Asignatura: Programación III
Curso 2º de Ingeniería Técnica Informática (Sistemas).
Pedro Pérez Ostiz
Tudela.- Septiembre 2002.

Programación III

Índice

Estos apuntes han sido realizados para el estudio de la asignatura PROGRAMACIÓN III del curso 2º de Ingeniería Técnica informática en la rama de Sistemas de la UNED.
He tomado base de los libros de texto recomendados por la UNED:

- ***Fundamentos de Algoritmia**. G. Brassard. - P. Bratley*
- ***Esquemas algorítmicos**. Julio Gonzalo Arroyo. - Miguel Rodríguez Artacho*

Tudela septiembre 2002.
Pedro Pérez Ostiz

1 PRELIMINARES

1.1 Introducción.

El objeto de estudio son los **algoritmos** y la **algoritmia**.

1.2 ¿Qué es un algoritmo?.

El nombre proviene del matemático persa del siglo IX *al-Khowârizmî*, y es sencillamente un *conjunto de reglas para efectuar algún cálculo*. Un ejemplo que seguiremos es el algoritmo para la multiplicación, del cual existen varios métodos.

Un algoritmo no debe dejar sitio a ninguna decisión subjetiva, ni a la intuición, ni a la creatividad. Por tanto, diremos por ejemplo, que una receta de cocina es un algoritmo si define claramente las cantidades y el tiempo de cocinado, pero no, si da definiciones vagas como "*salpimentar a gusto*", o "*guisar hasta que esté medio hecho*".

Al aplicar un algoritmo, lo normal es pensar que nos dará la respuesta correcta, pero en ocasiones, ese algoritmo tardaría mucho, o sería difícil de programar, o consumiría muchos recursos de memoria, por lo que en esas ocasiones, podemos preferir un algoritmo que no tenga esos problemas, aunque la respuesta sea aproximada. Por ejemplo, para calcular la raíz cuadrada de 2, no podemos esperar un resultado exacto, ya que tiene infinitos decimales distintos. Nos conformaremos con un algoritmo que nos dé el resultado con una precisión dada.

Además de los **algoritmos aproximados**, tenemos también los **algoritmos heurísticos**, en los que tenemos que fiarnos en nuestra buena suerte de encontrar una respuesta razonablemente válida, y en los que no podemos controlar el error, aunque quizás seamos capaces de estimar su magnitud.

Podemos definir ahora la **Algoritmia**, como *el estudio de los algoritmos*. Para decidir entre los posibles algoritmos que existen para un mismo problema, debemos sopesar los límites del equipo, el tiempo de que disponemos, el más fácil de programar...etc.

> La Algoritmia es por tanto la ciencia que nos permite evaluar el efecto de los distintos factores externos sobre los algoritmos disponibles, para poder seleccionar el que más se ajusta a nuestras circunstancias; también es la ciencia que nos indica la forma de diseñar un nuevo algoritmo para una tarea concreta.

Tomando como ejemplo la aritmética elemental, tenemos que multiplicar dos enteros positivos. El algoritmo que todos conocemos es el **algoritmo clásico**: *multiplicar sucesivamente el multiplicando por cada una de las cifras del multiplicador, tomadas de derecha a izquierda, y escribir esos resultados desplazando cada línea un lugar a la izquierda. Por último sumar estas filas para obtener la respuesta.*

```
   981
 x 1234
------
  3924
 2943
1962
 981
------
1210554
```

Algoritmo clásico de multiplicación

Pero existen otros como por ejemplo la **multiplicación à la russe**: *Se escribe el multiplicando y el multiplicador uno junto a otro. Se hacen dos columnas, una debajo de cada operando, repitiendo la regla siguiente hasta que el número de la columna izquierda sea un 1: se divide el número de la izquierda por 2 ignorando restos y se duplica el de la derecha. Por último, se suman los números de la columna derecha, cuyo compañero de la columna izquierda sea impar.*

981	1234
490	~~2468~~
245	4936
122	~~9872~~
61	19744
30	~~39488~~
15	78976
7	157952
3	315904
1	631808
	1210554

Multiplicación à la russe

Incluso existiría otro algoritmo mediante la técnica del **divide y vencerás**, y muchos otros más, cada uno con sus ventajas e inconvenientes.

