- ¿Cuántos árboles binarios distintos se pueden formar con n nodos? Sea $C(n)$ ese número.

Sabemos que $C(0)=1$ -el árbol vacío-, $C(1)=1$ -un árbol con sólo raíz- y $C(2)= 2$ los árboles:



Puedes dibujar los árboles para calcular $C(3)$ y $C(4)$.

- (a) Encuentra una definición recursiva para calcular $C(n)$ considerando el siguiente diagrama:



Puedes comprobarla con tus dibujos para $n=3,4$.

- (b) Escribe el algoritmo que calcule $C(n)$ siguiendo tu recurrencia con la técnica de *Programación Dinámica* e indica su complejidad.

56. Escribe la recurrencia que sirve de fundamento para la aplicación de la técnica de programación dinámica al problema de minimización del número de productos escalares en la multiplicación de una serie de matrices $A_1A_2...A_n$.

El algoritmo que resuelve este problema calcula una tabla $\text{factor}(1..n,1..n)$ tal que $\text{factor}(i,j)=k$ si la factorización óptima es $(A_i...A_k)(A_{k+1}...A_j)$. Escribe y analiza un algoritmo que, a partir de $\text{factor}(1..n,1..n)$, imprima la parentización óptima del producto $A_1A_2...A_n$.

57. Di de qué orden es la siguiente recurrencia:

$$T(n) = \begin{cases} 1 & n \leq 4 \\ T\left(\frac{n}{4}\right) + \sqrt{n} + 1 & n > 4 \end{cases}$$

58. Es conocido que $O(n) \subset O(n * \lg n) \subset O(n * \sqrt{n}) \subset O(n^2) \subset O(n^3)$. Clasifica $O(T(n))$ en esa cadena de inclusiones siendo:

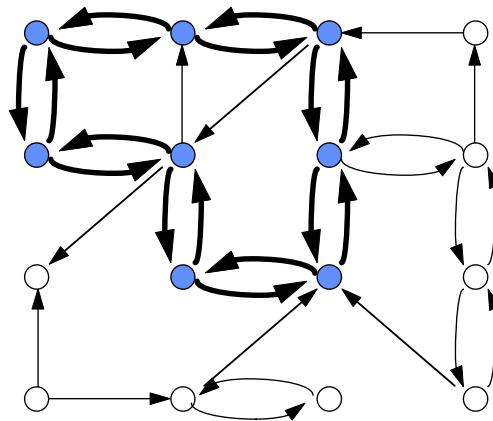
$$T(n) = \begin{cases} 1 & n \leq 1 \\ 4 * T\left(\frac{n}{3}\right) + n & n > 1 \end{cases}$$

Tal vez te resulte útil saber que: $\log_3 4 = 1.26184$

59. Escribe un algoritmo que dados un vector de n enteros y un entero X , determine si existen en el vector dos números cuya suma sea X . El tiempo de tu algoritmo debe ser de $O(n \cdot \lg n)$. Analiza tu algoritmo y demuestra que es así.

60. El **cuadrado** de un grafo dirigido $G = (V, A)$ es el grafo $G^2 = (V, B)$ tal que $(u, w) \in B$ si y sólo si para algún $v \in V$ tenemos que $(u, v) \in A$ y $(v, w) \in A$. Es decir, G^2 tiene un arco de u a w cuando G tiene un camino de longitud (exactamente) 2 de u a w . Describe algoritmos eficientes que calculen G^2 a partir de G para las dos representaciones conocidas: matriz de adyacencia y lista de adyacencias. Analiza el tiempo de tus algoritmos.

61. Dado un grafo dirigido, llamamos **anillo** a una serie de tres o más nodos conectados en forma de ciclo en los dos sentidos. Por ejemplo, en el grafo dirigido de la figura



se ha destacado el único anillo que contiene. Inspirándote en el método del recorrido en profundidad, diseña y analiza un algoritmo eficiente que indique si un grafo dirigido contiene o no al menos un anillo.

62. Recuerda que $\text{Fib}(0) = 0$, $\text{Fib}(1) = 1$ y $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$ para $n \geq 2$.

1. Supón que el coste de sumar, restar y multiplicar dos números es $O(1)$, independientemente del tamaño de los números.

Escribe y analiza un algoritmo que calcule $\text{Fib}(n)$ en tiempo $O(n)$.

2. Escribe y analiza otro algoritmo que calcule $\text{Fib}(n)$ en tiempo $O(\lg n)$ usando suma y multiplicación.

(Ayuda: Considera la siguiente matriz y sus potencias: $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} \text{fib}_0 & 1 \\ \text{fib}_1 & 1 \end{pmatrix}$)

3. Supón ahora que para sumar dos números de β bits se necesita un tiempo en $O(\beta)$ y que para multiplicarlos necesitamos tiempo en $O(\beta^2)$.

Analiza tu algoritmo del apartado (4.1) tomando el número de bits como tamaño de la entrada y considerando el correspondiente coste de las operaciones aritméticas.

Recuerda que
$$\text{Fib}(n) \approx \left(\frac{1 + \sqrt{5}}{2} \right)^n.$$

63. Dada una cantidad M ¿cuál es el máximo número X , con $X \leq M$, que podemos calcular a partir de una cantidad inicial C y aplicando repetidamente las operaciones $*2$, $*3$, $*5$? Emplea la técnica de la programación dinámica para determinar dicho número X y analiza el orden del algoritmo propuesto.

64.

- Justifica que el algoritmo típico que usas para multiplicar dos números enteros A y B realiza $O(m \cdot n)$ multiplicaciones *elementales* y $O(m \cdot n)$ sumas *elementales*, siendo m y n el número de dígitos de A y B respectivamente. (Llamamos *elemental* a la multiplicación (resp. suma) de dos dígitos decimales.)
- En realidad, las operaciones $A \times 10$, $\lfloor B/10 \rfloor$ y $B \bmod 10$ pueden considerarse de $O(1)$ (consiste simplemente en añadir, eliminar o seleccionar una cifra decimal). Por tanto para realizarlas no necesitamos multiplicaciones ni sumas. Analiza el número de multiplicaciones elementales (resp. el número de sumas elementales) que se realizan con el siguiente algoritmo de multiplicación:

```
función Multiplicar (A,B)
    si B=0 entonces resultado 0
    sino resultado A*(B mod 10)+Multiplicar(Ax10, ⌊B/10⌋)
```

donde $+$ y $*$ son las operaciones de suma y multiplicación típicas consideradas en el apartado (4.1). Fíjate que las operaciones “ $\times 10$ ”, “ $*$ ”, y “Multiplicar” son tres operaciones diferentes.

65. Supongamos n clases de monedas. La moneda de clase i (para cada $i=1..n$) es de valor v_i y tenemos exactamente c_i monedas de esa clase. Escribe y analiza un algoritmo que determine de cuántas formas **distintas** podemos sumar un valor M con esas monedas.

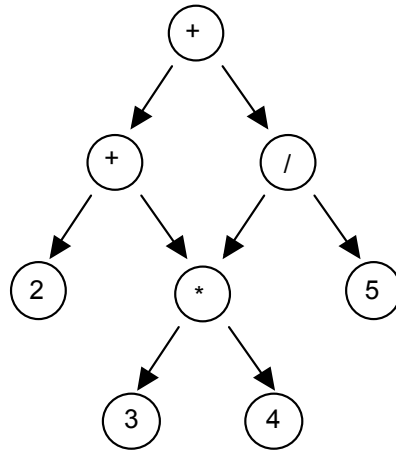
Ej.: Si las clases de monedas fueran dos con las siguientes cantidades:

$$v_1 = 25, c_1 = 10; \quad v_2 = 100, c_2 = 2$$

Valor	Formas distintas
250	3 formas: $2 \cdot 100 + 2 \cdot 25$, $1 \cdot 100 + 6 \cdot 25$, $10 \cdot 25$
300	2 formas: $2 \cdot 100 + 4 \cdot 25$, $1 \cdot 100 + 8 \cdot 25$

La combinación $25+100$ es la misma que $100+25$, y por ello es una única a considerar.

66. Una expresión aritmética puede representarse mediante un DAG (gafo dirigido acíclico). Esta representación es más compacta que la de árbol, porque las subexpresiones repetidas sólo tienen que aparecer una vez. Por ejemplo, la expresión $2+3*4+(3*4)/5$ podría representarse mediante el DAG de la figura. Cada operador (o nodo interno) tiene exactamente dos operandos (o arcos). Escribe y analiza un algoritmo lineal en el número de nodos que evalúe la expresión aritmética dada por un DAG de estas características.



PARTE II: Soluciones

1. Supuesto que $\forall n \geq n_0 \ f(n) \geq g(n) \geq 0$ y que $f(n), g(n) \in \Theta(h(n))$, ¿qué puedes decir del orden de $f(n)-g(n)$?

ℳ:

Sin conocer la definición de las funciones, todos los comentarios han de ser hipotéticos. Si existieran los límites siguientes

$$\left. \begin{array}{l} \bullet \lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} = a \\ \bullet \lim_{n \rightarrow \infty} \frac{g(n)}{h(n)} = b \end{array} \right\}$$

Obsérvese que:

$$\lim_{n \rightarrow \infty} \frac{f(n) - g(n)}{h(n)} = \lim_{n \rightarrow \infty} \frac{f(n)}{h(n)} - \lim_{n \rightarrow \infty} \frac{g(n)}{h(n)} = a - b \text{ pudiendo concluir ahora:}$$

- 1) $a=b \Rightarrow f(n)-g(n) \in O(h(n))$
- 2) $a>b \Rightarrow f(n)-g(n) \in \Theta(h(n))$

Pero obsérvese el ejemplo siguiente: si $f(n)=2^n+2$ y $g(n)=2^n+(-1)^n$ entonces $f(n)-g(n) \in O(1)$; y si sustituimos en $f(n)$ el 2 por una función $p(n) \in o(f(n))$ entonces $f(n)-g(n) \in O(p(n))$.

2. Demuestra que $\forall a, b \ (a, b > 1 \Rightarrow \lg_a n \in \Theta(\lg_b n))$.

Si $a \neq b$, ¿es cierto $2^{\lg_a n} \in \Theta(2^{\lg_b n})$?

ℳ:

$$a) \lim_{n \rightarrow \infty} \frac{\lg_a n}{\lg_b n} = \lim_{n \rightarrow \infty} \frac{\frac{\ln n}{\ln a}}{\frac{\ln n}{\ln b}} = \frac{\ln b}{\ln a} \in \mathbb{R}^+ \text{ ya que } a, b > 1. \text{ Y de aquí tenemos directamente}$$

$$\Theta(\lg_a n) = \Theta(\lg_b n)$$

$$b) \text{ Contraejemplo: } a=2 \text{ y } b=4 \text{ puesto que } 2^{\lg_2 n} = n \notin \Theta(2^{\lg_4 n}) = \Theta(2^{\lg_2 n^{\frac{1}{2}}}) = \Theta(\sqrt{n})$$

3. Justifica si son ciertas o falsas las afirmaciones siguientes, siendo $f(n)$ y $h(n)$ funciones de coste resultantes de analizar algoritmos:

- (a) $O(f(n)) = O(h(n)) \Rightarrow O(\lg f(n)) = O(\lg h(n))$
- (b) $O(\lg f(n)) = O(\lg h(n)) \Rightarrow O(f(n)) = O(h(n))$

ℳ:

$$(a) \ O(f(n)) = O(h(n)) \Rightarrow O(\lg f(n)) = O(\lg h(n))$$

Justificamos $O(\lg f(n)) \subseteq O(\lg h(n))$ y la inclusión inversa es análoga

$g(n) \in O(\lg f(n))$

$$\Rightarrow \exists k \in \mathbb{R}^+ \quad \forall n \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq k \lg f(n)$$

$$\Rightarrow \exists k \in \mathbb{R}^+ \quad \forall n \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq k \lg f(n) \leq k \lg (c h(n))$$

por hipótesis

$$\Rightarrow \exists k \in \mathbb{R}^+ \quad \forall n \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq k \lg c + k \lg h(n)$$

$$\Rightarrow \exists k \in \mathbb{R}^+ \quad \forall n \in \mathbb{N} \quad \forall n \geq n_0 \quad g(n) \leq (1+k) \lg h(n)$$

[*]

$$[*] \quad \forall n \geq n_1 \quad \text{siendo } n_1 \in \mathbb{N} \quad \text{t.q. } k \lg c \leq \lg h(n)$$

$$\Rightarrow g(n) \in O(\lg h(n))$$

(b) La implicación es falsa. Contraejemplo: $f(n) = 2^n$ y $h(n) = 3^n$

4. Sea $t(n)$ el número de líneas generadas por una realización del procedimiento $G(n)$.

```

procedure G ( x: in INTEGER) is
begin
  for I in 1..x loop
    for J in 1..I loop
      PUT_LINE(I,J,x);
    end loop;
  end loop;
  if x>0 then
    for I in 1..4 loop
      G(x div 2);
    end loop;
  end if;
end G;

```

Calcula el orden exacto de la función $t(n)$.

⚡:

$T(n)$ = nº de líneas escritas por el algoritmo:

$$T(0) = 0$$

$$T(1) = 1$$

$$T(n) = \sum_{i=1}^n \sum_{j=1}^i 1 + 4T(n/2) = \sum_{i=1}^n i + 4T(n/2)$$

Puesto que nos piden calcular el orden del algoritmo y no el nº exacto de líneas que escribe, calcularemos el orden de la siguiente función $f(n)$ cuyo orden es el mismo que el de $T(n)$:

$$f(1) = a_1$$

$$f(n) = n^2 + 4 f(n/2)$$

$$= n^2 + 4 \left((n/2)^2 + 4 f(n/2^2) \right) = 2n^2 + 4^2 f(n/2^2)$$

$$= 2n^2 + 4^2 \left((n/2^2)^2 + 4 f(n/2^3) \right) = 3n^2 + 4^3 f(n/2^3)$$

$$= \dots = i n^2 + 4^i f(n/2^i) \quad \text{tras } i \text{ pasos: } n = 2^i; \quad i = \lg n; \quad n^2 = 4^i$$

$$= n^2 \lg n + n^2 f(1) = n^2 \lg n + n^2 a_1 \in \Theta(n^2 \lg n)$$

5. Un natural $n \geq 1$ es triangular si es la suma de una sucesión ascendente no nula de naturales consecutivos que comienza en 1. (Por tanto, los cinco primeros números triangulares son **1**, **3**=1+2, **6**=1+2+3, **10**=1+2+3+4, **15**=1+2+3+4+5.)

(a) Escribe un programa que, dado un entero positivo $n \geq 1$, decida si éste es un número triangular con eficiencia incluida en $O(n)$ y empleando un espacio extra de memoria constante.

(b) Analiza tu programa.

:

Para comprobar que un número N es triangular podemos ir calculando uno a uno y ordenadamente por su valor los números triangulares. En cada paso:

- 1) generaremos el siguiente número triangular, NT .
- 2) si $NT=N$, habremos acabado con éxito;
si no si $NT>N$, habremos terminado con fracaso;
si no sucede que $NT<N$ y N aún puede ser un número triangular y se procede como en el paso 1).

```
function TRIANGULAR? (N: positive) return boolean is
  NT:positive:=1; Ultimo_Sumando:=1;
begin
  while NT<N loop
    Ultimo_Sumando:= Ultimo_Sumando + 1;
    NT:= NT + Ultimo_Sumando;
  end loop;
  return (NT=N);
end function TRIANGULAR?;
```

Las operaciones del bucle requieren tiempo constante y éste se realiza hasta que NT es igual o mayor que N . Esto es, el número de veces que se realiza este bucle es igual al número de sumandos que tiene la sucesión ascendente no nula de naturales consecutivos que comienza en 1 y cuya suma es $N = \sum_{i=1}^K i = \frac{(1+K)K}{2}$. Así pues, si despejamos K de la ecuación $K^2+K-2N=0$, obtenemos el orden del número de veces que se ejecuta el bucle: $\Theta(\sqrt{N})$, tanto si consideramos que hemos salido del bucle con éxito como con fracaso.

Observación: si se ha salido del bucle por $NT>N$ eso significa que en la iteración anterior NT aún era menor que N y tras realizar una suma más ha sucedido $NT>N$, pero el orden algoritmo sigue siendo el mismo.

Con respecto al espacio de memoria extra empleado, puesto que tan sólo se han empleado dos variables locales (NT y $Ultimo_Sumando$), es de tamaño constante es $\Theta(1)$.

6. Supongamos que cada noche disponemos de una hora de CPU para ejecutar cierto programa y que con esa hora tenemos suficiente tiempo para ejecutar un programa con una entrada, a lo sumo, de tamaño $n=1\,000\,000$. Pero el centro de cálculo tras una reasignación de tiempos decide asignarnos 3 horas diarias de CPU. Ahora, ¿cuál es el mayor tamaño de entrada que podrá gestionar nuestro programa, si su complejidad $T(n)$ fuera (para alguna constante k_i)?

- (a) $k_1 n$
- (b) $k_2 n^2$
- (c) $k_3 10^n$

✎:

$T(n)$ representa el tiempo para procesar entradas de tamaño n .

Para determinar el tamaño de entrada más grande que va a poder gestionar la nueva máquina, basta con despejar n de las respectivas inecuaciones

$$T_i(n) < T_i(1\ 000\ 000) \times 3$$

$$(a) T_1(n) < T_1(1\ 000\ 000) \times 3$$

$$k_1 n < k_1 1\ 000\ 000 \times 3 = 3 k_1 10^6 \quad \Rightarrow \quad n < 3 \cdot 10^6$$

$$(b) T_2(n) < T_2(1\ 000\ 000) \times 3$$

$$k_2 n^2 < k_2 (1\ 000\ 000)^2 \times 3 = 3 k_2 10^{12} \quad \Rightarrow \quad n < \sqrt{3} \cdot 10^6$$

$$(c) T_3(n) < T_3(1\ 000\ 000) \times 3$$

$$k_3 10^n < k_3 (10)^{10^6} \times 3 \Rightarrow n < \lg_{10} 3 + 10^6$$

7. Supongamos que cada noche disponemos de una hora de CPU para ejecutar cierto programa y que con esa hora tenemos suficiente tiempo para ejecutar un programa con una entrada, a lo sumo, de tamaño $n = 1\ 000\ 000$. En esta situación nuestro jefe compra una máquina 100 veces más rápida que la vieja. Ahora ¿cuál es el mayor tamaño de entrada que podrá gestionar nuestro programa en una hora, si su complejidad $T(n)$ fuera (para alguna constante k_i)?

- (a) $k_1 n$
- (b) $k_2 n^2$
- (c) $k_3 10^n$

✎:

El tiempo $t(n)$ se divide ahora por 100, y el tiempo de que disponemos es $t(10^6)$

Luego, para determinar el tamaño de entrada más grande que va a poder gestionar la nueva máquina, basta con despejar n de las respectivas inecuaciones

$$(T_i(n)/100) < T_i(1\ 000\ 000)$$

$$(a) T_1(n) < T_1(1\ 000\ 000) \times 100$$

$$k_1 n < k_1 1\ 000\ 000 \times 100 = k_1 10^8 \\ n < 10^8$$

$$(b) T_2(n) < T_2(1\ 000\ 000) \times 100$$

$$k_2 n^2 < k_1 (1\ 000\ 000)^2 \times 100 = k_2 10^{14} \\ n < 10^7$$

$$\begin{aligned}
(c) \quad T_3(n) &< T_3(1\,000\,000) \times 100 \\
k_3 \cdot 10^n &< k_3 (10)^{10^6} \times (10)^2 \\
10^n &< (10)^{10^6} \times (10)^2 = (10)^{10^6+2} \\
n &< 1\,000\,002
\end{aligned}$$

Interpretación de los resultados: cuanto más eficiente sea el algoritmo, al aumentar la potencia del ordenador, la ganancia en nº de operaciones que se podrán realizar en el mismo tiempo es mayor. Así, mientras en el caso lineal (a) se observa que de $n=10^6$ se pasa a poder resolver instancias con tamaño $n=10^8$, en el caso cuadrático (b) la ganancia es menor, de $n=10^6$ a $n=10^7$; finalmente observamos que en el caso (c) la ganancia es mínima, de $n=10^6$ a $n=10^6+2$.

8. Escribe un algoritmo que calcule los valores máximo y mínimo de un vector con n valores realizando para ello menos de $(3n/2)$ comparaciones entre dichos valores. Demuéstrese que la solución propuesta realiza menos comparaciones que las mencionadas.