1.3. Notación para los programas

Si intentamos explicar los programas en español, veremos que no es la manera adecuada, por lo que usaremos un pseudolenguaje de programación, que iremos viendo según aparezcan nuevos algoritmos.

Como ejemplo, escribiremos el algoritmo de la multiplicación à la russe:

```
función rusa(m,n)
    resultado ← 0
    repetir
        si m es impar entonces resultado ← resultado + n
        m ← m ÷ 2
        n ← n + n
    hasta que m = 1
    devolver resultado
```

1.4 Notación matemática.

La notación matemática que se va a utilizar, es la que habitualmente se utiliza para:

- Cálculo proposicional.
- Teoría de conjuntos.
- Enteros, reales e intervalos.
- Funciones y relaciones
- Cuantificadores
- Sumas y productos

1.5 Técnica de demostración 1: CONTRADICCIÓN.

La *demostración por contradicción*, que también se conoce como *prueba indirecta*, consiste en demostrar la veracidad de una sentencia demostrando que su negación da lugar a una contradicción.

Por ejemplo, para demostrar la sentencia "*existen infinitos número primos*", comenzaremos por suponer que existe un número finito de primos, y si terminamos en una afirmación que es evidentemente falsa (una contradicción), podemos deducir que la primera sentencia es verdadera.

1.6 Técnica de demostración 2: INDUCCIÓN MATEMÁTICA.

En la ciencia, hay dos enfoques opuestos fundamentales: **inducción** y **deducción**. La inducción consiste en *inferir una ley general a partir de casos particulares*, mientras que una deducción es una *inferencia de lo general a lo particular*. La deducción siempre es válida con tal de que sea aplicada correctamente. Sin embargo, aunque la inducción puede dar lugar a conclusiones falsas, no se puede despreciar este método.

En general, no se puede confiar en el resultado de la inducción, mientras queden casos particulares por comprobar. Por ejemplo, con el polinomio $p(n) = n^2 + n + 41$, obtenemos la serie de números 41, 43, 47, 53, 61, 71, 83, 97,113, 131, 151... de lo que es natural inferir que todos son primos. Sin embargo, $p(40) = 1.681$, es 41^2, es decir, un número compuesto.

Por contraste, el razonamiento deductivo no está sometido a errores de este tipo. Siempre y cuando la regla invocada sea correcta, y sea aplicable a la situación que se estudie, la conclusión que se alcanza es necesariamente correcta.

Entonces, *¿porqué se utiliza la inducción si es proclive a errores?.* Existen casos en los que es el único método. Por ejemplo, para determinar las leyes fundamentales que rigen el Universo, es preciso utilizar un enfoque inductivo a partir de datos obtenidos en experimentos. *Halley* predijo el retorno de su cometa por razonamiento inductivo y *Mendeleev* predijo no sólo la existencia de elementos químicos no descubiertos, sino también sus propiedades químicas.

Una de las técnicas *deductivas* más útiles que están disponibles en matemáticas tiene la mala fortuna de llamarse **inducción matemática**. Esta terminología resulta confusa, pero tenemos que vivir con ella.

El principio de la inducción matemática.

Vamos a considerar el siguiente algoritmo:

```
función cuadrado(n)
  si n=0 entonces devolver 0
  si no devolver 2n+cuadrado(n-1)-1
```

Haciendo pruebas con varios números, vemos que funciona: *cuadrado(0)=0, cuadrado(1)=1, cuadrado(2)=4*...etc. Pero cómo podemos asegurar que el algoritmo funciona para cada $n \geq 0$.

El **principio de inducción matemática** nos permite inferir esto. Este principio dice que la propiedad P es cierta para todo $n \geq a$ si:

1.- *P(a)* es cierto
2.- *P(n)* debe ser cierto siempre que *P(n-1)* sea válido para todos los enteros $n>a$.

Además de servirnos para demostrar la corrección de un algoritmo, el principio de inducción matemática, puede servirnos en ocasiones para crear algoritmos. Por ejemplo, en el *problema de embaldosado* que se describe en la página 23 del libro de texto.

1.7 Recordatorios.

Ver el libro, páginas 35 a 56, en el que se hace un recordatorio de algunos resultados elementales acerca de **límites**, **sumas de series** sencillas, **combinatoria** y **probabilidad**.

2 ALGORITMIA ELEMENTAL

2.1 Introducción.

Veremos como un ***problema*** (por ejemplo multiplicar dos enteros), tiene muchos, normalmente infinitos, ***ejemplares*** (por ejemplo multiplicar 981 x 1234). Los algoritmos deben funcionar correctamente en *todos* los casos del problema que afirman resolver.

Veremos qué se entiende por ***eficiencia***, y cómo cambia a medida que los casos del problema se vuelven mayores y por tanto (normalmente) más difíciles de resolver. Veremos por tanto también, la eficiencia en el peor caso posible.

2.2 Problemas y ejemplares.

En el capítulo anterior, veíamos diferentes formas de multiplicar 981 por 1234, que podemos expresar por (981,1234). De todas formas, los algoritmos vistos eran una forma *general* de multiplicar dos enteros positivos, por lo que podrían servir para el caso (5241,73) por ejemplo. Sin embargo, no servirían para el caso (-12,83.7), ya que el primero no es positivo, y el segundo no es entero. En este caso necesitaríamos un algoritmo *más general*, que nos permita multiplicar dos números cualquiera.

Los algoritmos deben funcionar para todos los casos que manifiestan resolver. De la misma forma que un teorema no es válido si encontramos un solo contraejemplo, un algoritmo puede rechazarse tomando como base un único resultado incorrecto. En ocasiones es difícil demostrar la corrección de un algoritmo. Para hacerlo posible, es importante definir su ***dominio de definición***, es decir, el conjunto de casos que deben considerarse. Hemos visto que los algoritmos de multiplicación vistos no funcionan para números negativos o fraccionarios. Sin embargo, esto no significa que no sean válidos, sino que esos números no están dentro de su *dominio de definición*.

Además de las limitaciones del dominio de definición de los algoritmos, también se debe limitar el tamaño del problema debido al dispositivo de cálculo utilizado, como consecuencia de que los números sean muy grandes, o de que agotemos la memoria del sistema... Sin embargo, esta limitación no debe achacarse al algoritmo que decidamos utilizar.

2.3 La eficiencia de los algoritmos.

Nos planteamos la pregunta de qué algoritmo elegir para resolver un problema. Si solamente debemos resolver unos pocos casos, quizás nos decidamos por el que sea más fácil de programar, o por alguno que ya esté programado. Pero si debemos aplicarlo a muchos ejemplares, o si el problema es más difícil, quizás debemos cuidar más esa decisión.

Un enfoque ***empírico*** (*a posteriori*) consiste en programar las técnicas e ir probándolas con diferentes casos. Por el contrario, un enfoque ***teórico*** (*a priori*), consiste en determinar matemáticamente la cantidad de recursos necesarios como *función del tamaño de los casos considerados*. Los recursos que más nos interesan son el **tiempo** de computación y el **espacio** de almacenamiento, siendo el primero, normalmente más importante. Diremos por tanto que un algoritmo es más eficiente si tarda menos en ejecutarse, y solamente en algunas ocasiones nos interesará también los requisitos de almacenamiento.

Usaremos la palabra "*tamaño*" para indicar cualquier entero que mida de alguna forma el número de componentes de un ejemplar: Por ejemplo, si hablamos de ordenaciones, será el número de ítems a ordenar; si hablamos de grafos, el número de nodos; si hablamos de operaciones con enteros, sería el valor de esos enteros.