En:

```

procedure MAX_MIN (B: in Vector, MAX, MIN: out Valor) is
    MAX1, MAX2, MIN1, MIN2: Valor;
    MID: Indice := B'First + (B'Last - B'First) div 2;
begin
    if B'Length <= 2 then
        if B(B'First) < B(B'Last)
            then MAX := B(B'Last); MIN := B(B'First);
            else MAX := B(B'First); MIN := B(B'Last);
            end if;
        else
            MAX_MIN (B(B'First, MID), MAX1, MIN1);
            MAX_MIN (B(MID, B'Last), MAX2, MIN2);
            if MAX1 > MAX2 then MAX := MAX1 else MAX := MAX2;
            if MIN1 < MIN2 then MIN := MIN1 else MIN := MIN2;
            end if;
        end if;
    end;

```

El número de comparaciones entre elementos del vector que realiza el algoritmo Max_Min propuesto se recoge en la siguiente ecuación de recurrencia: Suponiendo que n es potencia de 2

$$T(n) = \begin{cases} 1 & \text{si } n \leq 2 \\ 2T\left(\frac{n}{2}\right) + 2 & \text{si } n > 2 \end{cases}$$

El número total concreto se obtiene fácilmente expandiendo dicha ecuación:

$$T(n) = 2T\left(\frac{n}{2}\right) + 2 = \dots = 2^i T\left(\frac{n}{2^i}\right) + \sum_{k=1}^i 2^k \quad \text{con } \frac{n}{2^i} = 2; \frac{n}{2} = 2^i$$

Luego:

$$T(n) = 2^i \cdot 1 + \frac{2^{i+1} - 2}{1} = \frac{n}{2} + n - 2 = \frac{3}{2}n - 2$$

Así pues, el número de comparaciones entre valores del vector realizadas por la solución propuesta es menor a $(3/2)n$.

9. Resuélvase la siguiente ecuación de recurrencia. ¿De qué orden es?

$$T(n) = \begin{cases} a & n = 1 \\ 2T\left(\frac{n}{4}\right) + \lg n & n > 1 \end{cases}$$

✎:

$$\begin{aligned} T(n) &= 2 T(n/4) + \lg n \\ &= 2 (2 T(n/4^2) \lg(n/4)) + \lg n = 2^2 T(n/4^2) + 2 \lg(n/4) + \lg n \\ &= 2^2 (2T(n/4^3) + \lg(n/4^2)) + 2 \lg(n/4) + \lg n = 2^3 T(n/4^3) + 2^2 \lg(n/4^2) + 2 \lg(n/4) + \lg n \\ &= \dots = 2^i T(n/4^i) + 2^{i-1} \lg(n/4^{i-1}) + \dots + 2^2 \lg(n/4^2) + 2 \lg(n/4) + \lg n \\ &= 2^i T(n/4^i) + \sum_{k=0}^{i-1} 2^k \lg\left(\frac{n}{4^k}\right) = 2^i T(n/4^i) + \lg n \sum_{k=0}^{i-1} 2^k - \sum_{k=0}^{i-1} 2^k \lg 4^k \\ &\quad \bullet \text{ puesto que todos los logaritmos son del mismo orden, tomaremos logaritmos en base 4.} \\ &\quad \bullet \quad n=4^i; \lg_4 n = i; \sqrt{n} = 2^i \\ &= 2^i T(1) + \lg_4 n \sum_{k=0}^{i-1} 2^k - \sum_{k=0}^{i-1} 2^k \lg 4^k \\ &= 2^i \cdot 1 + \lg_4 n (2^i - 1) - (i-2) 2^i - 2 = 3 \sqrt{n} - \lg_4 n - 2 \in \Theta(\sqrt{n}) \end{aligned}$$

10. Calcula el orden temporal de los siguientes programas:

```
(a)      function total(n:positivo)
          if n=1 then 1 else total(n-1) + 2 * parcial(n-1)

          siendo
              function parcial (m:positivo)
                  if m=1 then 1 else 2 * parcial(m-1)

(b)      function total(n,m:positivo)
          if n=1 then m else m + total (n-1, 2 * m)
```

✎:

$$\begin{aligned} \text{(a)} \quad T_{\text{parcial}}(m) &= b + T_{\text{parcial}}(m-1) \in \Theta(m) \\ T_{\text{total}}(n) &= a + T_{\text{total}}(n-1) + T_{\text{parcial}}(n-1) \\ &= a + \Theta(n) + T_{\text{total}}(n-1) \in \Theta(n^2) \end{aligned}$$

(b) Obsérvese que el tamaño del argumento m no afecta si consideramos elemental la operación producto.

$$\begin{aligned} T_{\text{total}}(n) &= c_1 & \text{si } n=1 \\ &= c_2 + T_{\text{total}}(n-1) & \text{c.c} \end{aligned}$$

Por el método de expansión fácilmente se obtiene: $T_{\text{total}}(n) \in \Theta(n)$

11. El siguiente algoritmo es utilizado por muchos editores de texto. Busca la primera aparición de un string (esto es, de un array de caracteres B(1..m) en el string A(1..n)), devolviendo el índice de A donde comienza B, si procede. El valor $Limite = n - m + 1$ es la posición más a la derecha en A donde podría comenzar B.

```

procedure   StringSearch  (A,B: in String; Hallado: out boolean;
                           Comienzo: out Indice) is
  N:= A'Length;      Encontrado:= false;      I, J : Indice;
  M:= B'Length;      Limite:= n-m+1;          Com:= A'First;
begin
  while not Encontrado and (Com ≤ Limite) loop
    I:= Com; J:= B'First;
    while J/= M+1 and then (A(i)=B(j)) loop
      I:= I+1; J:=J+1;
    end loop;
    Encontrado:= (J=M+1);
    if not Encontrado then Com:= Com+1; end if;
  end loop;
  Hallado:= Encontrado;
  Comienzo:= Com;
end StringSearch;

```

¿Cuántas veces se ejecuta la comparación $A(i)=B(j)$ en el peor caso? ¿Qué entradas dan lugar al peor caso? Obsérvese que, usando el lenguaje de programación Ada, el test sólo se comprueba una vez que es cierta la condición $J \neq M+1$.

✎:

La secuencia B de caracteres puede comenzar en A en $(n-m+1)$ posiciones distintas, ocurriendo el caso peor cuando B se compara completamente (los m caracteres de B) a partir de cada una de ellas. Esto requerirá en el peor caso $((n-m+1) m)$ comparaciones entre caracteres de A y de B.

Las entradas que dan lugar al peor caso son:

$$\forall i, j (1 \leq i, j \leq n \Rightarrow A(i) = A(j)) \wedge \forall k, \forall j (1 \leq k \leq m-1 \wedge 1 \leq j \leq n \Rightarrow B(k) = A(j)) \wedge B(m) \neq A(1)$$

Un caso concreto sería, por ejemplo, $A(1..7)=aaaaaaa$ y $B(1..3)=aab$

12. Para ordenar el vector de n elementos $\langle e_1, e_2, \dots, e_n \rangle$ se utiliza una estrategia análoga al Mergesort pero dividiendo el vector en $n/2$ trozos de tamaño 2 y generalizando el Merge a $n/2$ secuencias.

- escribase la función de eficiencia temporal justificando cada uno de los sumandos de la misma, y
- determínese el orden.

✎:

Sea $t(n)$ la función de eficiencia temporal del algoritmo que sigue la estrategia marcada por el enunciado.

La parte recurrente de esta función debe considerar los sumandos siguientes:

- (a) Para resolver $n/2$ trozos de tamaño 2: $(n/2)*t(2)$.
 - (b) Para la división del problema de tamaño n en $n/2$ subproblemas de tamaño 2: $O(n)$.
 - (c) Para la mezcla de los $n/2$ trozos en un vector de tamaño n :
Siguiendo la analogía con el Mergesort clásico, debemos calcular -en cada iteración- el mínimo de $n/2$ elementos (los que están al frente de los $n/2$ trozos ordenados), ese cálculo es de $O(n)$ si lo realizamos con el algoritmo trivial, y como hay que realizar $O(n)$ iteraciones de esas, el resultado es $O(n^2)$.
- En resumen, y simplificando los sumandos; $t(n) \in O(n^2)$.

Observación importante: El sumando (c) puede mejorarse hasta $O(n \lg n)$ si para calcular el mínimo de los elementos correspondientes utilizamos una estructura de montículo.

13. Dado el algoritmo siguiente, que determina si una cadena C es palíndromo:

```
función PAL (C, i, j) devuelve booleano
  if  $i \geq j$  then devuelve cierto
  elseif  $C(i) \neq C(j)$  then devuelve falso
  else devuelve PAL(C, i+1, j-1)
```

Analiza la evaluación de $PAL(C, 1, n)$ en el caso peor y en el caso medio, suponiendo equiprobabilidad de todas las entradas y siendo $\{a, b\}$ el alfabeto que forma las cadenas.

⚡:

Sea $n=j-i+1$ el tamaño de la cadena. Estudiaremos, como operación característica, el número de comparaciones entre componentes ($C(i) \neq C(j)$). Es fácil ver que, en el caso peor, ese número viene dado por la función siguiente:

$$f(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ + f(n-2) & \text{si } n > 1 \end{cases}$$

que resolviendo por expansión resulta ser $f(n) = \left\lfloor \frac{n}{2} \right\rfloor$.

En el caso medio, la equiprobabilidad de las entradas hace que la probabilidad de que los extremos sean distintos es $\frac{1}{2}$ ya que el alfabeto es $\{a, b\}$, y en ese caso el número de comparaciones que se realizan es 1; en caso de que sean iguales - cuya probabilidad es $\frac{1}{2}$ - se producirán el número promedio de comparaciones para una cadena de tamaño $(n-2)$ más 1. Es decir:

$$t(n) = \begin{cases} 0 & \text{si } n \leq 1 \\ \frac{1}{2} + \frac{1}{2}(1 + t(n-2)) & \text{si } n > 1 \end{cases}$$

$$t(n) = 1 + \frac{1}{2} t(n-2) = \dots = \sum_{k=0}^{i-1} \frac{1}{2^k} + \frac{1}{2^i} t(n-2i) \text{ que cuando } i = \frac{n}{2} \text{ resulta } t(n) = 2 - \frac{1}{2^{i-1}} = 2 - \frac{2}{\sqrt{2}^n} \in O(1)$$

14. Para resolver cierto problema se dispone de un algoritmo trivial cuyo tiempo de ejecución $t(n)$ -para problemas de tamaño n - es cuadrático (i.e. $t(n) \in \Theta(n^2)$). Se ha encontrado una estrategia *Divide y Vencerás* para resolver el mismo problema; dicha estrategia realiza $D(n) = n \log n$ operaciones para dividir el problema en dos subproblemas de tamaño mitad y $C(n) = n \log n$ operaciones para componer una solución del original con la solución de dichos subproblemas. ¿Es la estrategia *Divide y Vencerás* más eficiente que la empleada en el algoritmo trivial?

Res:

La ecuación de recurrencia que recoge el tiempo que precisa el algoritmo nuevo es:

$$T(n) = \begin{cases} a & \text{si } n \leq 1 \\ 2T(n/2) + 2n \lg n & \text{si } n > 1 \end{cases}$$

Resolvemos la ecuación:

$$\begin{aligned} T(n) &= 2T(n/2) + 2n \lg n \\ &= 2^2 T(n/2^2) + 2n \lg n/2 + 2n \lg n \\ &= 2^3 T(n/2^3) + 2n \lg n/2^2 + 2n \lg n/2 + 2n \lg n \\ &= 2^i T(n/2^i) + 2n \sum_{j=0}^{i-1} \lg n/2^j \\ &= nT(1) + 2n \lg n \sum_{j=0}^{i-1} 1 - 2n \sum_{j=0}^{i-1} \lg 2^j \\ &= an + 2n \lg n i - 2n \sum_{j=1}^{i-1} j = an + 2n \lg n i - 2n \frac{i(i-1)}{2} \\ &= an + 2n (\lg n)^2 - n (\lg n)^2 - n \lg n = an + n (\lg n)^2 - n \lg n \in \Theta(n (\lg n)^2) \end{aligned}$$

Y puesto que

$$\lim_{n \rightarrow \infty} \frac{n (\lg n)^2}{n^2} = \lim_{n \rightarrow \infty} \frac{(\lg n)^2}{n} = \lim_{x \rightarrow \infty} \frac{2 (\lg x) \frac{1}{x} \ln 2}{1} = \lim_{x \rightarrow \infty} \frac{2 (\lg x)}{x \ln 2} = 0$$

concluimos con $\Theta(n (\lg n)^2) \subseteq \Theta(n^2)$; i.e., la nueva versión es más eficiente.

15. Dado el siguiente algoritmo para calcular el máximo y el mínimo de un vector de enteros, determínese el N° DE COMPARACIONES entre componentes del vector que se realizan en el caso peor.

```

proc MAX_MIN (T[i..j], Max, Min)
  -- i ≤ j
  si T[i] • T[j] entonces Max := T[j]; Min := T[i];
  sino Max := T[i]; Min := T[j];
  fin si;
  si i+1 • j-1 entonces
    MAX_MIN (T[i+1..j-1], Aux_Max, Aux_Min);
    si Max < Aux_Max entonces Max := Aux_Max; fin si;
    si Aux_Min < Min entonces Min := Aux_Min; fin si;
  fin si;
fin proc;

```

Res:

Las comparaciones entre componentes del vector son $T[i] \leq T[j]$, $\text{Max} < \text{Aux_Max}$ y $\text{Aux_Min} < \text{Min}$.

El número de comparaciones que realiza el procedimiento MAX_MIN sobre un vector de tamaño n es

$$t(n) = \begin{cases} 1 & n \leq 2 \\ t(n-2) + 3 & n > 2 \end{cases}$$

que resolviendo resulta $t(n) = 3n/2 - 2$.

16. Teniendo una secuencia de n claves distintas a ordenar, considérese la siguiente variación del algoritmo de Ordenación por Inserción:

Para $2 \leq i \leq n$: insertar $S(i)$ entre $S(1) \leq S(2) \leq \dots \leq S(i-1)$ empleando la búsqueda dicotómica para determinar la posición correcta donde debe ser insertado $S(i)$.

Es decir, el algoritmo de inserción para *determinar la posición* donde debe insertarse $S(i)$ hará uso del siguiente algoritmo:

```

Alg BUSQ_DICOT_POSICION(Sec, Valor) devuelve Posición
-- Valor  $\notin$  Hacer_Cjto(Sec)
si Sec.Length=1 ent
    si Sec(Sec.First) < Valor ent devolver Sec.First+1
    else devolver Sec.First
sino si Sec((Sec.length)/2) < Valor ent
    devolver BUSQ_DICOT_POSICION(Sec((Sec.length)/2)+1,
    ..Sec.Last),
    Valor)
sino devolver BUSQ_DICOT_POSICION(Sec(Sec.First..
    ((Sec.length)/2)-1)), Valor)
    
```

Ejemplos:

S								
2	5	8	18	20	30	Valor	...	S(n)
1	2	3	4	5	6	i	...	n

BUSQ_DICOT_POSICION(S(1..6), 1) → Posición: 1

BUSQ_DICOT_POSICION(S(1..6), 3) → Posición: 2

BUSQ_DICOT_POSICION(S(1..6), 9) → Posición: 4

BUSQ_DICOT_POSICION(S(1..6), 37) → Posición: 7

Analizar lo siguiente de esta variante del algoritmo de Ordenación por Inserción.

a) Comparaciones entre claves:

¿Cuál es un caso peor?

¿Cuál es el orden del número de comparaciones entre claves que se harán en ese peor caso?

b) Movimientos de claves:

¿Cuál es el caso peor?

¿Cuántos movimientos de claves en total se harán en el peor caso?

c) ¿Cuál es el orden del tiempo de ejecución del algoritmo en el peor caso?

✎:

(a) Cualquier caso es ejemplo de caso peor ya que siempre, salvo cuando la secuencia tiene tamaño unidad, se realiza llamada recursiva.

Sea $t(i)$ el número de comparaciones de $BUSQ_DICOT_POSICION(S(1..i), Valor)$. Se satisface la ecuación siguiente:

$$T(i) = \begin{cases} 1 & i = 1 \\ 1 + T\left(\left\lfloor \frac{i}{2} \right\rfloor\right) & i > 1 \end{cases}$$

Suponiendo que $i=2^k$ y resolviendo: $t(i) = t(i/2) + 1 = \dots = 1 + \lg i$

El número de comparaciones de claves del algoritmo de Ordenación será:

$$\sum_{i=1}^{n-1} (1 + \lg i) \in O(n \lg n)$$

(b) Un caso peor es cuando el vector de entrada viene ordenado decrecientemente, porque obliga a mover el máximo número de claves en cada inserción.

En la inserción de la componente i -ésima movemos (n° de asignaciones de claves) $i+1$ claves: $i-1$ “traslaciones” más dos para la inserción.

Tenemos que realizar una inserción por cada valor de y desde 2 hasta n , por consiguiente el número de movimientos será

$$\sum_{i=2}^n (i+1) \in O(n^2)$$

(c) El orden de esta variante del algoritmo de ordenación por inserción es $O(n^2)$, debido a que el número de movimientos es de ese orden y esos movimientos son operaciones elementales críticas (es decir, adecuadas para el análisis del algoritmo).

17. Hemos diseñado la siguiente versión del algoritmo Mergesort con intención de evitar el uso de espacio extra de orden lineal.

Analice su eficiencia temporal.

```

procedure MERGESORT_SOBRE_VECTOR (V: in out VECTOR,
                                     I, J: in INDICE) is
    K: INDICE :=  $\lfloor I+J/2 \rfloor$ ;
begin
    if ES_PEQUEÑO(I, J) then SIMPLE_SORT(V, I, J);
    else
        MERGESORT_SOBRE_VECTOR(V, I, K);
        MERGESORT_SOBRE_VECTOR(V, K+1, J);
        MERGE_SOBRE_VECTOR(V, I, K, J);
    end if;
end MERGESORT_SOBRE_VECTOR;

```

siendo $MERGE_SOBRE_VECTOR(V, I, X, J)$ un procedimiento que ordena el segmento $V(I..J)$ del vector V cuando recibe dos segmentos ordenados que ocupan posiciones consecutivas $V(I..X)$ y $V(X+1..J)$

```

procedure MERGE_SOBRE_VECTOR (V: in out VECTOR, I, X, J: in INDICE) is

```

```

Izq: INDICE:=I; Der: INDICE:=X+1;
Aux: ELEM_VECTOR;
begin
  while Izq \= Der loop
    if V(Izq)>V(Der) then
      Aux:= V(Izq);
      V(Izq):=V(Der);
      Insertar Aux en V(Der..J) dejándolo ordenado;
    end if;
    Izq:= Izq+1;
  end loop;
end MERGESORT_SOBRE_VECTOR;

```

✎:

El procedimiento Merge_Sobre_Vector(V, i, x, j) es de $O(n^2)$ siendo $n=j-i+1$ el tamaño del vector $V[i..j]$. Obsérvese que, en el caso peor, habrá que insertar $n/2$ elementos en $V[x+1..j]$, realizando $n/2$ movimientos de componentes del vector cuando Aux deba colocarse en $V(j)$.

Para analizar la eficiencia de MergeSort_Sobre_Vector($V, 1, n$) incluimos en la recurrencia sólo los sumandos fundamentales.

$$T(n) = \begin{cases} a & n = 1 \\ 2T(n/2) + T_{\text{merge}}(n) & n > 1 \end{cases} = \begin{cases} a & n = 1 \\ 2T(n/2) + n^2 & n > 1 \end{cases}$$

Luego, con cualquiera de estas dos formas necesitaría de un tiempo lineal en el número de elementos que del vector., teniendo que repetir el proceso k veces. Así pues, el orden del Merge_Sobre_Vector dado vector con n elementos es $O(n^2)$:

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n^2 = 2\left(2T\left(\frac{n}{2^2}\right) + \left(\frac{n}{2}\right)^2\right) + n^2 = 2^2T\left(\frac{n}{2^2}\right) + n^2\left(1 + \frac{1}{2}\right) \\
&= 2^3T\left(\frac{n}{2^3}\right) + n^2\left(1 + \frac{1}{2} + \frac{1}{2^2}\right) \quad \text{tras 3 pasos} \\
&= 2^i T\left(\frac{n}{2^i}\right) + n^2 \sum_{j=0}^{i-1} \frac{1}{2^j} \quad \text{tras } i \text{ pasos}
\end{aligned}$$

Haciendo $n=2^i$ llegamos al caso básico, y puesto que el otro sumando contiene a una progresión geométrica de razón $1/2$:

$$= a n + n^2 \left(2 - \frac{2}{n}\right) = 2n^2 + (a-2)n \in O(n^2)$$

De todo esto se concluye que el intento de ahorrar algo de espacio de memoria frente al empleado por el MergeSort tradicional repercute en el orden temporal de la ejecución empeorándolo.