La ventaja de la aproximación teórica, es que no depende de la computadora utilizada, ni del lenguaje de programación, ni de las habilidades del programador.

> El ***principio de invarianza*** dice que dos implementaciones distintas (en computadoras distintas o con distintos lenguajes de programación), no diferirán en su eficiencia más que en una constante multiplicativa.

Es decir, un cambio de computadora, o de lenguaje de programación o de compilador, puede resolver un problema 10 veces más deprisa, o 100 veces, pero siempre en un valor constante. Sin embargo, un cambio de algoritmo, puede conseguir un incremento que se vuelva cada vez más pronunciado a medida que crezca el tamaño de los casos.

Diremos que un algoritmo para algún problema requiere un tiempo ***del orden*** de *t(n)* para una función dada *t*, si existe una constante multiplicativa *c* y una implementación del algoritmo capaz de

resolver todos los casos de tamaño n en un tiempo que no sea superior a $ct(n)$ segundos. Este importante concepto, es conocido como **notación asintótica**.

Diremos que un algoritmo es:

- Algoritmo **lineal**, si requiere un tiempo *lineal*, es decir, del orden n
- Algoritmo **cuadrático**, si requiere un tiempo *cuadrático*, es decir del orden n^2
- Algoritmo **cúbico**, si requiere un tiempo *cúbico*, es decir del orden n^3
- Algoritmo **polinómico**, si requiere un tiempo *polinómico*, es decir del orden n^k
- Algoritmo **exponencial**, si requiere un tiempo *exponencial*, es decir del orden c^n

No hay que olvidar de todas formas las **constantes ocultas**, ya que puede equivocarnos en una elección: Por ejemplo, un algoritmo que tarde n^2 días y otro que tarde n^3 segundos, nos llevaría a elegir desde el punto de vista teórico, al algoritmo cuadrático como asintóticamente mejor. Sin embargo, si el tamaño de nuestros problemas es medio, debiéramos elegir el algoritmo cúbico, ya que solo el cuadrático es mejor para casos que requieran ¡20 millones de años! para resolverlos. Aunque el algoritmo cuadrático es asintóticamente mejor, contiene una constante oculta tan grande que lo hace imposible de utilizar para casos de tamaño normal.

2.4 Análisis de "caso medio" y "caso peor".

El tiempo que requiere un algoritmo o el espacio de almacenamiento necesario, puede variar entre dos ejemplares distintos del mismo tamaño.

Por ejemplo, comparando dos algoritmos de ordenación de matrices, ***inserción*** y ***selección***, tenemos que al entregarles una matriz del mismo tamaño, el tiempo requerido es dependiente de la ordenación de la matriz de entrada. Si originalmente está ordenada inversamente, ambos algoritmos requieren el mayor tiempo, sin embargo, el algoritmo de *selección*, no varía mucho (no más del 15%) su coste en tiempo esté como esté la matriz original. Por el contrario, el algoritmo de *inserción*, requiere un tiempo lineal cuando la matriz está originalmente ordenada, y cuadrático cuando está ordenada inversamente.

```
procedimiento insertar (T[1..n])
    para i ← 2 hasta n hacer
        x ← T[i]; j ← i-1
        mientras j>0 y x<T[j] hacer
            T[j+1] ← T[j]
            j ←j-1
        fmientras
        T[j+1] ← x
```

```
procedimiento seleccionar (T[1..n])
    para i ← 1 hasta n-1 hacer
        minj ← i; minx ← T[i]
        para j ← i+1 hasta n hacer
            si T[j] < minx entonces
                minj ← j
                minx ← T[j]
            fsi
        fpara
    T[minj] ← T[i]
    T[i] ← minx
```

Si pueden producirse variaciones tan grandes, ¿cómo es posible hablar de coste en tiempo para un problema de tamaño n?. Dependerá de los casos. Si por ejemplo estamos hablando de la seguridad de una planta nuclear, nos interesará conocer el comportamiento en el peor caso. Si por el contrario vamos a tener que resolver muchos casos distintos, quizás nos interese el comportamiento en el caso medio. Suele ser más difícil analizar el comportamiento medio que en el caso peor, ya que entonces interviene la probabilidad que tienen los diferentes casos en presentarse como ejemplares. Un análisis útil del comportamiento medio del algoritmo requiere por tanto un conocimiento *a priori* acerca de la distribución de los casos que hay que resolver, y eso suele ser poco realista.