18. Suponga que vamos a utilizar el siguiente algoritmo para encontrar las k mayores claves de una lista de n claves.

```

function LAS_K_MAYORES_CLAVES (Vec: in VECTOR; K: in INTEGER)
  return VECTOR is

```

```

M: VECTOR;
Claves: VECTOR(1..K);
begin
  Crear montículo M con las n claves;
  for J in 1 .. K loop
    Claves(J) := M(1);
    M(1) := M(n-J+1); -- trasladar el último a la raíz
    HUNDIR_RAIZ(M(1..n-J));
  end loop;
  return Claves;
end LAS_K_MAYORES_CLAVES;

```

¿ De qué orden (en función de n) puede ser k para que este algoritmo sea de $O(n)$?

✍:

Observado que el algoritmo consta de dos partes:

- construir montículo $\in O(n)$
- obtener los k mayores de un montículo que inicialmente tiene n elementos:

$$\sum_{j=1}^k \lg(n-j) \leq k \lg n, \text{ es decir, } O(k \log n)$$

y con objeto de que el algoritmo sea $O(n)$, será necesario acotar superiormente el tiempo del bucle de extracción de los k mayores elementos por dicho valor: $O(k \log n) \subseteq O(n)$. Basta aplicar directamente la definición de O

$k \log n \in O(n)$

$$\begin{aligned} &\Leftrightarrow \exists c \exists n_0 ((c \in \mathbb{R}^+) \wedge (n_0 \in \mathbb{N}) \wedge \forall n (n \geq n_0 \Rightarrow k \log n \leq c n)) \\ &\Leftrightarrow \exists c \exists n_0 ((c \in \mathbb{R}^+) \wedge (n_0 \in \mathbb{N}) \wedge \forall n (n \geq n_0 \Rightarrow k \leq c n / \log n)) \\ &\Leftrightarrow k \in O(n / \log n) \end{aligned}$$

19. El siguiente algoritmo puede emplearse para buscar un nombre en un listín telefónico (Obviamente el listado está ordenado alfabéticamente):

```

function BUSCO (Lis: in Listado; Apellido: in String;
                Salto: in Positive) return Integer is
-- Se supone que Apellido está en Lis.
-- Devuelve el índice donde se encuentra Apellido en Lis.
begin
  Indice:= Lis'First+Salto-1;
  while Lis'Last>Indice and then Apellido>Lis(Indice) loop
    Indice:= Indice+Salto;
  end loop;
  if Lis'Last>Indice
  then Hasta:= Indice;
  else Hasta:= Lis'Last;
  end if;
  return (BUSQUEDA_LINEAL(Lis(Indice-Salto+1..Hasta),Apellido));
end BUSCO;

```

- a) ¿Cuántas veces se ejecuta la comparación de Apellido con una componente de Lis en el peor caso? (Sirve la misma observación del ejercicio anterior.)
Sabemos que BUSQUEDA_LINEAL(Lis(X..Y),Apellido) produce $Y-X$ comparaciones.

- b) ¿Cambiaría el orden del algoritmo si en vez de `BUSQUEDA_LINEAL` invocásemos a `BUSQUEDA_DICOTOMICA(Lis(X..Y),Apellido)` que produce $1 + \lfloor \log(Y-X) \rfloor$ comparaciones?

✎:

Dependiendo de que n sea o no múltiplo de Salto, el mayor número de comparaciones será:

$$(a1) \left\lfloor \frac{n}{\text{salto}} \right\rfloor + \text{salto} - 1 \quad \text{si } n \bmod \text{salto} \neq 0$$

$$(a2) \frac{n}{\text{salto}} - 1 + \text{salto} - 1 \quad \text{si } n \bmod \text{salto} = 0$$

En el caso (a1) el peor caso es cuando `Apellido` está o bien en la posición $\left(\left\lfloor \frac{n}{\text{salto}} \right\rfloor * \text{salto}\right) - 1$ o bien en la última posición (o penúltima, porque requiere el mismo nº de

comparaciones) si ésta es $n = \left\lfloor \frac{n}{\text{salto}} \right\rfloor * \text{salto} + \text{salto} - 1$, necesitando para cualquiera de los casos el número de comparaciones anteriormente mencionadas.

Por lo tanto, el nº mayor de comparaciones es el máximo de (a1) y (a2).

Por otro lado, mencionar que el orden del algoritmo puede cambiar si invocamos `BUSQUEDA_DICOTOMICA`, dependiendo de la magnitud de `salto`:

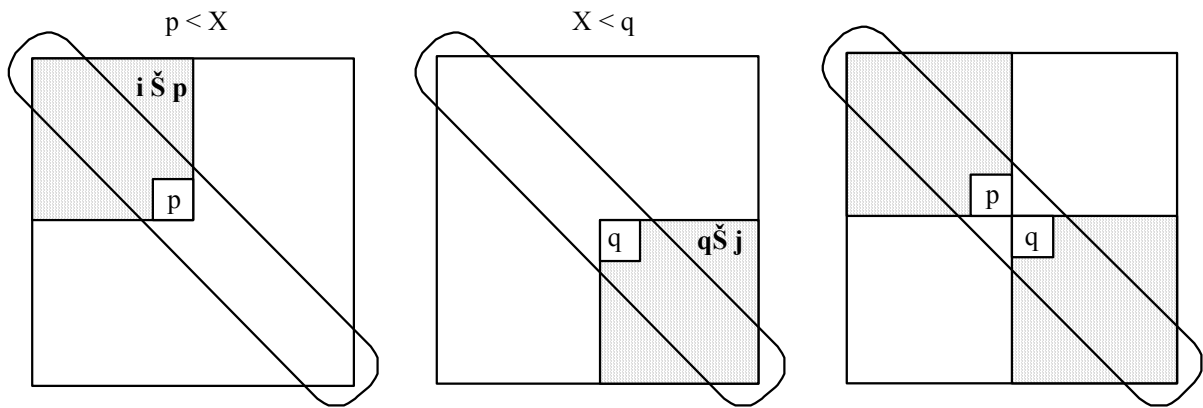
$$(b1) \left\lfloor \frac{n}{\text{salto}} \right\rfloor + 1 + \lg \text{salto} \quad \text{si } n \bmod \text{salto} \neq 0$$

$$(b2) \frac{n}{\text{salto}} - 1 + 1 + \lg \text{salto} \quad \text{si } n \bmod \text{salto} = 0$$

Por ejemplo, si $\text{salto} \in \Theta(n)$ entonces el algoritmo pasaría a ser $O(\lg n)$

20. Para determinar si un ítem X se halla presente en una matriz cuadrada de ítems $T(1..n, 1..n)$ cuyas filas y columnas están ordenadas crecientemente (de izquierda a derecha y de arriba a abajo respectivamente), se utiliza el siguiente algoritmo: (Sea benevolente con el enunciado, admitiendo que siempre tenga sentido eso de “la diagonal”)

- Con búsqueda dicotómica se busca X en la “diagonal”.
- Si encontramos X , terminamos.
- Si no, (encontramos p y q tales que $p < X < q$) descartamos los dos trozos de la matriz donde no puede estar X e invocamos recursivamente el mismo procedimiento sobre los dos trozos restantes.



¿De qué **orden** es este algoritmo en el **caso peor**?

✍:

El caso peor sucede cuando X no se halla en la diagonal, sucediendo $p < X < q$, y concretamente cuando esto sucede en la mitad de la diagonal, ya que ello conlleva a que tan sólo se desprecien dos trozos cada uno con un cuarto de elementos de la matriz.

Ahora bien, hay dos posibles alternativas a la hora de considerar el tamaño de la entrada:

- a) Sea n el número de filas (o columnas) puesto que la matriz es cuadrada. La búsqueda dicotómica requerirá tiempo $\lg n$, mientras que las dos llamadas recursivas se harán con matrices de tamaño $n/2$.

$$T(n) = \begin{cases} a & n = 1 \\ \lg n + 2T\left(\frac{n}{2}\right) & n > 1 \end{cases}$$

$$T(n) = \lg n + 2T\left(\frac{n}{2}\right) \text{ y haciendo } n=2^k$$

$$T(2^k) = \lg 2^k + 2T(2^{k-1}) \quad T_k - 2T_{k-1} = k; \quad (x-2)(x-1)^2 = 0;$$

$$T_k = c_1 2^k + c_2 1^k + c_3 1^k k$$

que tras deshacer el cambio, obtenemos: $T(n) = c_1 n + c_2 + c_3 \lg n \in O(n)$

- b) Sea n el número de elementos de la matriz. En este caso, la búsqueda dicotómica requerirá tiempo $\lg \sqrt{n}$, mientras que las dos llamadas recursivas se harán con matrices de tamaño $n/4$.

21. Un array T contiene n números naturales distintos. Queremos conocer los menores m números de T . Sabemos que m es mucho menor que n . ¿Cómo lo harías para que resultara más eficiente?:

- Ordenar T y tomar los m primeros.
- Invocar $\text{SELECCIONAR}(T, k)$ para $k=1, 2, \dots, m$.
- Usar algún otro método.

Justifica tu respuesta contrastando en tu estudio los análisis de caso peor y caso medio conocidos.

¿? :

(a) 1ª ordenar (HeapSort) ($\Theta(n \lg n)$) + 2ª seleccionar los m menores ($\Theta(m)$):
 $\Theta(n \lg n + m)$

(b) SELECCIONAR_K(T(1..N), K) lo ejecutamos m veces con K=1, K=2,... y K= m:

- en el caso peor: $T_p \in \Theta(m n^2)$

- en el caso medio: $T_m \in \Theta(m n)$

(c) Hay distintas posibilidades: la primera es la más sencilla y la peor. Las dos siguientes son mejores:

(c1) 1º: escoger el menor entre n elementos $n-1$ comparaciones

2º: escoger el menor entre los n-1 restantes $n-2$ comparaciones

...

mº: escoger el menor entre los n-(m-1) restantes $\underline{n-m}$ comparaciones
 $\in \Theta(n m)$

(c2) Se puede mejorar (c1) empleando montículos empleando la misma idea que en el HeapSort:

```

Crear_Monticulo_de_Mínimos(T(1..N))           $\in \Theta(n)$ 
for I in 1 .. m loop
    Intercambiar_Raíz_y_Último( T(1..N-I+1) );
    Hundir_Raíz (T(1..N-I));                     $\in \Theta(m \lg n)$ 
end loop;
Devolver_los_m_elementos(T(N-M+1..N))           $\in \Theta(m)$ 
-----
 $\in \Theta(n + m \lg n + m) = \Theta(n + m \lg n)$ 

```

(c3) Se puede mejorar (c1) empleando SELECCIONAR_K:

Una única llamada con SELECCIONAR_K(T(1..N), m) nos devuelve en las primeras (m-1) posiciones de T los m-1 elementos menores o iguales del vector y en la posición m-ésima el m-ésimo más pequeño. Basta con luego devolver los m primeros elementos:

- en el caso peor: $T_p \in \Theta(n^2 + m)$

- en el caso medio: $T_m \in \Theta(n + m)$

COTEJO: sabemos $m \ll n$

	<u>peor</u>	<u>medio</u>
(a)	$\Theta(n \lg n + m)$	$\Theta(n \lg n + m)$
(b)	$\Theta(m n^2)$	$\Theta(m n)$
(c)		
(c1)	$\Theta(n m)$	$\Theta(n m)$
(c2)	$\Theta(n + m \lg n)$	$\Theta(n + m \lg n)$
(c3)	$\Theta(n^2 + m)$	$\Theta(n + m)$

En el caso medio la mejor solución es (c3). En el peor caso la mejor solución es (c2).

OBS: Si no se os ha ocurrido (c3), la mejor solución tanto en el caso peor como en el medio es (c2).

Si tampoco se os ha ocurrido (c2), entonces la mejor solución dependerá del valor de m ya que, por ejemplo, en los casos medios:

$m = \sqrt{n}$	$O(n \lg n + m) \subseteq O(nm) = O(n\sqrt{n})$ <hr/> (a) (b) y (c1)
$m = \lg n$	$O(n \lg n + m) = O(n \lg n) = O(nm)$ <hr/> (a) (b) y (c1)

22. Una persona piensa un número entero positivo W . Escribe un algoritmo para que otra persona lo adivine realizándole preguntas con la relaciones de orden: $<$, $>$, $=$. El número de preguntas debe ser $o(W)$.

ℳ :

Ya que se pide hallar una solución de orden mejor que lineal en W , esto indica que la búsqueda del número W no se debe realizar por incrementos constantes (1,2 o en general c constante). Se intenta entonces buscarlo con saltos potencias de 2, hasta que hallemos W o el valor del salto exceda al mismo. Si sucede este último caso, buscamos dicotómicamente en el intervalo de número existentes entre el penúltimo y último salto. El realizar saltos potencias de 2 permite saber con exactitud el número de elementos (en este caso números) que hay en el intervalo que se realiza la búsqueda dicotómica, lo cual es imprescindible para determinar el orden de la misma.

Algoritmo:

```

procedure Adivina ( $W$ ) is
  begin
     $j := 0$ ;
    while EsMenor? ( $2^j, W$ ) loop
       $j := j + 1$ ;
    end loop;
    if NoEs? ( $2^j, W$ ) then BusqDicotomica ( $2^{j-1} + 1, 2^j - 1, W$ ); end if;
    Escribir("El número que has pensado es: ",  $W$ );
  end;

```

Análisis del coste:

El algoritmo propuesto realiza el siguiente número de preguntas del tipo ($<$, $=$, $>$):

1. Si $W = 2^j$ entonces j preguntas o, lo que es lo mismo, $(\lg W)$.
2. Si $2^{j-1} < W < 2^j$ entonces j preguntas más las realizadas por la búsqueda dicotómica en $(2^{j-1}, 2^j)$. Como en dicho intervalo hay 2^{j-1} elementos, se realizan $O(\log 2^{j-1})$ preguntas en la búsqueda dicotómica, o lo que es lo mismo, $O(\lg W)$ (por $2^{j-1} < W < 2^j$)

Debido a que tanto en (1) como (2) el número de preguntas realizadas es $O(\lg W)$ el orden de la solución propuesta es $O(\lg W) \subset o(W)$.

23.

(a) Analiza el algoritmo siguiente, que transforma el array $T(1..n)$ en un montículo.

```

proc crear_montículo (T(1..n))
para i:=  $\lfloor n/2 \rfloor$  disminuyendo hasta 1 hacer hundir(T(1..n), i)

```

(b) Con el análisis del apartado (a) habrás descubierto que crear_montículo(T(1..n)) es de $O(n)$. Entonces, ¿por qué es incorrecto el razonamiento siguiente para analizar el algoritmo HeapSort?:

```

proc HeapSort (T(1..n))
  crear_montículo (T(1..n))
  para i:= n disminuyendo hasta 2 hacer
    intercambiar T(1) y T(i)
    hundir(T(1..i-1), 1)

```

Yo veo que crear_montículo (T(1..n)) realiza $\lfloor n/2 \rfloor$ veces el procedimiento hundir, y que el bucle de $n-1$ iteraciones de HeapSort(T(1..n)) puede verse como 2 veces $\lfloor n/2 \rfloor$ realizaciones del procedimiento hundir (más el intercambio que es de $O(1)$). Por tanto HeapSort(T(1..n)) realiza consecutivamente tres fragmentos de programa de $O(n)$ cada uno y concluyo que HeapSort(T(1..n)) es de $O(n)$.

✎ :

(a) Sea $p = \lfloor \lg n \rfloor$ la profundidad de un montículo con n elementos. En el caso peor el procedimiento hundir, hunde un nodo hasta convertirlo en hoja:

- Por cada nivel que baja el elemento se realizan dos comparaciones (una entre los hijos y otra entre el hijo mayor y el elemento a hundir en cuestión).
- Un nodo de nivel j cuando tras hundirlo se ha convertido en hoja, ha requerido pasar por $(p-j)$ niveles. O lo que es lo mismo, era la raíz de un montículo de profundidad $(p-j)$ y tras hundir se ha convertido en hoja de ese mismo montículo.
- En cada nivel j , el número de nodos internos (n° de nodos a hundir) es 2^j .

Agrupando los resultados anteriores tenemos que el número de comparaciones realizadas por el procedimiento hundir en el caso peor es:

$$\begin{aligned}
 T(n) &= \sum_{j=0}^{p-1} 2^{p-j} \cdot 2^j = 2^p \sum_{j=0}^{p-1} 2^{-j} = 2^p \sum_{j=1}^p \frac{1}{2^j} = 2^p (2^0 - 2^{-p}) = 2^p (1 - 2^{-p}) = 2^p - 2^0 = 2^p - 1 \\
 &= 2^{\lfloor \lg n \rfloor + 1} - 1 \leq 2^{\lfloor \lg n \rfloor + 1} = 2 \cdot 2^{\lfloor \lg n \rfloor} = 2 \cdot n
 \end{aligned}$$

$$T(n) \in O(n)$$

(b) La incorrección del razonamiento está en “tres fragmentos de programa de $O(n)$ cada uno”, ya que no es cierto que realizar $\lfloor n/2 \rfloor$ veces hundir sea siempre de $O(n)$, depende de los parámetros de hundir. Obsérvese que hundir (T(1..n),i) es de $O(\lg n - \lg i)$ y, por ejemplo, hundir $n/2$ veces una raíz -hundir (T(1..n),1)- resulta $O(n \lg n)$. Concretamente, en el bucle del HeapSort se hunde la raíz en montículos de tamaño $(n-1)$, $(n-2)$,..., y finalmente

de tamaño 2. Ello implica que el trabajo realizado por hundir es, en el peor de los casos: $\lg(n-1) + \lg(n-2) + \dots + \lg 2 \in O(n \lg n)$

24. Considérese el algoritmo de ordenación siguiente.

```

procedure TerciosSort (B: in out Vector; I,J: in Integer) is
    K: Integer :=  $\left\lfloor \frac{J-I+1}{3} \right\rfloor$ ;
begin
    if B(I) > B(J) then Intercambiar(B(I),B(J)); end if;
    if J-I+1 <= 2 then return; end if;
    TerciosSort (B, I, J-K);
    TerciosSort (B, I+K, J);
    TerciosSort (B, I, J-K);
end TerciosSort;

```

(a) Analiza su tiempo de ejecución.

(b) Compáralo con la eficiencia de *InsertionSort* y *HeapSort*.

Observaciones:

- Puedes considerar valores de n de la forma $(3/2)^k$.
- $\log_b a = \frac{\log_c a}{\log_c b}$
- $\log_3 2 = 0.63092$

✎ :

El tiempo de ejecución de TerciosSort se ha recogido en la siguiente ecuación de recurrencia.

$$T(n) = \begin{cases} a & \text{si } n \leq 2 \\ 3 T\left(\frac{2}{3}n\right) + b & \text{si } n > 2 \end{cases}$$

Una vez desarrollada la ecuación de recurrencia se halla el patrón general:

$$T(n) = 3^i T\left(\left(\frac{2}{3}\right)^i n\right) + b \sum_{j=0}^{i-1} 3^j$$

Suponiendo que $n = (3/2)^k$ y que el caso básico se obtiene cuando $i=k$, tenemos que:

$$T(n) = 3^k a + b \frac{3^k - 1}{2} = \left(a + \frac{b}{2}\right) 3^k - \frac{b}{2}$$

Como el orden de ejecución se tiene que dar en función de n , sustituimos k por su valor equivalente en función de n :

$$n = (3/2)^k \Rightarrow \log_3 n = k \log_3 \frac{3}{2}; \quad k = \frac{\log_3 n}{\log_3 \frac{3}{2}} = \frac{\log_3 n}{1 - \log_3 2} = \frac{1}{1 - \log_3 2} \log_3 n = c \log_3 n$$

Obsérvese el valor de $c = \frac{1}{1 - \log_3 2} = \frac{1}{0,36908} = 2,7094$; así pues, $2 < c < 3$.

Entonces: $3^k = 3^{c \log_3 n} = n^c$ y por tanto $T(n) \in O(n^c)$ con $2 < c < 3$.

Por ello, concluimos que TerciosSort es asintóticamente peor que $O(n \log n)$ ya que $O(n \lg n) \subset O(n^2)$.

25. Completa el siguiente algoritmo para que calcule el k-ésimo mayor elemento de los n números enteros que el procedimiento **get-number** (X: **out** integer) irá generando. Nos interesa minimizar el espacio extra necesario, por tanto busca soluciones que no requieran siempre espacio extra de $\Omega(n)$.

```

for i := 1 to n loop
  get-number (X)
  ....
  ....
end loop
return K_ESIMO

```

✎:

Aprovecharemos que el k-ésimo mayor elemento de n números es también el (n-k+1)-ésimo menor de los mismos. Para minimizar el espacio extra utilizaremos un vector de tamaño k si $k \leq n/2$ o bien de tamaño (n-k+1) si $K > n/2$. Para realizarlo de modo eficiente, ese vector será un montículo:

- Si $k \leq n/2$: Registraremos los k mayores elementos recibidos hasta el momento en un montículo de mínimos (valor de nodo padre menor o igual que el valor de nodo hijo).
- Si $k > n/2$: Registraremos los (n-k+1) menores elementos recibidos hasta el momento en un montículo de máximos.