En lo que sigue, a no ser que se diga lo contrario, nos referiremos siempre al comportamiento en el **caso peor**.

2.5 ¿Qué es una operación elemental?

Una *operación elemental* es aquella cuyo tiempo de ejecución puede acotarse superiormente por una constante que solo dependerá de la implementación particular usada. De esta forma, la constante no depende ni del tamaño ni de los parámetros del ejemplar que se está considerando.

Por ejemplo, si en un algoritmo se necesita efectuar a adiciones, m multiplicaciones y s asignaciones. Suponiendo que el coste en tiempo de cada una de estas operaciones es t_a, t_m y t_s respectivamente, el tiempo total requerido estará acotado por:

$$t \le at_a + mt_m + st_s \le \max(t_a,t_m,t_s) \times (a + m + s)$$

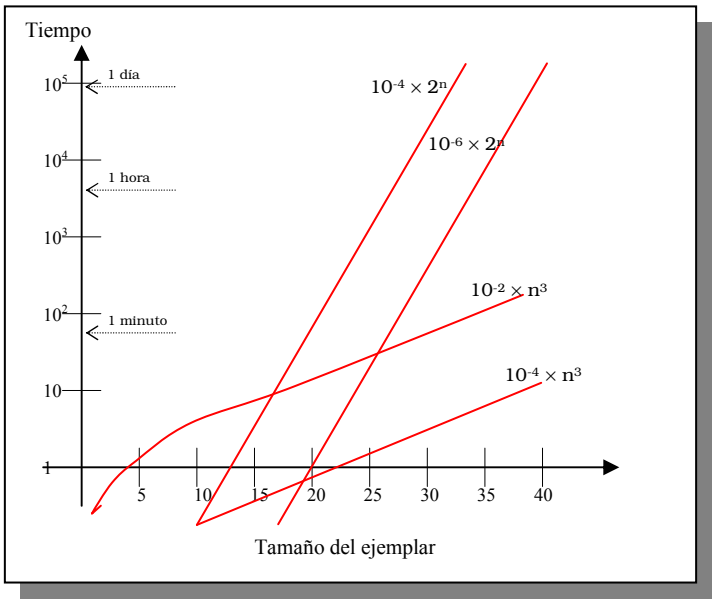
Como el tiempo exacto requerido por cada operación elemental no es importante, simplificaremos diciendo que las operaciones elementales se pueden ejecutar **a coste unitario**.

En ocasiones, lo que podrían parecer operaciones elementales, pueden no serlo dependiendo del tamaño de los operandos. Por ejemplo puede ser una operación elemental la suma de dos enteros, pero en un ordenador de 32 bits, dejará de serlo si los operandos son mayores que 65.535.

Es decir, incluso la decisión de si una instrucción tan aparentemente inocua como "$j \leftarrow i + j$" se puede considerar o no elemental, requiere el uso de nuestro juicio. En adelante, mientras no se diga lo contrario, se considerarán elementales y por tanto de coste unitario, las **sumas**, **restas**, **multiplicaciones**, **divisiones**, **operaciones de módulo**, **operaciones booleanas**, **comparaciones** y **asignaciones**.

2.6 ¿Por qué hay que buscar la eficiencia?.

Conforme van avanzando las técnicas de diseño de ordenadores, podríamos pensar que no merece la pena esforzarnos en conseguir algoritmos mejores, sino esperar a conseguir ordenadores más rápidos. Pero esto no es así. Por ejemplo, si tenemos un algoritmo exponencial que resuelve un caso de tamaño n, en $10^{-4} \times 2^n$ segundos. Para un tamaño 10, tardaría una décima de segundo. Para un tamaño 20 unos dos minutos, y aun teniéndolo todo un año ininterrumpidamente trabajando, no podríamos resolver un problema de más tamaño que 38.