Obsérvese que, en el caso $k \leq n/2$, si el elemento recibido x es mayor que el valor de la raíz del montículo, nos interesa considerarlo. Por el contrario, en el caso $k > n/2$, nos interesa cuando x es menor que el valor de la raíz.

Buscando simplicidad en la expresión del algoritmo, presentamos la versión siguiente:

```

begin
  if k ≤ n/2
  then
    declare  GRANDES: monticulo(1..k) := (others => System.MIN_INT);
             G: boolean := true;
    end;
  else
    declare  PEQUEÑOS: monticulo(1..n-k+1) := (others => System.MAX_INT);
             G: boolean := false;
    end;
  end if;

  for I in 1..n loop
    Get_Number (X);
    case
      G:    if x > Raiz (GRANDES) then
             GRANDES (1) := X; Hundir (GRANDES, 1);
            end if;
      not G: if x < Raiz (PEQUEÑOS) then
             PEQUEÑOS (1) := X; Hundir (PEQUEÑOS, 1);
            end if;
    end case;
  end loop;

```

```

    if G then K_ESIMO:= Raiz (GRANDES);
    else K_ESIMO:= Raiz (PEQUEÑOS);
    end if;
    return K_ESIMO;
end;

```

Las operaciones de hundir raíz son las que determina la eficiencia de este algoritmo, el resto son operaciones elementales o de inicialización que no van a elevar el orden. En el peor caso hay que realizar n operaciones hundir en un montículo con $\min(k, n-k+1)$ elementos. Por lo tanto, $O(n \lg(k, n-k+1))$.

26. Un mínimo local de un array $A(a-1...b+1)$ es un elemento $A(k)$ que satisface $A(k-1) \geq A(k) \leq A(k+1)$. Suponemos que $a \leq b$ y $A(a-1) \geq A(a)$ y $A(b) \leq A(b+1)$; estas condiciones garantizan la existencia de algún mínimo local. Diseña un algoritmo que encuentre algún mínimo local de $A(a-1...b+1)$ y que sea substancialmente más rápido que el evidente de $O(n)$ en el peor caso. ¿De qué orden es tu algoritmo?

$\left. \begin{array}{l} a \leq b \\ A(a-1) \geq A(a) \\ A(b) \leq A(b+1) \end{array} \right\}$ son las condiciones que garantizan la existencia de un mínimo local.

Solución con la técnica "divide y vencerás":

- Si hay 3 elementos: $A(a-1) \geq A(a) \leq A(a+1)$. El mínimo local está en la posición **a**
- Si hay más de 3 elementos:
Puesto que se pide un algoritmo mejor que lineal, hay que evitar comparar todos los elementos.
Si dividimos en dos partes iguales:

- si $A((n/2)-1) \geq A(n/2) \leq A((n/2)+1)$ entonces el mínimo local está en la posición **$n/2$**
- si no -- alguna de las inecuaciones de la condición de arriba no es cierta
 - si $A((n/2)-1) < A(n/2)$ entonces nos encontramos en las condiciones que garantizan la existencia de un mínimo local entre $(a-1)$ y $(n/2)$
 - si no entonces nos encontramos en las condiciones que garantizan la existencia de un mínimo local entre $((n/2)-1)$ y $(b+1)$

```

function MIN_LOCAL (A: in Vector) return Indice is
  medio: Integer;
begin
  if A'Length=3 then return(A'First+1);
  else medio:= A'Length/2;
    if A(medio-1) > A(medio) & A(medio) > A(medio+1)
    then return(medio);
    elsif A(medio-1) < A(medio)
    then return MIN_LOCAL(A(A'First..medio));
    else return MIN_LOCAL(A(medio-1..A'Last));
    end if;
  end if;
end MIN_LOCAL;

```

En cuanto al orden del algoritmo fácilmente se puede observar que en el caso general se realiza una única llamada recursiva de tamaño $(n/2)$. Por ello, podemos decir que a MIN_LOCAL le corresponde la misma ecuación de recurrencia que al algoritmo de "búsqueda dicotómica" ($T(n) = a + T(n/2)$) concluyendo por ello que $T(n) \in O(\lg n)$.

27.

- a) Considera el algoritmo recursivo de búsqueda dicotómica en un vector ordenado:
(Si X no estuviera en el vector devuelve el índice 0):

```

proc BUSQ_DICO (T[i..j], X, Ind)
  si i < j entonces
    k := (i+j) div 2;
    si T[k] = X entonces devolver Ind := k;
    sino
      si T[k] < X entonces BUSQ_DICO(T[k+1..j], X, Ind);
      sino BUSQ_DICO(T[i..k-1], X, Ind);
    fin si;
  fin si;
  sino si i=j y también T[i] = X
    entonces devolver Ind := i;
    sino devolver Ind := 0;
  fin si;
fin proc;

```

Considérense dos implementaciones distintas del mismo algoritmo, en las cuales en una el paso de parámetros se realiza por referencia y en la otra por copia (o valor) ¿Habría alguna diferencia en la eficacia temporal entre esas dos implementaciones?

- b) En el Mergesort, ¿habría alguna diferencia en la eficacia temporal entre las dos posibilidades de paso de parámetro antes mencionadas?

⚡:

(a.1) Paso del vector por referencia: Tiempo necesario para el paso $O(1)$. Función temporal de BUSQ_DICO:

$$t(n) = \begin{cases} a & n \leq 1 \\ t\left(\frac{n}{2}\right) + b & n > 1 \end{cases}$$

Es conocido que $t(n) \in O(\lg n)$.

(a.2) Paso del vector por valor: Tiempo necesario para el paso $O(n)$ si el vector es de tamaño n. Función temporal en este caso:

$$t(n) = \begin{cases} a & n \leq 1 \\ t\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

Que resolviendo resulta $t(n) \in O(n)$.

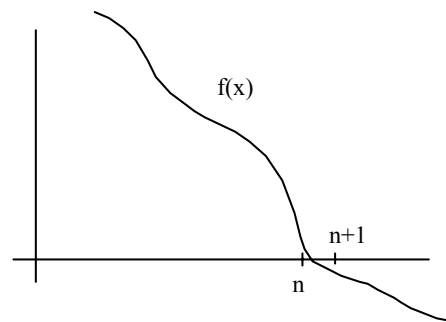
(b.1) Para el algoritmo Mergesort, el paso del vector por referencia determina una función temporal como la siguiente:

$$t(n) = \begin{cases} a & n \leq 1 \\ 2t\left(\frac{n}{2}\right) + n & n > 1 \end{cases}$$

(b.2) El paso del vector por valor, necesitaría un tiempo $O(n)$ para vectores de tamaño n , pero la incidencia de esto en el orden de la función temporal es nula ya que sólo afecta al coeficiente que pueda multiplicar al sumando n de $2t(n/2) + n$.

28. Sea $F(x)$ una función monótona decreciente y sea N el mayor entero que cumple $F(N) \geq 0$. Asumiendo que N existe, un algoritmo para determinar dicho N es el siguiente:

```
I:=0;
while F(I) > 0 loop
  I:= I + 1;
end loop
N = I-1;
```



No obstante, esta solución tiene complejidad $O(N)$. Escribase un algoritmo cuyo comportamiento asintótico sea mejor en función de N .

Ans:

Ya que se pide hallar una solución de orden mejor que lineal en N , esto indica que la búsqueda del número N no se debe realizar por incrementos constantes (1,2 o en general c constante). Se intenta entonces buscarlo con saltos potencias de 2, hasta que hallemos N o lo superemos por primera vez. Si sucede este último caso, buscamos dicotómicamente en el intervalo de números existentes entre el penúltimo y último salto. El realizar saltos potencias de 2 permite saber con exactitud el número de elementos (en este caso, números) que hay en el intervalo en el que se realiza la búsqueda dicotómica, lo cual es imprescindible para determinar el orden de la misma..

Algoritmo:

```
procedure Adivina (N) is
begin
  j:=0;
  while F(2j) > 0 loop
    j:=j+1;
  end loop;
  if F(2j) = 0 then N:= 2j;
  else BusqDicotomica(2j-1+1, 2j-1, N); end if;
  Escribir("F(x) no negativa hasta x= " & N);
end;
```

Análisis del coste:

El algoritmo propuesto tiene el siguiente coste:

1. Si $N = 2^j$ entonces j iteraciones del while (cada una de $O(1)$) o, lo que es lo mismo, $O(\lg N)$.
2. Si $2^{j-1} < N < 2^j$ entonces j iteraciones del while (cada una de $O(1)$) + las operaciones realizadas por la búsqueda dicotómica en $(2^{j-1}, 2^j)$. Como en dicho intervalo hay 2^{j-1} elementos, se realizan $O(\log 2^{j-1})$ preguntas en la búsqueda dicotómica, o lo que es lo mismo, $O(\lg N)$ (porque $2^{j-1} < N < 2^j$)

Debido a que tanto en (1) como (2) el número de operaciones realizadas es $O(\lg N)$, el orden de la solución propuesta es $O(\lg N) \subset o(N)$.

29. Sea una red internacional de n ordenadores. Cada ordenador X puede comunicarse con cada uno de sus vecinos Y al precio de 1 doblón por comunicación. A través de las conexiones de Y y subsiguientes, X puede comunicarse con el resto de ordenadores de la red pagando a cada ordenador que utilice para la conexión el doblón correspondiente.

Describe un algoritmo que **calcule la tabla PRECIO(1..n)** tal que $\text{PRECIO}(i)$ es el número mínimo de doblones que le cuesta al ordenador 1 establecer una conexión con el ordenador i

✎:

Basta un recorrido en anchura del grafo que representa a la red, partiendo del nodo 1.

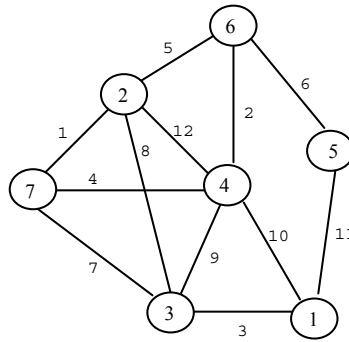
```

proc MARCA_ANCHO (v: Vértice)
  C:= cola_vacia
  marca(v) := true
  PRECIO(v) := 0
  añadir(C, v)
  mientras no_vacia(C) hacer
    u:= retirar_primer(C)
    para cada vértice w adyacente a u hacer
      si not marca(w) entonces
        PRECIO(w) := PRECIO(u) + 1
        marca(w) := true
        añadir(C, w)
    fin para

```

Es evidente que el tiempo de este algoritmo es el mismo que el de un recorrido en anchura: $\Theta(n+a)$, siendo a el número de aristas (comunicaciones) del grafo.

30. Dado el siguiente grafo:



- (a) Escribe, en orden de selección, las aristas seleccionadas por el algoritmo de *Prim* sobre ese grafo.
- (b) Lo mismo que el apartado (a) pero con el algoritmo de *Kruskal*.

✎:

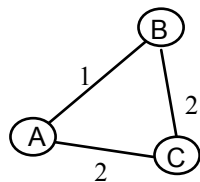
- (a) Supongamos que el vértice de partida es el 1: (1,3),(3,7),(7,2),(4,7),(4,6),(5,6)
- (b) (2,7),(4,6),(1,3),(4,7),(5,6),(3,7)

31. Supóngase que un grafo tiene exactamente 2 aristas que tienen el mismo peso.

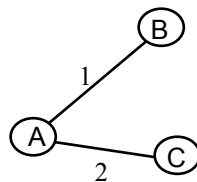
- (a) ¿Construye el algoritmo de *Prim* el mismo árbol de expansión mínimo, independientemente de cuál de esas aristas sea seleccionada antes?
- (b) Lo mismo que el apartado (a) pero con el algoritmo de *Kruskal*.

✎:

No tiene porque construir el mismo árbol de expansión en ninguno de los dos casos. Véase el siguiente ejemplo. Tanto *Kruskal* como *Prim* podrían dar la solución Solución 1 ó Solución 2.



Solución 1



Solución 2.

32. Estúdiese el siguiente algoritmo para calcular las componentes conexas de un grafo no dirigido $G=(\text{VERTICES}, \text{ARISTAS})$ usando la estructura de partición:


```

proc COMPONENTES_CONEXAS (G)
  -- Inicialización de la partición
  para cada vértice  $v \in \text{VERTICES}$  hacer Crear_Parte(v) fin para;
  para cada arista  $(x,y) \in \text{ARISTAS}$  hacer
    si BUSCAR(x)  $\neq$  BUSCAR(y)
      entonces UNIR(ETIQUETA(x), ETIQUETA(y))
    fin si;
  fin para;
fin proc;

```

Si G tiene K componentes conexas, ¿cuántas operaciones BUSCAR se realizan? ¿Cuántas operaciones UNIR se realizan? Exprésese el resultado en términos de $|\text{VERTICES}|$, $|\text{ARISTAS}|$ y K.

✎:

El número de operaciones BUSCAR es 2 por cada arista del grafo: $2 * |\text{ARISTAS}|$. El número de operaciones UNIR es $|\text{VERTICES}| - K$.

Justificación: Comenzamos con $|\text{VERTICES}|$ “componentes conexas”, y cada operación UNION reduce en 1 ese número. Si al final el número es K, el número de operaciones UNIR habrá sido $|\text{VERTICES}| - K$.

33. Dado un grafo G no dirigido, el algoritmo de Kruskal determina el árbol de expansión mínimo de G empleando para ello la estructura partición UNION-FIND. El método comienza ordenando el conjunto de aristas del grafo en orden creciente según sus pesos, para pasar a continuación a tratar una a una estas aristas. Analícese una nueva versión de dicho algoritmo en la cual se emplea un montículo (heap) para almacenar las aristas, y de donde se irán cogiendo una a una para ser tratadas de la misma forma que en la versión original. Escribese el algoritmo y calcúlese su orden en el caso peor.

✎:

```

Algoritmo KRUSKAL_MODIFICADO (G=(N, A), ...)
  Crear_Montículo(A, MONTON)
  Inicializar_partición(N)
  while Número de aristas seleccionadas  $\neq$  n-1 loop
     $\{u,v\} \leftarrow \text{Mínimo}(\text{MONTON})$ 
    Eliminar_raíz(MONTON)
    "SEGUIR IGUAL QUE KRUSKAL ORIGINAL"
  end loop
end

```

- Crear_Montículo(A, MONTON) es de $O(a)$ siendo a el número de aristas de A.
- Inicializar_partición(N) es de $O(n)$ siendo n el número de vértices de N.
- Mínimo(MONTON) es de $O(1)$.
- Eliminar_raíz(MONTON) es de $O(\lg m)$ siendo m el número de elementos de MONTON.

Hay que realizar el mismo número de operaciones unir, buscar que en el algoritmo de Kruskal original, es decir $O(a)$; pero además hay que rehacer el montículo (Eliminar_raíz(MONTON)) después de cada toma de la arista de menor peso, lo que en el caso peor significa hacerlo a veces. Como empezamos con a-1 aristas, esto resulta

$$\lg(a-1) + \lg(a-2) + \dots + \lg 1 \in O(\lg a)$$

Concluyendo, esta variante del algoritmo de Kruskal es de $O(a \lg a) = O(a \lg n)$.

34. Considérese la información registrada en una agencia de viajes referente a los vuelos existentes entre todos los aeropuertos del mundo. Supongamos que toda conexión aérea es de ida y vuelta. Queremos resolver el siguiente problema: Dados los aeropuertos A y B ¿Cuál es el mínimo número de transbordos en un viaje aéreo de A a B?

- (a) De los algoritmos de recorrido de grafos conocidos, cuál elegirías para resolver este problema y por qué. Indica cual sería la modificación pertinente del algoritmo elegido.
- (b) ¿Cuál es la complejidad temporal de cada uno de los algoritmos sobre grafos que conoces que resuelvan este problema?
- (c) En conclusión, cuál de ellos elegirías y por qué.

:

(a) De los dos algoritmos de recorrido conocidos preferiría el de recorrido en anchura. Por que me parece más adecuado para resolver un problema de distancia mínima en este grafo. Iniciar el proceso partiendo del nodo A, y terminar el proceso, aunque no hayamos recorrido todo el grafo, en cuanto se visite el nodo B, dando la distancia correspondiente.

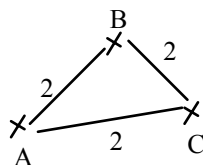
(b) Recorrido en anchura $O(n+a)$. Algoritmo de Dijkstra de cálculo de las distancias mínimas de un nodo a todos los demás $O(n^2)$. Algoritmo de Floyd que calcula las distancias mínimas entre cada par de vértices $O(n^3)$.

(c) Con la modificación indicada en (a) el recorrido en anchura parece el más eficiente; aunque en este caso el algoritmo de Dijkstra realiza casi la misma tarea.(en cada selección voraz elige “al más cercano”, y esa es precisamente la forma de trabajar del recorrido en anchura).

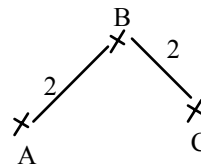
35.- Supongamos que tenemos un algoritmo $CAM(G,v)$ que, dado un grafo G con pesos no negativos asociados a las aristas y un vértice v del mismo, nos devuelve las distancias mínimas para ir de v al resto de vértices del grafo G . ¿Es cierto $CAM(G,v) = CAM(KRUSKAL(G),v)$? Recuerdese que $KRUSKAL(G)$ es un árbol de expansión mínimo de G .

:

No es cierto. He aquí un contraejemplo:
Dado el grafo



Una solución de Kruskal podría ser:



De manera que la distancia desde el vértice A al C es de 4 en esa solución.

Ahora bien, el camino más corto de A a C tiene longitud 2, que sería el resultado que obtendríamos al hacer CAM(G,A).

36. Tras haber visto el algoritmo de Dijkstra, el cual dado un grafo dirigido determina los costes de los caminos mínimos desde un vértice al resto, se pide modificar dicho algoritmo preservando el orden de manera que:

- (a) calcule además el NUMERO de caminos mínimos y
- (b) calcule lo suficiente para reconstruir esos caminos mínimos.

Justificar las respuestas.

[Pista:

Para que el algoritmo de Dijkstra original, además de determinar los costes de los caminos mínimos, calcule los propios caminos (secuencias de vértices), basta con añadirle otra tabla CAMINOS(1..n) que acumule dichos caminos de la siguiente forma: En CAMINOS(x) se registra la lista de vértices que preceden inmediatamente al vértice x en caminos mínimos de 1 a x. Por ejemplo si CAMINOS(x) = [x1, x2, x3] esto significa que (x1, x), (x2, x) y (x3, x) son últimas aristas respectivas de caminos mínimos distintos de 1 a x.]

✍:

Usaremos una tabla NCM(1..n) de números naturales para registrar el número de caminos mínimos del nodo 1 a cada nodo Y.

Indicamos la inicialización y la modificación pertinente del algoritmo.

```

for Y in 1..n loop
  D(I) := G(1, Y)
  if D(I) = • then
    NCM(I) := 0
    CAMINOS(I) := Lista_vacía
  else
    NCM(I) := 1
    CAMINOS(I) := Añadir( Lista_vacía, Nodo(1) )
  end loop

```

Una vez seleccionado el vértice X, para cada uno de sus adyacentes no seleccionados Y habrá que realizar lo siguiente:

```

if D(Y) > D(X) + G(X, Y) then
  D(Y) := D(X) + G(X, Y)

```

```

NCM(Y) := NCM(X)
CAMINOS(Y) := Añadir( Lista_vacíá, Nodo(X))
--Cambiar la lista anterior por el vértice X
elseif D(Y) = D(X)+G(X, Y) then
  NCM(Y) := NCM(Y)+NCM(X)
  CAMINOS(Y) := Añadir( CAMINOS(Y), Nodo(X)) --Caminos alternativos
end if

```

37. El problema de encontrar un subconjunto T de aristas de un grafo conexo G de manera que todos los vértices del grafo queden conectados empleando tan sólo las aristas de T, y la suma de los pesos de las aristas de T sea la menor posible sigue teniendo sentido aún cuando el grafo G tenga aristas con pesos negativos. Sin embargo, la solución puede que ya no sea un árbol. Adáptese el algoritmo de Kruskal para que trabaje con grafos cuyas aristas pueden tener pesos negativos.

En:

Para que la suma de pesos de las aristas de T sea mínima, deben incluirse en T todas las aristas negativas del grafo más las positivas (de menos peso) estrictamente necesarias para interconectar a todo los nodos del grafo. Una adaptación simple del algoritmo de Kruskal es la siguiente:

```

procedure KruskalExtendido (G: in Grafo; CA: out Aristas) is
  ...
begin
  L_Aris:=Ordenar_Crecientemente_las_Aristas_del_Grafo(ARISTAS(G));
  NumCompConexas:= CuantosVertices(VERTICES(G));
  ConjuntoVacio(CA);
  Inic_P_conjuntos_cada_uno_con_un_vértice_del_grafo(VERTICES(G),
                                                    Comp_Conexas);
  Primera_arista_sin_considerar_y_eliminarla_del_cjto(L_Aris, (X,Y));
  while NumCompConexas>1 or Peso(G, (X,Y))<0 loop
    X_dentro:= Buscar3(Comp_Conexas, X);
    Y_dentro:= Buscar3(Comp_Conexas, Y);
    if Peso(G, (X,Y))<0
    then Añadir_Arista_Cnjto(CA, (X,Y));
      if Distintos(Comp_Conexas, X_dentro, Y_dentro)
      then NumCompConexas:= NumCompConexas-1;
      end if;
      Fusionar3(Comp_Conexas, (X,Y));
    else
      if Distintos(Comp_Conexas, X_dentro, Y_dentro)
      then Añadir_Arista_Cnjto(CA, (X,Y));
        NumCompConexas:= NumCompConexas-1;
        Fusionar3(Comp_Conexas, (X,Y));
      end if;
    end if;
    Primera_arista_sin_considerar_y_eliminarla_
                                     del_cjto(L_Aris, (X,Y));
  end loop;
end;

```

Estudio del orden de ejecución de la solución propuesta:

Ambas soluciones tan sólo difieren en el código del 'if' principal del bucle. Pero en ambas soluciones, las ramas más costosas de los mismos requieren el mismo orden. Por otro lado, también en ambas soluciones el caso peor resulta cuando es necesario añadir al conjunto solución la última arista tratada (por ejemplo, cuando todas las aristas tiene peso negativo).

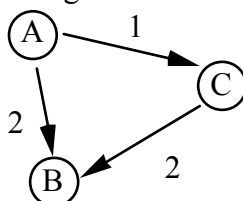
Así pues, la solución propuesta, al igual que la original de Kruskal es de $O((p+a) \log (p+a))$.

38. Sea un grafo dirigido acíclico con pesos no negativos en los arcos. Queremos un algoritmo que calcule la máxima distancia de un vértice origen a cada uno de los demás vértices. ¿Sería correcto un algoritmo voraz con la selección voraz siguiente?: Sea S el conjunto de vértices seleccionados (inicialmente $S = \{\text{Origen}\}$). Seleccionar el vértice $v \notin S$ cuyo *camino especial* sea de longitud máxima.
(La noción de *camino especial* es la análoga a la usada en el algoritmo de Dijkstra de cálculo de caminos mínimos.)

✎:

La selección voraz no es correcta. Proponemos el siguiente contraejemplo:

En el siguiente grafo, sea A el vértice origen



En el ejemplo primeramente existen dos caminos especiales máximos elegibles. $|AB|=2$ y $|AC|=1$. Puesto que el primero de ellos, es mayor, se escogería y admitiría éste, pasando así, el vértice B a pertenecer al conjunto S : $S=\{A,B\}$. Puesto que el algoritmo sería voraz, ningún candidato previamente considerado se reconsideraría, obligando ello a que la distancia máxima hasta B calculada fuera 2, cosa que es falsa: $|ACB|=3$.

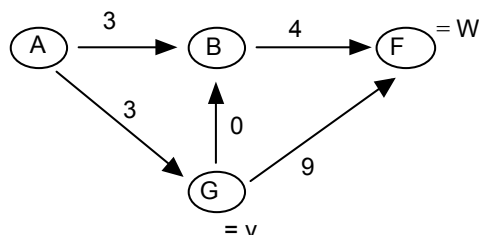
39. Sea un grafo dirigido con pesos mayores o iguales que cero en los arcos.

En el algoritmo de Dijkstra, sea w un nodo fuera del conjunto de seleccionados S tras haber añadido un nuevo nodo v a S . ¿Es posible que algún camino especial mínimo del origen a w pase por v y después por algún otro nodo de S ? La respuesta debe quedar justificada matemáticamente.

✎:

Sí, es posible que cree un nuevo camino ESPECIAL MÍNIMO también que partiendo del origen llegue a V , atraviere vértices de S y finalmente llegue hasta W (tal que $W \notin S$).

Ejemplo:



$S = \{ A, B \}$ y D :

	7	3	
F	G		

Se escoge G (que equivale al vértice V del enunciado) y se actualizan el conjunto solución S y la tabla de distancias D a:

$S = \{ A, B, G \}$ y D :

	7	-1	
F	G		

Hay DOS caminos ESPECIALES MÍNIMOS hasta F : $A-B-F$ y el nuevo $A-G-B-F$.

40. Escriba y analice un algoritmo que, dado un árbol T , determine la profundidad K con el máximo número de nodos de T . Si hubiera varias de esas profundidades, determínese la mayor.

Algoritmo:

Los nodos se tratan en el orden del recorrido en anchura partiendo de la raíz. Así, cada vez que se encole un nodo j de profundidad k , encolamos tanto el nodo como su profundidad. El tratamiento de un elemento tomado de la cola consiste en actualizar las variables pertinentes y encolar todos sus hijos.

```

procedure PROF_CON_MAS_NODOS (T: in ARBOL; Mayor_Prof: out entero) is
  COLA: COLA_TAD := VACIA_C;      Adyacentes_Y: L_NODOS;
  Y, Z: VERTICE;                  Actual_prof: entero;
  Prof: entero := 0;              --profundidad del nodo en curso
  Cont: entero := 0;              --número de nodos contabilizados en Actual_prof
  Max: entero;                    --número de nodos en Mayor_Prof
begin
  if ES_ARBOL_VACIO(T) then Mayor_Prof := -1;
  else
    Mayor_Prof := 0;   Max := 1;   Actual_prof := 0;
    AÑADIR_COLA(COLA, (RAIZ(T), Prof)); --la profundidad de la raíz es 0
    while not(ES_VACIA_C(COLA)) loop
      (Y, Prof) := CABEZA(COLA);
      RETIRAR_PRIMERO_DE(COLA);

      if Prof = Actual_prof then Cont := Cont + 1
      else if Cont > Max
        then Max := Cont; Mayor_Prof := Actual_prof
      end if
      Actual_prof := Prof
    end if;

    Adyacentes_Y := ADYACENTES(T, Y);
    while not (ES_VACIA_L(Adyacentes_Y)) loop
      Z := CABEZA(Adyacentes_Y);
      RESTO_L?(Adyacentes_Y);
      AÑADIR_COLA(COLA, (Z, Prof + 1));
    end loop;
  end loop;
  end if;
end PROF_CON_MAS_NODOS;

```

Análisis del orden del tiempo de ejecución:

Al recorrido en anchura tan sólo se le han añadido algunas operaciones, las cuales necesitan un tiempo constante de ejecución. Recordando que en un árbol $A=N-1$, el bucle es $O(N)$, donde $N=n^\circ$ de nodos y $A=n^\circ$ de arcos.

41.- Sea la función recursiva

$$f(n, m) = \begin{cases} 0 & \text{si } m = 0 \\ n & \text{si } m = 1 \\ f\left(n, \left\lfloor \frac{m}{2} \right\rfloor\right) + f\left(n, \left\lceil \frac{m}{2} \right\rceil\right) & \text{si } m > 1 \end{cases}$$

- (a) Es evidente que un programa recursivo ingenuo que calcule $f(n, m)$ repetirá cálculos. Al efecto de evitar estos cálculos redundantes, escribe el programa que calcula dicha función empleando para su diseño la técnica de programación dinámica.
- Utilizando funciones con memoria.
 - Utilizando una solución iterativa.
- (b) Analiza la complejidad de las soluciones del apartado anterior. ¿Encuentras alguna diferencia en el número de sumas a realizar?

✎ :

Puesto que la recursión para que terminen los cálculos depende de un sólo argumento, m , bastará con almacenar los resultados intermedios en una tabla unidimensional: $TAB(0..m)$.

(a.1) Ya que los cálculos intermedios de $f(n, m)$ no generarán nunca el valor -1, se escoge dicho valor como valor por defecto. (Suponemos $N \geq 0$, sino el valor por defecto a emplear podría ser $\min(-1, N-1)$.) Se inicializará toda la tabla con el mismo salvo las posiciones 0 y 1, las cuales se iniciarán con los valores 0 y n respectivamente. El resto de casillas que haya que rellenar, se rellenarán por demanda de las llamadas recursivas:

```
function PRINCIPAL (M,N: integer) return integer is
  Tab: array (integer range 0.. M)
        of integer := (0=>0; 1=>N; others=>-1);

  procedure AUX (M: integer) is
  begin
    if Tab(TECHO(M/2))=-1 then AUX(TECHO(M/2)); end if;
    if Tab(SUELO(M/2))=-1 then AUX(SUELO(M/2)); end if;
    Tab(M) := Tab(TECHO(M/2)) + Tab(SUELO(M/2));
  end AUX;

begin
  if Tab(M/2)=-1 then AUX(M); end if;
  return Tab(M);
end PRINCIPAL;
```

(a.2) En la solución iterativa es necesario rellenar una a una y de la posición 0 a la m todas las casillas de la tabla siguiendo la definición matemática de la función f :

```

function PRINCIPAL (M,N: integer) return integer is

    Tab: array (integer range 0.. M) of integer:= (0=>0; 1=>N);

begin
    for P in 2 .. M loop
        Tab(P) := Tab(TECHO(P/2)) + Tab(SUELO(P/2));
    end loop;
    return Tab(M);
end PRINCIPAL;

```

(b.1) El orden del tiempo de inicialización es $\Theta(M)$. Mientras que el cálculo de Tab(M) requerirá a lo sumo el relleno de $M/2$ celdas (ya que no es necesario calcular nuevamente los valores de las celdas de las posiciones [TECHO(P/2)+1..M-1]). Las que se calculan, tal y como indica la función f, necesitan tiempo constante: dos accesos a posiciones anteriores + una suma + una asignación. Por tanto concluimos, que el orden de la solución con funciones con memoria es $\Theta(M)+O(M)=\Theta(M)$.

(b.2) En este caso todas las celdas de la tabla se calculan una única vez. Cada posición necesita tiempo constante para ser calculada y almacenada. Por tanto, la solución iterativa es de $\Theta(M)$.

Siguiendo la argumentación de los dos puntos anteriores, fácilmente se puede observar que mientras que en la solución recursiva se necesitarán a lo sumo de $M/2$ sumas en la iterativa serán necesarias $M-2$.

42. ¿Qué técnica usarías para escribir un programa eficiente que calcule $f(n) = n + \frac{2}{n} \sum_{i=1}^{n-1} f(i)$ siendo $f(1)=1$? Justifica brevemente de qué orden sería ese programa.

✎:

Observado que la función esta definida mediante una ecuación recursiva y que el cálculo de ciertas sumas se reitera una y otra vez, la técnica de programación a emplear para obtener un programa eficiente sería la programación dinámica.

Aunque a primera vista parece suficiente con almacenar los valores intermedios $f(i)$ ($i \in [1,n]$) en una tabla, ello no evitaría el tener que calcular el sumatorio (y en consecuencia repetir muchas sumas) para cada nuevo valor de la función (en su rango correspondiente). Al no evitar recálculos de sumas, esta solución no se admitiría como programación dinámica.

No obstante, si tuviéramos una matriz bidimensional tal que:

Tabla F:

1	F(1)		F(I-1)	F(I)		F(n)
2	0		F(1)+...+F(I-1)	F(1)+...+F(I)		
	1		I-1	I		n

El relleno de la tabla se realizaría de izquierda a derecha de manera que en la iteración I bastaría con calcular:

```

Tabla_F(1)(I):= I+ 2/n Tabla_F(2)(I-1)
Tabla_F(2)(I):= Tabla_F(2)(I-1) + Tabla_F(1)(I)

```


Dichos cálculos requerirían tiempo constante. Puesto que habría que realizar n iteraciones, el cálculo de $F(n) \in O(n)$.

Obsérvese que en un nuevo refinamiento se podría evitar el tener una tabla bidimensional. Bastaría con dos variables, una para tener $F(I)$ y la otra para tener $F(1)+\dots+F(I)$ e ir las actualizando en cada iteración I . No obstante, el orden del tiempo de ejecución se mantendría, aunque el espacio extra de memoria empleado se reduciría ostensiblemente, hasta constatarlo.

43.- Sea A : array $(1..n)$ of integer con $n > 0$, y sea la función

```
function SUMA?(A(1..i)) return boolean is
-- i>0
  if i=1  $\wedge$  A(1) $\neq$ 0 then return false;
  elsif A(i)=suma(A(1..i-1)) then return true;
  else return SUMA?(A(1..i-1));
```

siendo

```
function suma(A(1..k)) return integer is
  s:integer:=0;
  for j:=1 to k loop s:=s+A(j) end loop
  return s;
```

La evaluación de $SUMA?(A(1..n))$ devuelve un valor booleano que indica si alguno de los elementos del vector $A(1..n)$ coincide con la suma de todos los elementos que le preceden. Analice la eficiencia de esta evaluación.

Utilice otra técnica para escribir un programa que resuelva el problema de un modo más eficiente. Analice la eficiencia de su propuesta.

\mathcal{L}_D :

La función $\text{suma}(A(1..i))$ es, obviamente, de $\Theta(i)$. Por tanto, la función de coste de $SUMA?(A(1..i))$ viene determinada por la recurrencia siguiente:

En el peor caso, se comprueba $A(i) = \text{suma}(A(1..i-1))$ y se invoca recursivamente a $SUMA?(A(1..i-1))$.

$$t(n) = \begin{cases} a & n \leq 1 \\ (n-1) + t(n-1) & n > 1 \end{cases}$$

Resolviéndola por expansión, se llega a una ecuación patrón $f(n) = \sum_{i=1}^k (n-i) + t(n-k)$ que

cuando $n-k=1$ resulta resolver $f(n) = \sum_{i=1}^{n-1} i + a$; y por tanto $f(n) = a + \frac{n(n-1)}{2} \in \Theta(n^2)$

Para resolver el problema de modo más eficiente basta con dedicar una variable

AUX:integer a registrar $\sum_{i=1}^{k-1} a(i)$ para cada k pertinente.

```
function SUMA?(A(1..i)) return boolean is
-- y>0
  AUX: integer:=0;
  k: Indice:=1;
begin
  while k<=i and AUX/=A(k) loop
    AUX:= AUX + A(k);
    k:= k+1;
  end loop
  if k<=i then return true;
  else return false;
```

```

    end if;
end;

```

Este algoritmo es de $\Theta(n)$ para $i=n$.

44.- Diseña y analiza un algoritmo que calcule la **clausura transitiva** \mathcal{R} de una relación binaria R . Dada una relación binaria R , $x\mathcal{R}y \Leftrightarrow xRy \vee (\exists z. x\mathcal{R}z \wedge z\mathcal{R}y)$

:

xRy puede representarse como un arco de un grafo dirigido. Esto significa que calcular la clausura transitiva equivale a determinar la existencia de un camino entre cada par de vértices de un grafo dirigido. Sea R la matriz de adyacencia de tal grafo:

$$R \times y = \begin{cases} true & \text{si } xRy \\ false & \text{si } \neg(xRy) \end{cases}$$

Una recurrencia análoga a la del algoritmo de Floyd para el cálculo de caminos mínimos nos resuelve el problema.

Sea $D_{ij}^k = \text{true}$, si existe camino de i a j usando los nodos $\{1,2,\dots,k\}$; y falso, si no.

Se satisface la recurrencia siguiente:

$$D_{ij}^k = D_{ij}^{k-1} \vee (D_{ik}^{k-1} \wedge D_{kj}^{k-1}) \quad \forall k > 1$$

$$D_{ij}^1 = R_{ij}$$

Y deseamos calcular D_{ij}^n para cada i,j . Siguiendo el mismo estudio que el algoritmo de Floyd, podemos calcular D_{ij}^n con n iteraciones de proceso sobre la misma matriz D .

```

begin
  D := R;
  for K in 1..N loop
    for I in 1..N loop
      for J in 1..N loop
        D(I,J) := D(I,J) or (D(I,K) and D(K,J));
      end loop;
    end loop;
  end loop;
end;

```

Este algoritmo, como el de Floyd, es de $\Theta(n^3)$.

45. Describe un algoritmo de **programación dinámica** para el problema de selección de actividades: Subconjunto de máxima cardinalidad de los intervalos $[s_i, f_i)$ $1 \leq i \leq n$, y sin solapamientos dos a dos. Se basará en el cálculo **iterativo** de NUM_i para $i=1,2,\dots,n$ donde NUM_i es el máximo número de actividades mutuamente compatibles del conjunto de actividades $\{1,2,\dots,i\}$. Supóngase que la entrada viene ordenada ascendentemente según los tiempos de finalización: $f_1 \leq f_2 \leq \dots \leq f_n$

Compara el tiempo que necesita este algoritmo frente al de la solución voraz: Selecciona el que antes termine y no se solape con los ya seleccionados.

✎:

Definimos **NUM(i)** = el máximo número de actividades mutuamente compatibles del conjunto de actividades $\{1,2,\dots,i\}$, con la recurrencia siguiente:

$$\text{NUM}(0) = 0$$

$$\text{NUM}(i) = \text{máximo} \{ \text{NUM}(i-1), \text{NUM}(j)+1 : f_j \leq s_i \}$$

(el 1 sumado a NUM(j) representa a la actividad número i formando parte de la selección).

Es evidente, con esta definición, que para toda actividad $i \geq 1$: $\text{NUM}(i-1) \leq \text{NUM}(i)$. Por tanto se satisface lo siguiente:

$$\text{NUM}(0) = 0$$

$$\text{NUM}(i) = \text{máximo} (\text{NUM}(i-1), \text{NUM}(k)+1) \text{ siendo } f_k = \max \{ f_j : f_j \leq s_i \}$$

El cálculo de NUM(i) siguiendo esta definición sin ninguna optimización precisa determinar el valor k, cálculo que se podría efectuar dicotómicamente con un coste $\Theta(\lg i)$ para cada $i=1..n$. Sin embargo, dicho cálculo es evitable, empleando espacio extra:

F(i) = hora de finalización mínima (la menos restrictiva) de la tareas que hacen máximo a Num(i).

$$[\text{NUM}(1), F(1)] = [1, f_1]$$

$$[\text{NUM}(i), F(i)] = [\text{NUM}(i-1)+1, f_i] \quad \text{si } F(i-1) \leq s_i$$

$$= [\text{NUM}(i-1), F(i-1)] \quad \text{si } F(i-1) > s_i$$

Esto permite un algoritmo simple de cálculo de NUM(n) en $\Theta(n)$ y con MEE(n) en $\Theta(n)$.

46. Tenemos n objetos de pesos p_1, \dots, p_n y valores v_1, \dots, v_n y una mochila de capacidad C . Escriba y analice un algoritmo que determine un subconjunto de objetos cuyo peso total no exceda la capacidad de la mochila y cuyo valor total sea maximal. Todas las cantidades C , v_i y p_i para $i=1, \dots, n$ son números naturales.

(No interesan soluciones que empleen el esquema de Backtracking.)

✎:

Obsérvese que las selecciones voraces siguientes no son correctas:

(

a) Seleccionar el objeto que menos pesa.

Contraejemplo: $p_1=1, v_1=1$ y $p_2=2, v_2=2$ y $C=2$.

(b) Seleccionar el objeto que más vale.

Contraejemplo: $p_1=2, v_1=3$ y $p_2=1, v_2=2$ y $p_3=1, v_3=2$ y $C=2$.

(c) Seleccionar el objeto con mayor proporción v_i/p_i .

Contraejemplo: $p_1=7, v_1=8$ y $p_2=5, v_2=5$ y $p_3=6, v_3=6$ y $C=11$.

El intento de especificar el beneficio máximo que buscamos como:

$$B(C) = \max [i:1..n] \{ v_i + B(C-p_i) / p_i \leq C \}$$

no es adecuado ya que no se considera si $B(C-p_i)$ se conseguirá utilizando el objeto i.

Esto nos lleva a proponer la siguiente especificación

Sea $TASA(C, k)$ el valor máximo que puede contener la mochila cuando sólo se pueden introducir en ella los objetos $1..k$ y sin exceder nunca la capacidad C .
 Habrá que estudiar si el valor máximo para $TASA(C, k)$ se consigue usando el objeto k o sin usarlo, y determinar los casos básicos:

$$\begin{aligned} TASA(0, k) &= 0 \\ TASA(C, 0) &= 0 \\ TASA(C, k) &= 0 \quad \text{si } \forall (1 \leq i \leq k). \text{ PESO}(i) > C \\ TASA(C, k) &= \max \{ TASA(C, k-1), TASA(C-p_k, k-1) + v_k / p_k \leq C \}. \end{aligned}$$

El valor deseado $TASA(C, n)$ puede calcularse rellenando una matriz **tasa** ($i:0..C, j:0..n$) según la fórmula y en el orden siguiente: para cada $j:0..n$ rellenar cada $i:0..C$.

Recogeremos en **objetos**(i, j) el objeto $\{j\}$ si con él se consiguió el valor máximo para **tasa**(i, j) o bien $\{\}$ en otro caso (Nótese que podría representarse con los valores booleanos **true** y **false** respectivamente) Necesitaremos, posteriormente, un algoritmo que recolecte los objetos seleccionados.

Sea **OBJETOS**(C, k) un subconjunto de objetos de $1..k$ que maximiza $TASA(C, k)$

El valor deseado **OBJETOS**(C, n) se calcula con la fórmula siguiente:

$$\begin{aligned} \text{OBJETOS}(C, 0) &= \{\} \\ \text{OBJETOS}(0, n) &= \{\} \\ \text{OBJETOS}(C, k) &= \text{OBJETOS}(C, k-1) \text{ si } \text{objetos}(C, n) = \{\} \\ &= \{n\} \cup \text{OBJETOS}(C-p_n, n-1) \text{ si } p_n \leq C \text{ y } \text{objetos}(C, n) = \{n\} \end{aligned}$$

Por todo ello es fácil comprobar que el cálculo de **OBJETOS**(C, n) es de $O(n)$ y el de $TASA(C, n)$ es de $O(Cn)$ y por tanto este último sería el orden de nuestro algoritmo. Asimismo, es obvio que el espacio extra utilizado es de $O(Cn)$.

47. Dadas n clases de monedas v_1, v_2, \dots, v_n y sabiendo que de cada clase se disponen de tantas monedas como se precisen, escríbase un algoritmo que determine cuantas combinaciones **distintas** de monedas hay para devolver una cierta cantidad C .

Ej.: Si las clases de monedas fueran: $v_1 = 25$, $v_2 = 50$, $v_3 = 75$ y $v_4 = 100$

Cantidad	Combinaciones distintas
100	5 formas: $25+75$, $2 * 50$, 100 , $4 * 25$, $50 + 2 * 25$
116	0, no hay ninguna combinación

Observación: La combinación $25+75$ es la misma que $75+25$ y por ello es una única combinación a considerar.

✍:

Utilizaremos la técnica de programación dinámica. Planteamos la recurrencia siguiente para resolver el problema.

Sea $\text{COMB}(C, k)$ el número de combinaciones distintas de monedas $\{v_1, v_2, \dots, v_k\}$ para sumar C .

-Para $C > 0$ y $k > 0$:

$$\text{COMB}(C, k) = \text{COMB}(C, k-1) + \text{COMB}(C-v_k, k-1) + \dots + \text{COMB}(C-\lfloor C/v_k \rfloor v_k, k-1)$$

-Para $C=0$ y $k \geq 0$

$$\text{COMB}(0, k) = 1 \quad \text{--La combinación es dar cero monedas de cada clase.}$$

-Para $C > 0$ y $k=0$

$$\text{COMB}(C, 0) = 0 \quad \text{--Si no hay monedas no hay forma de sumar algo mayor que cero.}$$

Pero otra recurrencia posible es la siguiente

$\text{COMB}(C, k) = \text{COMB}(C, k-1) + \text{COMB}(C-v_k, k)$ dejando la definición de los casos básicos como ejercicio para el lector. El valor que deseamos calcular es $\text{COMB}(C, n)$. Para ello, rellenaremos una tabla indexada por $1..C$ y $\{v_1, v_2, \dots, v_k\}$ siguiendo la fórmula de esta segunda recurrencia. Necesitamos rellenar $C \cdot n$ casillas y cada casilla podemos calcularla en tiempo constante. Por tanto el orden es $O(C \cdot n)$.

Analiza el algoritmo que resulta de utilizar la primera recurrencia tal cual está descrita.

48. Una *subsecuencia* de una secuencia S se obtiene retirando de S cualquier número de elementos de cualesquiera posiciones. Por ejemplo: $acxb$ es una subsecuencia de $bbacabxxb$.

La función recursiva f siguiente define (por casos) una *subsecuencia común de longitud maximal* de dos secuencias:

$$f(R, \epsilon) = \epsilon$$

$$f(\epsilon, S) = \epsilon$$

$$f(a \cdot R, a \cdot S) = a \cdot f(R, S)$$

$$f(a \cdot R, b \cdot S) = \text{la de mayor longitud entre } f(R, b \cdot S) \text{ y } f(a \cdot R, S), \quad (a \neq b)$$

Por ejemplo:

$$f(\text{promotor}, \text{prometedor}) = \text{promtor},$$

$$f(\text{aturdido}, \text{tranquilo}) = \text{trio},$$

$$f(\text{si}, \text{no}) = \epsilon.$$

Diseña un algoritmo, utilizando la técnica de programación dinámica (sin usar funciones con memoria), que calcule la longitud de la subsecuencia $f(R, S)$. ¿De qué orden es el tiempo y espacio empleado por tu solución?

✎:

Como toda solución que deba ser implementada con la técnica de la "programación dinámica" daremos:

- ecuación recursiva que resuelve el problema,
- tabla o matriz que se necesitará para almacenar las soluciones intermedias,
- algoritmo,
- orden tanto espacial como temporal requerido por la solución calculada.

- (1) $llcs(R, \epsilon) = 0$
 $llcs(\epsilon, S) = 0$
 $llcs(a \bullet R, a \bullet S) = 1 + llcs(R, S)$
 $llcs(a \bullet R, b \bullet S) = \max \{ llcs(R, b \bullet S), llcs(a \bullet R, S) \}$ si $(a \neq b)$
- (2) Necesitamos una matriz $n \times m$ tal que $n = R.Length$ y $m = S.Length$

TAB_llcs

	ϵ	R(n)	R(n-1..n)	j°: R(n-j+1..n)	R(1..n)
ϵ	0	0	0		0
S(m)	0				
S(m-1..m)	0				
i°: S(m-i+1..m)				③	
S(1..m)	0				

TAB_llcs(i,j)

= ③ = longitud de la subsecuencia común de mayor longitud de R(n-j+1..n) y S(m-i+1..m); es decir, los últimos j elementos de R y los últimos i elementos de S.

Primero, hay que inicializar a 0 las casillas de la fila 0 y la columna 0. Luego, la forma de rellenar la tabla es de arriba hacia abajo y de izquierda a derecha con los. La definición recursiva se recoge en la tabla de la forma siguiente:

TAB_llcs(i,j)

= $1 + TAB_llcs(i-1, j-1)$ si $R(n-j+1) = S(m-i+1)$
= $\max \{ TAB_llcs(i-1, j), TAB_llcs(i, j-1) \}$ si $R(j) \neq S(i)$

La solución a devolver es la cantidad que se obtenga en TAB_llcs(m,n).

- (3) Suponemos que las secuencias son S(1..m) y R(1..n). Puesto que se pide una solución iterativa:

```
function LONG_LCS (S,R: in Vector) return Integer is
  m : S.Length;      n: R.Length;
  TAB_llcs: array (0..m,0..n);
begin
  for I in 0..m loop TAB_llcs(I,0) := 0; end loop;
  for J in 0..n loop TAB_llcs(0,J) := 0; end loop;
  for I in 1..m loop
    for J in 1..n loop
      if S(m-I+1) = R(n-J+1)
      then TAB_llcs(I,J) := 1+ TAB_llcs(I-1,J-1);
      else TAB_llcs(I,J) := MAX( TAB_llcs(I-
1,J), TAB_llcs(I,J-1));
      end if;
    end loop;
  end loop;
end LONG_LCS;
```

- (4) Suponemos que las secuencias son S(1..m) y R(1..n).
- Orden temporal: $T(m,n) \in \Theta(m+n+mn) = \Theta(mn)$
 - Orden espacial: $MEE(m,n) \in \Theta(mn)$ debida a la matriz empleada para almacenar los resultados intermedios.

49. Sea $G=(V,A)$ un grafo orientado con pesos no negativos en los arcos y $|V|=n$. Sea L la matriz de adyacencia que lo representa. Queremos calcular la distancia mínima entre cada par de vértices $i,j \in V$, usando la siguiente fórmula:

D_{ij}^p = distancia mínima de i a j cuando podemos dar, a lo sumo, p pasos.

Tenemos los valores básicos $D_{ij}^1 = L_{ij}$ y para todos los i y j buscamos D_{ij}^{p-1} sabiendo que D_{ij}^p se define recursivamente: $D_{ij}^p = \min_{k \text{ adyacente a } i} \{D_{ij}^{p-1}, L_{ik} + D_{kj}^{p-1}\}$

Resuelve el problema expuesto empleando la técnica de la programación dinámica (sin funciones con memoria), la cual deberá recoger la definición recursiva arriba expuesta; es decir, se pide:

- determinar la estructura de almacenamiento de las soluciones parciales que se va a emplear: vector o matriz, cada posición de la estructura qué recoge o representa, cuál es la inicialización de la estructura, cómo se irá rellenando, dónde está la solución.
- algoritmo iterativo que calcula la definición recursiva sobre la estructura escogida y descrita en el apartado (a).
- análisis del orden del coste temporal y espacial: $O(T(n))$ y $O(MEE(n))$.

✎:

(a) Para calcular la fórmula del enunciado de forma directa son necesarias dos matrices $n \times n$, D y D_aux , puesto que una vez calculada la componente (i,j) en el paso p -ésimo (es decir, D_{ij}^p) todavía se podría necesitar D_{ij}^{p-1} para calcular otra componente (h,j) del paso p -ésimo (es decir, D_{hj}^p) si resulta que i es adyacente a h ya que $D_{hj}^p = \min\{D_{hj}^{p-1}, \dots, L_{hi} + D_{ij}^{p-1}, \dots\}$ Así pues, mientras que en la iteración p las distancias mínimas dando a lo sumo $(p-1)$ pasos estarán almacenadas en la matriz D , en la otra se irán almacenando las distancias mínimas dando a lo sumo p pasos. Una vez, que todas las distancias mínimas con a lo sumo p paso se hayan calculado, estas se copiarán a la matriz D . Inicialmente la matriz D se iniciará con los valores de L , ya que $D_{ij}^1 = L_{ij}$. El valor de las celdas de la matriz D_aux se pueden calcular de izquierda a derecha y de arriba a bajo. De esta forma, fijados un p , un i y un j tras operar $D(i,j) = D_{ij}^{p-1}$ y $D_aux(i,j) = \min_{k \text{ adyacente a } i} \{D_{ij}^{p-1}, L_{ik} + D_{kj}^{p-1}\}$.

(b) El siguiente algoritmo calcula las distancias mínimas entre pares de vértices tal y como lo indica el enunciado y empleando/rellenando las matrices descritas en el apartado anterior es el siguiente.

```

procedure Dist_Minimas      (L: in MAT_REAL; D: out MAT_REAL) is
    Aux: FLOAT;
    D_aux: MAT_REAL;

```

```

begin
  D:=L;
  for p in D'RANGE(1) loop
    for i in D'RANGE(1) loop
      for j in D'RANGE(1) loop
        Aux:=D(i,j);
        para k adyacente desde i hacer
          Aux:= min(Aux, L(i,k)+D(k,j);
        fin para;
        D_aux(i,j) :=Aux;
      end loop;
    end loop;
    D:=D_aux;
  end loop;
end;

```

(c) Es obvio que el algoritmo es de $O(n^4)$. Por otro lado, y ya que el algoritmo ha necesitado de la matriz auxiliar D_aux para calcular resultados intermedios, el orden de memoria extra consumida es cuadrático en n , esto es, $O(MEE(n^2))$.

50.- Sea el siguiente programa, con X e Y números positivos:

```

function PRINCIPAL (Num: positive) return real is
  Tabla: array (integer range 1.. Num)
    of real:= (1=>X; 2=>Y; others=>0);
  function AUX (Num: natural) return real is
  begin
    if Tabla(Num-1)=0
    then Tabla(Num-1) := AUX(Num-1);
    end if;
    if Tabla(Num-2)=0
    then Tabla(Num-2) := AUX(Num-2);
    end if;
    Tabla(Num) := (Tabla(Num-1)+Tabla(Num-2)) / 2
  end AUX;
begin
  if Num≤2 then return Tabla(Num);
  else return AUX(Num);
  end if;
end PRINCIPAL;

```

- Define la función que calcula $PRINCIPAL(n)$.
- Indica qué técnica se ha usado en el diseño del programa anterior.
- Analiza el tiempo de ejecución de $PRINCIPAL(n)$.
- ¿Usarías este algoritmo si como datos de entrada tuviéramos $X=1$ e $Y=1$?
¿Por qué?

Res:

(a)

$$PRINCIPAL(n) = \begin{cases} X & \text{si } n = 1 \\ Y & \text{si } n = 2 \\ \frac{PRINCIPAL(n-1) + PRINCIPAL(n-2)}{2} & \text{si } n > 2 \end{cases}$$

(b) La técnica empleada es programación dinámica con funciones con memoria. Tabla es estructura de memoria donde se acumulan los resultados parciales a partir de los cuales se genera el resultado de subproblemas de tamaño mayor.

(c) Claramente se distinguen los siguientes bloques de operaciones con sus respectivos órdenes:

- La inicialización de las n posiciones de `Tabla` requiere $\Theta(n)$.
- Las posiciones $[3..n]$ se rellenan una vez más. Para calcular el valor a asignar a una de esas posiciones (i) hay que acceder a las dos posiciones anteriores ($i-2$ e $(i-1)$), hacer una suma y una división y luego asignar el valor resultante en la posición (i). Todo ello requiere tiempo constante. Pero como se realiza $(n-2)$ veces, estas operaciones necesitan también en total $\Theta(n)$.
- La devolución necesita tiempo constante.

Aplicando la regla de la suma el algoritmo es $\Theta(n)$.

(d) No. Ya que en ese caso calcula `PRINCIPAL(n)` es la función constante 1(i.e. bastaría con para cualquier entrada n positiva devolver 1), para lo cual es suficiente con $\Theta(1)$.

51.- Escribe un programa que decida si un natural dado N es producto de tres naturales consecutivos y cuyo orden temporal sea $\Theta(n)$. Calcula su orden temporal.

 :

Nos piden determinar si existe cierto natural Y , que verifique $Y * (Y+1) * (Y+2) = N$ siguiendo una técnica dicotómica. De la ecuación anterior se desprende que en caso de que exista, $Y \in [0..N]$.

```
function ESTR_DIC (CInf,CSup: in Indice) return Par is
-- Se supone: CSup≥CInf
TalVez_Y: Indice := SUELO((CSup-CInf)/2);
N_Aux: Natural;
begin
  if CSup<CInf then return (False, 0);
  else N_Aux:= TalVez_Y * (TalVez_Y+1) * (TalVez_Y+2);
    if N_Aux = N then return (True, TalVez_Y);
    elsif N_Aux>N
    then return ESTR_DIC(CInf, Tal_Vez_Y-1);
    else return ESTR_DIC(Tal_Vez_Y+1, CSup);
    end if;
  end if;
end ESTR_DIC;
```

La estrategia del programa anterior es claramente dicotómica puesto que el intervalo de búsqueda del natural Y se reduce en cada llamada recursiva a un intervalo de tamaño mitad. Además, ya que para cada intervalo el número de operaciones a realizar requieren tiempo constante, el orden de la solución expuesta aquí es $O(\lg N) \subseteq \Theta(N)$.

52. Es conocido que la versión clásica del algoritmo *QuickSort*($T(1..n)$) es de $\Theta(n^2)$ en el caso peor. Usando los siguientes algoritmos, ambos de $\Theta(n)$ en el caso peor: *Mediana*($T(1..n)$, M) que calcula la mediana M del vector de enteros $T(1..n)$ y *Clasificar*($T(1..n)$, P) que permuta los elementos de $T(1..n)$ de manera que, al final:

si $T(i) < P$ y $T(j) = P$ entonces $i < j$

si $T(j)=P$ y $T(k)>P$ entonces $j<k$

escribe y analiza otra versión de $QuickSort(T(1..n))$ cuya complejidad temporal sea $O(n^2)$ en el caso peor.

✎ :

```

proc QuickSort_mediana (T(i..j))
  if i<j then
    (1)      Mediana (T(i..j), M)
    (2)      Clasificar (T(i..j), M)
    (3)      p:= El menor índice de i..j tal que T(p)=M
    (4)      q:= El mayor índice de i..j tal que T(q)=M
              QuickSort_mediana (T(i..p-1))
              QuickSort_mediana (T(q+1..j))
  end if

```

En el peor caso (1)+(2)+(3)+(4) es de $\Theta(n)$ y la función temporal de QuickSort_mediana ($T(1..n)$) responde a la recurrencia siguiente $f(n) = 2f(n/2) + \Theta(n)$ que pertenece a $\Theta(n \lg n)$.

53. El algoritmo de Floyd calcula la distancia mínima entre cada par de nodos de un grafo orientado con pesos no negativos asociados a los arcos. Modifícalo para que además calcule el número de caminos con distancia mínima que hay entre cada par de nodos.

✎ :

La recurrencia definida por Floyd es: $C_{i,j}^k = \min \{C_{i,j}^{k-1}, C_{i,k}^{k-1} + C_{k,j}^{k-1}\}$ la cual indica:

- $C_{i,j}^k = \min \{C_{i,j}^{k-1}, C_{i,k}^{k-1} + C_{k,j}^{k-1}\}$ el coste más barato para ir de i a j atravesando a lo sumo los vértices $[1..k]$
- $C_{i,k}^{k-1} + C_{k,j}^{k-1}$ indica el coste de ir i a j atravesando k.

Al igual que se calcula en la matriz C los costes, podemos emplear otra matriz N (con las mismas dimensiones) para determinar el nº de caminos con distancias mínimas que hay entre cada par de nodos. LA misma habrá que actualizarla de la siguiente forma:

- Hay que iniciar N correctamente. Primeramente, sólo se puede ir de i a j si hay arco entre i y j. Esto es, si $C_{i,j}^0 < \infty$ y en este caso el camino es único (suponemos que la no existencia de arco está señalizado con peso ∞).
- Si $C_{i,j}^{k-1} = C_{i,k}^{k-1} + C_{k,j}^{k-1}$ y ($i \neq k$ y $k \neq j$)
entonces $C_{i,j}^k = C_{i,j}^{k-1}$ y $N_{i,j}^k = N_{i,j}^{k-1} + N_{i,k}^{k-1} * N_{k,j}^{k-1}$.
- Si $C_{i,j}^{k-1} > C_{i,k}^{k-1} + C_{k,j}^{k-1}$ entonces $C_{i,j}^k = C_{i,k}^{k-1} + C_{k,j}^{k-1}$ y $N_{i,j}^k = N_{i,k}^{k-1} * N_{k,j}^{k-1}$.

- Si $C_{i,j}^{k-1} < C_{i,k}^{k-1} + C_{k,j}^{k-1}$ entonces $C_{i,j}^k = C_{i,j}^{k-1}$ y $N_{i,j}^k = N_{i,j}^{k-1}$.

Esto significa que el algoritmo de Floyd retocado sería:

```

procedure FloydRetocado  (Dist: in MAT_REAL; C: in out MAT_REAL;
                           N: in out MAT_REAL) is
begin
  C := Dist;
  Iniciar_A_Ceros_Y_Unos(C, N);
  for K in D'RANGE(1) loop
    for I in D'RANGE(1) loop
      for J in D'RANGE(1) loop
        if C(i,j) = C(i,k)+C(k,j)
        then N(i,j) := N(i,j) + N(i,k)*N(k,j);
        elsif C(i,j) > C(i,k)+C(k,j)
        then C(i,j) := C(i,k)+C(k,j);
              N(i,j) := N(i,k)*N(k,j);
        end if;
      end loop;
    end loop;
  end loop;
end;

```

54. Imagina un robot situado sobre unos raíles sin fin, con posibilidad de movimiento de un paso a derecha o izquierda con *actualizar(situacion, sentido)* y con posibilidad de advertir la presencia de un objeto buscado con *esta_en(situacion)*. El siguiente algoritmo mueve “pendularmente” al robot en busca del objeto citado, partiendo del origen 0.

```

limite:= 1
sentido:= derecha
loop
  situacion:= 0
  while situacion<limite loop
    actualizar(situacion, sentido)
    if esta_en(situacion)
    then return (situacion, sentido)
    end if
  end loop
  sentido:= cambiar(sentido)
  for i in 1..limite loop
    actualizar(situacion, sentido)
  end loop
  limite:= 2*limite
end loop

```

Sabiendo que el objeto será encontrado, analiza el número de veces que se realizará la operación *actualizar(situacion, sentido)* en función de la distancia del objeto al origen.

✎ :

Los bucles (1) y (2) son, respectivamente, de ida y vuelta. Sumados, realizan $2 \cdot \text{limite}$ veces la operación.

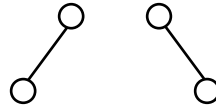
Supongamos que la distancia al origen es d siendo $2^{k-1} < d \leq 2^k$. Entonces, en el peor caso, el número de veces que se realiza la operación es el siguiente (obsérvese que al principio $\text{limite}=1$)

$$2 + 2 \cdot 2 + 2 \cdot 2^2 + \dots + 2 \cdot 2^{k-1} + 2^k = 2 \cdot (2^k - 1) + 2^k = 3 \cdot 2^k - 2$$

Siendo $d = 2^k$ tenemos el número de realizaciones de la operación es $\Theta(d)$.

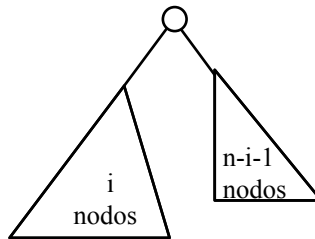
55.- ¿Cuántos árboles binarios distintos se pueden formar con n nodos? Sea $C(n)$ ese número.

Sabemos que $C(0)=1$ -el árbol vacío-, $C(1)=1$ -un árbol con sólo raíz- y $C(2)=2$ los árboles:



Puedes dibujar los árboles para calcular $C(3)$ y $C(4)$.

- (a) Encuentra una definición recursiva para calcular $C(n)$ considerando el siguiente diagrama:



Puedes comprobarla con tus dibujos para $n=3,4$.

- (b) Escribe el algoritmo que calcule $C(n)$ siguiendo tu recurrencia con la técnica de *Programación Dinámica* e indica su complejidad.

✍ :

- (a) Obsérvese que por cada árbol binario que podamos formar con i nodos tenemos $C(n-i-1)$ posibilidades en la parte derecha. Por tanto tenemos:

$$\begin{aligned} C(0) &= 1 \\ C(1) &= 1 \\ C(n) &= \sum_{i=0}^{n-1} (C(i) * C(n-i-1)) \quad \text{si } n \geq 2 \end{aligned}$$

- (b) En esta ocasión hemos optado por la programación dinámica iterativa:

```
function N_ARBIN (Num: natural) return natural is
  C: array (integer range 0.. Num) of natural :=
    (0, 1 => 1; others => 0);

  Suma: natural;

begin
  for K in 2..Num loop
    Suma := 0;
    for I in 0..K-1 loop
      Suma := Suma + (C(I) * C(K-I-1));
    end loop;
    C(K) := Suma;
  end loop;
  return C(Num);
```

end N_ARBIN;

Obs: Se podría incluso haber evitado el uso de la variable auxiliar Suma lo cual requeriría acumular directamente sobre C(K) y hacer sucesivos accesos y asignaciones a posiciones de la tabla C.

La complejidad de la solución viene determinada por el sumatorio

$$T(n) = \sum_{K=2}^n \sum_{i=0}^{K-1} 1 = \sum_{K=2}^n K \in \Theta(n^2)$$

Y el espacio extra de memoria consumido lo determina la tabla C empleada, i.e., $\Theta(n)$.

56. Escribe la recurrencia que sirve de fundamento para la aplicación de la técnica de programación dinámica al problema de minimización del número de productos escalares en la multiplicación de una serie de matrices $A_1 A_2 \dots A_n$.

El algoritmo que resuelve este problema calcula una tabla factor(1..n,1..n) tal que factor(i,j)=k si la factorización óptima es $(A_i \dots A_k)(A_{k+1} \dots A_j)$. Escribe y analiza un algoritmo que, a partir de factor(1..n,1..n), imprima la parentización óptima del producto $A_1 A_2 \dots A_n$.

 :

$M(i,j) = \text{mínimo } [i \leq k \leq j-1] \{M(i,k) + M(k+1,j) + d_{i-1} d_k d_j\}$ para $1 \leq i < j \leq n$

$M(i,i) = 0$

siendo $d_{i-1} d_i$ la dimensión la matriz A_i .

```

proc IMPRIME_PARENTESIS (i, j)
if i<j then
  Imprime (" (")
  IMPRIME_PARENTESIS (i, factor(i,j))
  Imprime (" * ")
  IMPRIME_PARENTESIS (factor(i,j)+1, j)
  Imprime (" ")
else --i=j
  Imprime ("Ai")

```

Es un recorrido en inorden del árbol binario de análisis de la expresión parentizada. Por tanto es lineal en el número de nodos de ese árbol que es $\Theta(n)$.

57. Di de qué orden es la siguiente recurrencia:

$$T(n) = \begin{cases} 1 & n \leq 4 \\ T\left(\frac{n}{4}\right) + \sqrt{n} + 1 & n > 4 \end{cases}$$

 :

Se opta por resolver una simplificación de la ecuación propuesta

$$T(n) = \begin{cases} 1 & n \leq 4 \\ T\left(\frac{n}{4}\right) + \sqrt{n} & n > 4 \end{cases}, \text{ puesto que en cuanto a orden son equivalentes:}$$

$$\begin{aligned}
T(n) &= T(n/4) + \sqrt{n} \\
&= T(n/4^2) + \sqrt{\frac{n}{4}} + \sqrt{n} \\
&= T(n/4^3) + \sqrt{\frac{n}{4^2}} + \sqrt{\frac{n}{4}} + \sqrt{n} \\
&= \dots = T(n/4^i) + \sqrt{\frac{n}{4^{i-1}}} + \dots + \sqrt{\frac{n}{4^2}} + \sqrt{\frac{n}{4}} + \sqrt{n} \\
&= T(n/4^i) + \sqrt{n} + \sqrt{n} \sum_{k=1}^{i-1} \frac{1}{\sqrt{4^k}} = T(n/4^i) + \sqrt{n} + \sqrt{n} \sum_{k=1}^{i-1} \frac{1}{2^k} \\
&\quad \bullet \quad n=4^i; \quad \sqrt{n}=2^i; \quad \lg \sqrt{n}=i \\
&= 1 + \sqrt{n} + \sqrt{n} \left(1 - \frac{2}{\sqrt{n}}\right) = 1 + 2\sqrt{n} - 2 \in \Theta(\sqrt{n})
\end{aligned}$$

58. Es conocido que $O(n) \subset O(n * \lg n) \subset O(n * \sqrt{n}) \subset O(n^2) \subset O(n^3)$. Clasifica $O(T(n))$ en esa cadena de inclusiones siendo:

$$T(n) = \begin{cases} 1 & n \leq 1 \\ 4 * T\left(\frac{n}{3}\right) + n & n > 1 \end{cases}$$

Tal vez te resulte útil saber que: $\log_3 4 = 1.26184$

 :

$$\begin{aligned}
T(n) &= 4 T(n/3) + n \\
&= 4 (4 T(n/3^2) + (n/3)) + n = 4^2 T(n/3^2) + 4/3 n + n \\
&= 4^2 (4 T(n/3^3) + (n/3)^2) + 4/3 n + n = 4^3 T(n/3^3) + (4/3)^2 n + 4/3 n + n \\
&= \dots = 4^i T(n/3^i) + (4/3)^{i-1} n + \dots + (4/3)^2 n + 4/3 n + n \\
&= 4^i T((n/3)^i) + n \sum_{k=0}^{i-1} \left(\frac{4}{3}\right)^k = 4^i T(1) + n \frac{\left(\frac{4}{3}\right)^i - 1}{\frac{4}{3} - 1} \\
&\quad \bullet \quad \text{puesto que todos los logaritmos son del mismo orden, tomaremos logaritmos en base 3} \\
&\quad \bullet \quad n=3^i; \quad \lg_3 n = i; \\
&= 4^i + 3n \frac{4^i}{3^i} - 3n = 4^{\lg_3 n} + 3 \cdot 4^{\lg_3 n} - 3n \\
&= 4 n^{\lg_3 4} - 3n = 4 n^{1.26184} - 3n \in \Theta(n^{1.26})
\end{aligned}$$

En lo referente a la clasificación del resultado, es sabido que:

$$O(n * \sqrt{n}) = O(n^{1.5})$$

$$O(n \cdot \lg n) \subset O(n^{1+a}) \quad \forall a > 0$$

Luego, $O(n \cdot \lg n) \subset O(n^{1.26}) \subset O(n \cdot \sqrt{n})$

59. Escribe un algoritmo que dados un vector de n enteros y un entero X , determine si existen en el vector dos números cuya suma sea X . El tiempo de tu algoritmo debe ser de $O(n \cdot \lg n)$. Analiza tu algoritmo y demuestra que es así.

✎ :

Se propone el siguiente algoritmo:

```
function buscarYZ (Val: in Tabla(1..N); X: in Integer) return
Integer is
  Y,Z: Integer; Hallado:boolean;
  Valor: Tabla(1..N);
begin
  MergeSort(Val, Valor);
  for Ind in 1..N-1 loop
    Y:=Valor(Ind);
    Z:=X-Y;
    BUSQUEDA_DICOTOMICA(Valor(Ind+1..N, Z, Pos);
    if Valor'First<=Pos and Pos <=Valor'Last
      then return Pos; --naturalmente la pareja es (Ind, Pos)
    end if;
  end loop;
  return 0; -- suponiendo que 0 no sea índice válido de Val, o bien
            -- podría devolverse Val'First-1
end;
```

Análisis del orden del algoritmo:

1. La ordenación de n enteros mediante el MergeSort es $\Theta(n \lg n)$
2. Se repetirá a lo sumo n veces un bucle cuyo coste temporal es:
 - Dos asignaciones + sentencia if requieren $O(1)$
 - La búsqueda dicotómica es logarítmica en el número de elementos: $O(\lg i)$ (y concretamente $\forall i(i \leq n \rightarrow O(\lg i) \subseteq O(\lg n))$)
 Luego, el bucle es $O(n \lg n)$

Así pues, y aplicando la regla de la suma, la solución propuesta tiene orden: $\Theta(n \lg n)$

60. El **cuadrado** de un grafo dirigido $G = (V, A)$ es el grafo $G^2 = (V, B)$ tal que $(u, w) \in B$ si y sólo si para algún $v \in V$ tenemos que $(u, v) \in A$ y $(v, w) \in A$. Es decir, G^2 tiene un arco de u a w cuando G tiene un camino de longitud (exactamente) 2 de u a w . Describe algoritmos eficientes que calculen G^2 a partir de G para las dos representaciones conocidas: matriz de adyacencia y lista de adyacencias. Analiza el tiempo de tus algoritmos.

 :

Podemos usar la misma idea con cualquiera de las dos representaciones. Llamemos V al conjunto de adyacentes de u y W al conjunto de los adyacentes de los nodos de V . El conjunto W son los adyacentes de u en el nuevo grafo G^2 .

- matriz de adyacencia

El enunciado no indica nada si el grafo es pesado o no, ni si la matriz de adyacencia es de booleanos o numérica, ni si tiene ciclos de longitud 1 o no. Las siguientes suposiciones no restringen generalidad alguna de la solución:

1. Supondremos que la matriz de adyacencia es booleana, tal que si el vértice v es adyacente de u entonces $G(u,v)=\text{True}$, y False en caso contrario.
2. Supondremos que no hay ciclos de longitud 1; es decir, para cualquier vértice u del grafo $G(u,u)=\text{False}$.

Algoritmo propuesto:

```

procedure cuadrado (G: in grafo; G2: out grafo) is
begin
  Incializar_con_False(G2);
  for u in Rango_de_Vértices (G) loop
    for v in Rango_de_Vértices (G) loop
      if G(u,v) then
        for w in Rango_de_Vértices (G) loop
          if G(v,w) then G2(u,w) := True; end if;
        end loop;
      end if;
    end loop;
  end loop;
end;

```

Análisis del orden:

Es evidente que el orden es cúbico en el número de vértices de G , que es el mismo que el de G^2 .

- lista de adyacencias

Algoritmo propuesto:

```

procedure cuadrado (G: in grafo; G2: out grafo) is
begin
  Incializar_con_sublistas_vacias_el_vector_de_adyacencias(G2);
  for u in Rango_de_Vértices (G) loop
    (1) V := Vertices_Adyacentes(G,u);
    while not Es_Vacio(V) loop
      (1) v1 := Primer_Vertice_en_Lista(V); Resto_Lista(V);
      (1) W := Vertices_Adyacentes(G,v1);
      (1) Concat_la_Lista_a_los_Adyacentes_del_vertice(G2,u,W);
      (2) Eliminar_Repetidos_De_Adyacentes(G2,u);
    end loop;
  end loop;
end;

```

Análisis del orden: G tiene n vértices y a arcos.

Bucle de inicialización: $O(n)$, ya que la creación de una lista vacía precisa de tiempo constante.

Primer bucle: En el caso peor para cada nodo u de G tenemos $(n-1)$ vértices adyacentes.

- suponemos que se puede hacer en tiempo constante a el acceso a los adyacentes a un vértice y la concatenación de dos listas. Ref. (1) en el programa
- hay que eliminar vértices repetidos. Puede haber n vértices distintos a lo sumo, y en cada iteración del while, sólo puede tener dicha lista longitud $2n$. Sabiendo esto, se puede eliminar las repeticiones en tiempo $O(n)$, usando un vector ESTA de tamaño n donde registramos lo siguiente:

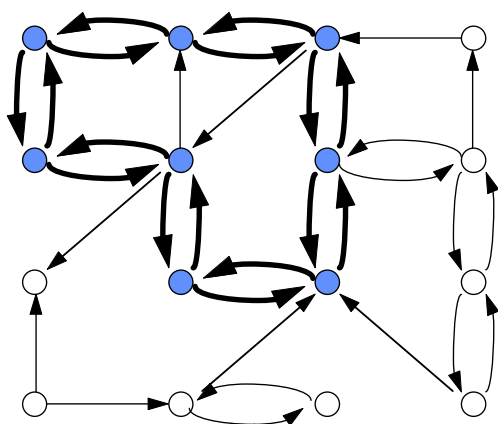
ESTA(i) = true \Leftrightarrow el nodo i está en la lista actual de adyacentes de u en G_2 .

Ref. (2) del programa

Luego, el primer bucle requiere: $O(n * n * (a + bn)) = O(n^3)$

Concluimos pues que el orden de dicha solución es $O(n^3)$ en el peor caso.

61. Dado un grafo dirigido, llamamos **anillo** a una serie de tres o más nodos conectados en forma de ciclo en los dos sentidos. Por ejemplo, en el grafo dirigido de la figura



se ha destacado el único anillo que contiene. Inspirándote en el método del recorrido en profundidad, diseña y analiza un algoritmo eficiente que indique si un grafo dirigido contiene o no al menos un anillo.

✍ :

Observaciones:

- Nos interesa que el grafo venga dado mediante matriz de adyacencia para determinar en tiempo constante si existe doble-arco entre dos vértices (esto es, dos arcos en direcciones opuestas)
- Nos basamos en el recorrido en profundidad. Es preciso controlar:
 - si un vértice ha sido ya visitado o no (para ello se empleará una tabla que llamaremos Marca)
 - estando en un vértice ya visitado, desde dónde habíamos llegado hasta él (para ello se empleará una tabla que llamaremos Padre), con vistas a avanzar en profundidad

y no ciclar en el recorrido entre dos vértices conectados mediante un doble-arco. Hay que evitar esto por varios motivos:

- (a) por riesgo a ciclar indefinidamente en entre dos vértices: $v \rightarrow w \rightarrow v \rightarrow w \dots$
- (b) por riesgo a concluir que existe anillo, cuando realmente sólo se ha comprobado: $v \leftrightarrow w$

Adaptación del recorrido en profundidad sobre grafos:

```

procedimiento EXISTE_ANILLO (G: Grafo; Hay_anillo: Boolean)
  Marca, Padre: array (RANGO(G));
  procedimiento Marcar_Prof (v: Vértice)
    empezar
      Marca(v) := Cierto;
      para cada w ∈ ADYACENTES_A(v) hacer      -- equivale a que existe  $v \rightarrow w$ 
        si v ES_ADYACENTE_A w;                  -- equivale a "existe?"  $v \leftarrow w$ 
          entonces
            si Marca(w)=Falso entonces
              Padre(w) :=v; MARCAR_PROF(w);
            sino -- w está marcado
              si Padre(v)≠w      -- ¡HAY ANILLO!
                entonces Hay_Anillo:= Cierto; Salir;
              fin si;
            fin si;
          fin si;
        fin para;
      fin procedimiento;

  empezar
    Hay_Anillo:=Falso;      -- aún no se ha encontrado ningún anillo
    para cada v ∈ VÉRTICES(G) hacer
      Marca(v) := Falso; Padre(v) := ""
    fin para;
    para cada v ∈ VÉRTICES(G) hacer
      si Hay_Anillo
        entonces Salir;
      sino si Marca(v)= Falso; entonces MARCAR_PROF(v);
      fin si;
    fin para;
  fin procedimiento;

```

Análisis del coste temporal de la solución propuesta:

- Suponemos que el grafo tiene n vértices y a arcos.
 - En el caso peor se tratan todos los vértices y todos los arcos. Cada fila de la matriz de adyacencia sólo se recorre una vez y tan sólo a veces habrá que efectuar algún tratamiento de $O(1)$ (el tratamiento recursivo es una forma de controlar las operaciones, pero no modifica el número de las mismas).
- Así pues, el orden de la solución propuesta es $O(n^2)$.

62. Recuerda que $\text{Fib}(0) = 0$, $\text{Fib}(1) = 1$ y $\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$ para $n \geq 2$.

1. Supón que el coste de sumar, restar y multiplicar dos números es $O(1)$, independientemente del tamaño de los números.
Escribe y analiza un algoritmo que calcule $\text{Fib}(n)$ en tiempo $O(n)$.
2. Escribe y analiza otro algoritmo que calcule $\text{Fib}(n)$ en tiempo $O(\lg n)$ usando suma y multiplicación.

(Ayuda: Considera la siguiente matriz y sus potencias: $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix} = \begin{pmatrix} \text{fib}_0 & 1 \\ \text{fib}_1 & 1 \end{pmatrix}$)

3. Supón ahora que para sumar dos números de β bits se necesita un tiempo en $O(\beta)$ y que para multiplicarlos necesitamos tiempo en $O(\beta^2)$.

Analiza tu algoritmo del apartado (4.1) tomando el número de bits como tamaño de la entrada y considerando el correspondiente coste de las operaciones aritméticas.

Recuerda que $\text{Fib}(n) \approx \left(\frac{1+\sqrt{5}}{2} \right)^n$.

✎:

62.1)

```
function Fibonacci62.1 (N: in Integer) return Integer is
  K, Fa, Fb, Aux: Integer;
begin
  if N=0 or N=1 then      return N;
  else Fa:=0; Fb:=1;
    for K in 2..N loop
      Aux:= Fa + Fb;      Fa:= Fb;      Fb:= Aux;
    end loop;
    return Fb;
  end if;
end;
```

La solución propuesta es evidentemente lineal en el valor de la entrada, n: $O(n)$

62.2) Proponemos una solución sencilla basada en cálculo de operaciones (sin repetición) de las potencias de una cierta base, la cual, en este caso, es la matriz sugerida:

```
function Fibonacci62.2 (N: in Integer) return Integer is
  Sol, M: MatCuadra2:=  $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ ;

  function potenciaFib (F: in MatCuadra; Y: in Exponente)
    return MatCuadra is
    Aux: MatCuadra;
  begin
    if Y=1 then return F;
    else Aux:= potenciaFib (F, Y div 2);
      Aux:= Producto_MatCuadrada (Aux,Aux);
      if Impar(Y)
        then return Producto_MatCuadrada (Aux, M);
        else return Aux;
      end if;
    end if;
  end;
begin
  if N=0 then return 0;
  else Sol:= PotenciaMat( $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ , N);
    return Sol(2,1);
  end if;
end;
```

Análisis del orden temporal:

$$T_{\text{potencia}}(y) = a + T_{\text{potencia}}(y/2) \in O(\lg y)$$

$$T_{\text{fib}}(n) = a + T_{\text{potencia}}(n) \in O(\lg n)$$

62.3) Sabemos que:

- $\text{Fib}(n) \approx \left(\frac{1+\sqrt{5}}{2}\right)^n$. Si A es un número: $\lceil \lg_2 A \rceil$ es el número de bits del valor A y $\lceil \log A \rceil$ es el número de dígitos de A. Luego, $\text{Fib}(n)$ tiene $\left\lceil \lg_2 \left(\frac{1+\sqrt{5}}{2}\right)^n \right\rceil$ bits, esto es, $n \left\lceil \lg_2 \left(\frac{1+\sqrt{5}}{2}\right) \right\rceil$ bits o en notación asintótica $O(n)$.

Entonces, en el algoritmo de 62.1, al sumar los **valores** $\text{Fib}(i)$ $i=0..n$, el orden con respecto al número de bits de los valores responde a la suma

$$1+2+\dots+i+\dots+n \in O(n^2).$$

Observa que si el tamaño en número de bits de una entrada es β entonces su **valor** es de $O(2^\beta)$.

De modo que si el **valor** de la entrada (que es n) tiene tamaño β , el orden -en función de ese tamaño- es

$$O(2^\beta * 2^\beta) = O(4^\beta).$$

- 63.** Dada una cantidad M ¿cuál es el máximo número X, con $X \leq M$, que podemos calcular a partir de una cantidad inicial C y aplicando repetidamente las operaciones $*2$, $*3$, $*5$? Emplea la técnica de la programación dinámica para determinar dicho número X y analiza el orden del algoritmo propuesto.

 :

Para darle solución un problema con la técnica de la programación dinámica los pasos a seguir son siempre:

- Definición de la ecuación de recurrencia que da solución al problema
- Definición de la estructura que almacenará las soluciones intermedias. Determinar claramente qué almacena cada posición de la estructura, cómo se inicializará, en qué orden ha de completarse y como a partir de ella se obtiene el resultado del problema original
- Algoritmo recursivo o iterativo que implementa lo expresado por la ecuación del punto 1, recogiendo los resultado parciales sobre la estructura definida en 2 y evitando a toda costa repetir cálculos realizados con anterioridad.
- Análisis del orden de la solución propuesta.

Una solución posible a nuestro problema es la siguiente es:

- a)** Supongamos $V, L \geq 0$ y $L \geq V$

$$\begin{aligned} \text{tanteo}(V, L) &= \max \{ \text{tanteo}(2*V, L), \text{tanteo}(3*V, L), \text{tanteo}(5*V, L) \} && \text{si } L \geq 5*V \\ &= \max \{ \text{tanteo}(2*V, L), \text{tanteo}(3*V, L) \} && \text{si } 5*V > L \geq 3*V \\ &= \text{tanteo}(2*V, L) && \text{si } 3*V > L \geq 2*V \\ \text{tanteo}(V, L) &= V && \text{si } 2*V > L \end{aligned}$$

- b)** Los resultado intermedios los almacenaremos en una tabla unidimensional:

TApuestas

C		i		M

donde

TApuestas(i): es el valor más cercano y no superior a M que se pueda obtener partiendo de i y tan sólo aplicándole productos *2, *3, *5

Inicialización de la estructura:

- por la segunda ecuación de tanteo, sabemos $\forall j (M \geq j \geq \lfloor M/2 \rfloor + 1 \rightarrow \text{tanteo}(j, M) = j)$, luego para esos valores j: TApuestas(j)=j
- el resto se inicializa a -1, que indicará que aún el valor no se ha calculado

El resultado: TApuestas(C)

c)

```
procedure Apuestas (C,M: in integer; X: out Integer) is
  TApuestas: array (C..M) of Integer;
  function Tanteo (V,L) is
  begin
    if TApuestas(V)=-1 and 2*V>L then
      then TApuestas(V) := V; return TApuestas(V);
    else if 5*V<=L
      then if TApuestas(5*V)=-1
        then t5:= Tanteo(5*V,L);
        else t5:= TApuestas(5*V);
        end if;
      else t5:=0;
      end if;

      if 3*V<=L
      then if TApuestas(3*V)=-1
        then t3:= Tanteo(3*V,L);
        else t3:= TApuestas(3*V);
        end if;
      else t3:=0;
      end if;

      if 2*V<=L
      then if TApuestas(2*V)=-1
        then t2:= Tanteo(2*V,L);
        else t2:= TApuestas(2*V);
        end if;
      else t2:=0;
      end if;

      TApuestas(V) := max (t2,t3,t5);
      return TApuestas(V);
    end if;

  begin
    for J in C .. (M div 2) loop TApuestas(J) := -1; end loop;

    for J in (M div 2)+1.. M loop TApuestas(J) := J; end loop;

    X:= Tanteo(C,M);

  end;
```

d) Análisis del orden:

El número de celdas a rellenar son M-C+1

- La inicialización rellena una vez y en tiempo constante cada una de las celdas. En total, $O(M-C+1)$
- Sólo se hacen llamadas recursivas cuando el subproblema aún no ha sido calculado (no se hacen llamadas inútiles. Se recalculan a lo sumo M/2 celdas (aquellas inicializadas con valor -1). Para calcular el valor a almacenar en cada una de éstas celdas se ejecutan

3 sentencias if en tiempo constante, se obtiene el máximo de tres valores y se almacena y devuelve dicho valor. Esto es, a los sumo el valor de $M/2$ celdas se recalcula nuevamente en tiempo constante cada una de ellas. Esto requiere $O(M)$

Concluimos añadiendo que la solución propuesta tiene orden $O(M)$.

64.

1. Justifica que el algoritmo típico que usas para multiplicar dos números enteros A y B realiza $O(m \cdot n)$ multiplicaciones *elementales* y $O(m \cdot n)$ sumas *elementales*, siendo m y n el número de dígitos de A y B respectivamente. (Llamamos *elemental* a la multiplicación (resp. suma) de dos dígitos decimales.)
2. En realidad, las operaciones $A \times 10$, $\lfloor B/10 \rfloor$ y $B \bmod 10$ pueden considerarse de $O(1)$ (consiste simplemente en añadir, eliminar o seleccionar una cifra decimal). Por tanto para realizarlas no necesitamos multiplicaciones ni sumas. Analiza el número de multiplicaciones elementales (resp. el número de sumas elementales) que se realizan con el siguiente algoritmo de multiplicación:

```
función Multiplicar (A,B)
    si B=0 entonces resultado 0
    si no resultado A*(B mod 10) + Multiplicar (Ax10,
     $\lfloor B/10 \rfloor$ )
```

donde + y * son las operaciones de suma y multiplicación típicas consideradas en el apartado (4.1). Fíjate que las operaciones “ $\times 10$ ”, “*”, y “Multiplicar” son tres operaciones diferentes.

✍ :

1. supongamos $m \geq n$ (el otro caso es análogo)

$$\begin{array}{rcccccc}
 A_m & A_{m-1} & \dots & A_2 & A_1 & \\
 & B_n & \dots & B_2 & B_1 & \\
 \hline
 & A_m \cdot B_1 \dots & & A_2 B_1 & A_1 B_1 & \\
 A_m \cdot B_2 \dots & & A_2 B_2 & A_1 B_2 & & \\
 \\
 A_m \cdot B_n \dots & & A_2 B_n & A_1 B_n & & \\
 \hline
 \end{array}$$

n^a máximo de multiplicaciones:

- una multiplicación entre dos dígitos lleva tiempo constante.
 - para obtener cada fila se precisan hacer m multiplicaciones.
 - hay n filas
- el n^a de multiplicaciones a realizar precisa tiempo $O(n \cdot m)$

n^a máximo de sumas:

- las sumas se hacen por columnas.

- en cada columna a lo sumo hay n dígitos, uno por cada fila (+1 si hubiera llevada de la columna anterior). Sumar todos ellos requiere $O(n)$, considerando que la suma de dos dígitos tiene orden $O(1)$.
- a lo sumo hay $O(n+m)$ columnas (orden del número de dígitos del valor resultante) el n^a de sumas a realizar precisa tiempo $O((n+m)*n)=O(n^2+m*n)=O(m*n)$

2. $NM(m,n)=n^o$ de multiplicaciones que se realizan cuando el primer número tiene m dígitos y el segundo n .

$$NM(m,n) = m + NM(m+1, n-1) = m + NM(m+1, n-1) \quad \text{si } n > 1$$

$$NM(m,1) = m$$

Cálculo del orden:

$$\begin{aligned}
 NM(m,n) &= m + NM(m+1, n-1) = m + (m+1) + NM(m+2, n-2) \\
 &= m + (m+1) + (m+2) + NM(m+3, n-3) = \dots \\
 &= m + (m+1) + (m+2) + \dots + (m+(n-2)) + NM(m+(n-1), n-(n-1)) \\
 &= m + (m+1) + \dots + (m+(n-2)) + NM(m+n-1, 1) \\
 &= m + (m+1) + (m+2) + \dots + (m+(n-2)) + m \\
 &= n*m + (1+2+3+\dots+n-2) \in O(n*m) + O(n^2) =_{(\sup m > n)} \mathbf{O(n*m)}
 \end{aligned}$$

La multiplicación de A y B con $|A|=m$ y $|B|=n$ dígitos nos puede dar un n^o con $O(m+n)$ dígitos.

$NS(m,n)=n^o$ de sumas que se realizan cuando el primer número tiene m dígitos y el segundo n .

No nos interesa el número exacto de sumas, pero sí el orden.

$$NS(m,n) = O(m+n) + NM(m+1, n-1) =_{(\sup m > n)} m + NM(m+1, n-1) \quad \text{si } n > 1$$

$$NS(m,1) = m+1$$

No es preciso repetir los cálculos para determinar el orden del número de sumas realizadas ya que la ecuación simplificada obtenida es igual a la del subapartado anterior. Así pues, el orden de sumas que realizará dicha solución es **$O(n*m)$** .

65. Supongamos n clases de monedas. La moneda de clase i (para cada $i=1..n$) es de valor v_i y tenemos exactamente c_i monedas de esa clase. Escribe y analiza un algoritmo que determine de cuántas formas **distintas** podemos sumar un valor M con esas monedas.

Ej.: Si las clases de monedas fueran dos con las siguientes cantidades:

$$v_1 = 25, c_1 = 10; \quad v_2 = 100, c_2 = 2$$

Valor	Formas distintas
250	3 formas: $2*100 + 2*25$, $1*100 + 6*25$, $10*25$
300	2 formas: $2*100 + 4*25$, $1*100 + 8*25$

La combinación $25+100$ es la misma que $100+25$, y por ello es una única a considerar.

✍ :

Se opta resolver el problema mediante la técnica de programación dinámica:

Para darle solución a un problema con dicha técnica los pasos a seguir son:

1. Definición de la ecuación de recurrencia que da solución al problema.
2. Definición de la estructura que almacenará las soluciones intermedias. Determinar claramente qué almacena cada posición de la estructura, cómo se inicializará, en qué orden ha de completarse y como a partir de ella se obtiene el resultado del problema original.
3. Algoritmo recursivo o iterativo que implementa lo expresado por la ecuación del punto 1, recogiendo los resultados parciales sobre la estructura definida en 2 y evitando a toda costa repetir cálculos realizados con anterioridad.
4. Análisis del orden de la solución propuesta.

Una solución posible a nuestro problema es la siguiente:

1. Suponemos que de la clase i de monedas $V(i)$ es el valor y que exactamente tenemos $C(i)$ monedas. Sea

CombiRes(j,i): el número de combinaciones distintas existentes para sumar el valor j empleando las clases $[1..i]$ de monedas, sabiendo que:

$$\forall p (i \geq p \geq 1 \rightarrow V(p) = \text{valor numérico de la moneda de la clase } p \wedge C(p) = \text{cantidad de monedas disponibles de la clase } p)$$

$$\text{CombiRes}(0,i) = 1$$

-- La combinación empleada para devolver la cantidad 0 es no emplear moneda alguna

$$\text{CombiRes}(B,0) = 0 \quad \text{-- El valor } B \text{ no es descomponible}$$

$$\text{CombiRes}(B,i) = \text{CombiRes}(B,i-1) + \text{CombiRes}(B-V(i),i-1) + \text{CombiRes}(B-2V(i),i-1) + \dots + \text{CombiRes}(B-j V(i),i-1) \quad \text{con } j = \min\{C(i), \lfloor B/V(i) \rfloor\}$$

2. Los resultados intermedios los almacenaremos en una matriz bidimensional:

Comb

	0	1	...	j	...	M
0	1	0	0	0	0	0
1	1					
...	1					
i	1					
...	1					
n	1					

donde $\text{Comb}(j,i) = \text{CombiRes}(j,i)$

Inicialización de la estructura:

- por la primera ecuación de recurrencia: $\forall i (n \geq i \geq 0 \rightarrow \text{Comb}(0,i)=1)$
- por la segunda ecuación de recurrencia: $\forall j (M \geq j \geq 1 \rightarrow \text{Comb}(j,0)=0)$
- el resto se inicializa a -1, que indicará que aún el valor no se ha calculado

El resultado en: $\text{Comb}(M,n)$:

Completado: mediante llamadas recursivas efectivas (= que calculan el valor por primera y única vez). La primera llamada provocará el completado de la matriz y en concreto el de la casilla solución a devolver.

3.

```

procedure CombinacionesDistintas (M: in Integer;
                                   V,C: in Tabla(1..N))
                                   return Integer is

  Comb: array (0..M,0..N) of Integer:=(others=>(others=>-1));

  procedure Combinaciones (Val: in Integer; K: in Integer) is
  begin
    -- Comb(Val, K)=-1
    if Comb(Val, K-1)=-1 then Combinaciones(Val, K-1); end if;
    AcumulaCombDistintas:= Combinaciones(Val, K-1);

    for Veces in 1.. Mínimo{C(K), M div V(K)} loop
      if Comb(Val - Veces*V(K), K-1)=-1
      then Combinaciones(Val - Veces*V(K), K-1);
      end if;
      AcumulaCombDistintas:= AcumulaCombDistintas
                           + Comb(Val - Veces*V(K), K-1);
    end loop;

    Comb(Val, K):= AcumulaCombDistintas;
  end;

begin
  for I in 0 .. N loop Comb(0,I):=1; end loop;
  for J in 1 .. M loop Comb(J,1):=0; end loop;
  if Comb(M,N)=-1 then Combinaciones(M, N);
  return Comb(M,N);

end;

```

4. Análisis del coste temporal:

La matriz Comb con M*n celdas:

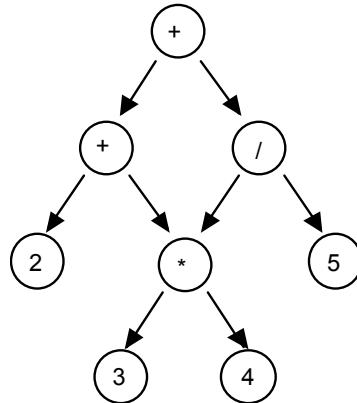
- Se inicializa cada celda en tiempo constante, O(1)
- El valor de O(M*n) celdas se calcula una vez más, según la ecuación tercera de la definición recursiva. El coste temporal para completar la celda Comb(j,i) depende de la cantidad j, del valor V(i) de las monedas de la clase i y de la cantidad de monedas C(i) que se disponen de dicha clase. Así pues, en el caso peor, cuando j es máximo (M) el número mayor de veces que se repetirá el bucle con índice Veces se efectuará

$$\begin{aligned}
 & \max \left\{ \min \left(\left\lfloor \frac{\text{Val}}{V(1)} \right\rfloor, C(1) \right), \min \left(\left\lfloor \frac{\text{Val}}{V(2)} \right\rfloor, C(2) \right), \dots, \min \left(\left\lfloor \frac{\text{Val}}{V(n)} \right\rfloor, C(n) \right) \right\} \\
 & = \max_{1 \leq i \leq n} \left\{ \min \left(\left\lfloor \frac{\text{Val}}{V(i)} \right\rfloor, C(i) \right) \right\}
 \end{aligned}$$

Concluimos que el orden temporal de la solución propuesta es:

$$O \left(M * n * \max_{1 \leq i \leq n} \left\{ \min \left(\left\lfloor \frac{\text{Val}}{V(i)} \right\rfloor, C(i) \right) \right\} \right)$$

66. Una expresión aritmética puede representarse mediante un DAG (gafo dirigido acíclico). Esta representación es más compacta que la de árbol, porque las subexpresiones repetidas sólo tienen que aparecer una vez. Por ejemplo, la expresión $2+3*4+(3*4)/5$ podría representarse mediante el DAG de la figura. Cada operador (o nodo interno) tiene exactamente dos operandos (o arcos). Escribe y analiza un algoritmo lineal en el número de nodos que evalúe la expresión aritmética dada por un DAG de estas características.



✏️ :

Supongamos que los nodos (internos o no) están numerados.

¿Es conocido el índice o numeración del nodo raíz?

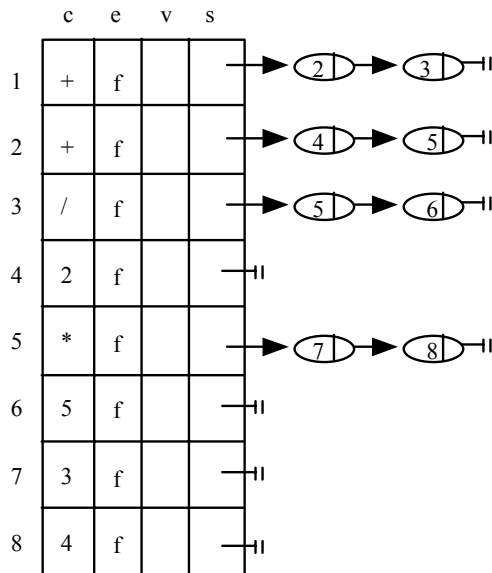
Supondremos que nos lo da la función Raíz(G).

Idea del proceso general a seguir para calcular el valor de un dag:

- si el vértice a examinar ya ha sido evaluado, devolvemos dicho valor
- si el vértice aún no ha sido evaluado su valor se obtiene de la siguiente forma
 - si el vértice es hoja: devolvemos su valor numérico
 - si el vértice no es hoja: evaluamos, si son precisos, sus subgrafos izquierdo y derecho y operamos dichos valores con el operador almacenado en el vértice

Es decir, lo que realizamos es un recorrido en profundidad en un grafo que de partida sabemos quién es su raíz y que cada nodo interno tiene exactamente dos subgrafos adyacentes, denominados izquierdo y derecho. Es imprescindible llevar constancia de si un subgrafo ha sido o no ya evaluado (=marcado) para evitar repetir cálculos.

Optamos por una representación del siguiente estilo:



Cada vértice o nodo i del grafo tiene cuatro componentes:

- c: contenido del vértice i : valor numérico u operador binario aritmético.
- e: indicará si el subgrafo con raíz el nodo i ha sido (t) o no (f) evaluado.
- v: valor del subgrafo con raíz el nodo i ; campo con valor asociado siempre y cuando el nodo i haya sido previamente evaluado.

s: si el nodo i es hoja entonces la lista vacía, si no, la lista de sus dos subgrafos adyacentes (siguientes), primero representará al operando izquierdo y el segundo al operando derecho del operador aritmético que está en el campo c de i .

Algoritmo:

```

begin
  Inicializar_campos_E_a_no_evaluados(G);
  evaluar_nodo(G,Raiz(G));           -- Raiz(G): da el índice del nodo raíz de G
  return devolver_valor_del_subgrafo_con_raiz(G,Raiz(G));
                                     -- el campo v del nodo r
end;

evaluar_nodo(G,k)
if Es_numérico(G,k)                -- Es una hoja
then   Almacenar_valor_en_el_campo_v(G,k,Campo_C(G,k))
else   AI:=Adyacente_Izquierdo(G,k); -- índice del nodo-izquierdo del nodo k

      AD:= Adyacente_Derecho(G,k);
      if No_Evaluado(G, AI) then evaluar_nodo(G, AI); end if;
      if No_Evaluado(G, AD) then evaluar_nodo(G, AD); end if;
      Res:= operar(Campo_C(G,k),
                   devolver_valor_del_subgrafo_con_raiz(G,AI),
                   devolver_valor_del_subgrafo_con_raiz(G,AD));
      Almacenar_valor_en_el_campo_v(G,k,Res);

  end if;
  Marcar_Evaluado(G,k)              -- almacenar a true en el campo E

```

Análisis del orden:

G tiene n vértices y a arcos, pero por las características de nuestros dag, $a < 2n$

- Raiz(G): Si es conocido se precisa tiempo constante. Si no es conocido hay que recorrer cada uno de los nodos y cada uno de sus arcos una sola vez para calcular el indegree del grafo y devolver aquel nodo que no tenga ningún arco entrante. Es conocido que este proceso se puede hacer en tiempo $O(n+a)$, luego en nuestro caso $O(n)$.
- Se inicializan los n nodos a no evaluados en tiempo constante cada uno, luego: $O(n)$
- Cada nodo se evalúa una sola vez operando de la siguiente forma:
 - si es hoja: acceso a un campo, obtener valor y almacenarlo en otro campo del nodo: $O(1)$
 - si es nodo interno: se obtienen los valores de los subgrafos izquierdo y derecho, se obtiene el operador aritmético y se opera. El resultado se almacena en otro campo del nodo. Todo ello se realiza en $O(1)$
 - La evaluación total de todos los nodos precisa tiempo $O(n)$.
 - No obstante, un nodo podría ser visitado más de una vez; esto es, tantas veces como predecesores (arcos entrantes) tenga (de hecho, tantas veces como se repita la subexpresión aritmética que representa en la expresión aritmética total). Pero cada arco se trata también una sola vez (en tiempo constante) y estos son menos de $2n$.

Concluimos pues, que la evaluación de dags propuesta requiere $O(n+a)=O(n+2n)=O(n)$