Aunque consiguiéramos una máquina 100 veces más rápida, en todo un año no podríamos resolver un problema de más de 45. Sin embargo, si en lugar de hacer eso, conseguimos mejorar el algoritmo para que tenga un coste cúbico, por ejemplo de $10^{-2} \times n^3$ segundos, para resolver un tamaño de 10, tardaría 10 segundos, y uno de 20 seguirá necesitando 1 o 2 minutos, pero en la comparación del año entero trabajando, conseguiríamos resolver un tamaño de 1500.

Naturalmente esto sería rentable para tamaños grandes del problema, ya que para el primer intento, con un tamaño de 10, sería más eficiente el primer algoritmo. Por supuesto, podríamos crear un tercer algoritmo que analice primero el tamaño del problema, y aplique un algoritmo u otro en función del tamaño del problema.

2.7 Ejemplos.

Ordenación

Se trata de disponer por orden una colección de n objetos sobre los cuales está definida una ***ordenación total***. Es decir, tomando dos objetos, se sabe perfectamente cuál va antes que el otro. Por ejemplo, en los números naturales o en las fechas está claro que cumplen una *ordenación total*, pero en otros casos como los números complejos o el orden alfabético, no está tan claro.

n	*Inserción*	*Quicksort*
1.000	3 seg	0,2 seg
5.000	1:30 minutos	1 seg
10.000	9:30 horas	30 seg

```
procedimiento casilla(T[1..n])
    {ordena enteros entre 1 y 10.000}
    matriz U[1..10000]
    para k ← 1 hasta 10000 hacer U[k] ← 0
    para i ← 1 hasta n hacer
        k ← T[i]
        U[k] ← U[k] + 1
    i ← 0
    para k ← 1 hasta 10000 hacer
        mientras U[k] ≠ 0 hacer
            i ← i+1
            T[i] ← k
            U[k] ← U[k] - 1
```

Existen varios algoritmos de ordenación. Ya vimos dos de ellos en la sección 2.4. Ambos algoritmos (*selección* e *inserción*), requieren un tiempo cuadrático tanto en el caso peor como en el caso medio. Esto está bien cuando n es pequeño, pero para ejemplares mayores, existen otros algoritmos más eficiente.

Por ejemplo, tendríamos el algoritmo de **ordenación por montículo** "*heapsort*" de Willians, **por fusión** "*mergesort*", o el algoritmo de **ordenación rápida** "*quicksort*" de Hoare. Todos ellos requieren un tiempo que está en el orden de (n log n) por término medio. Los dos primeros, requieren ese tiempo incluso en el peor caso.

En la tabla anterior vemos las relaciones de tiempo en un experimento con el algoritmo de *inserción* y el *quicksort*.

Se pueden encontrar algoritmos mucho mejores aunque solo sirven para casos especiales. Por ejemplo, si hay que ordenar enteros y sabemos que están entre 1 y 10.000 podemos utilizar el procedimiento *casilla*, el cual tiene un coste del orden de n. La constante oculta, depende de la cota superior de los elementos a ordenar (en nuestro ejemplo, 10.000).

Multiplicación de enteros muy grandes.

Si los operandos son muy grandes, implica que son demasiado largos para poder almacenarlos en una sola palabra de la computadora que estemos usando, por lo que dejan de ser *operaciones elementales*. En este caso, podemos utilizar otra representación como la "*doble precisión*" de Fortran o más generalmente la ***aritmética de precisión múltiple***. La mayoría de los lenguajes tienen clases predefinidas que hacen sencilla esta tarea. Pero debemos preguntarnos cómo aumenta el tiempo para sumar, restar, multiplicar o dividir estos operandos.

Podemos medir su tamaño como el número de palabras de computadora o el número de bits necesarios para su representación.

Tanto el *algoritmo clásico* de multiplicación como la multiplicación *à la russe*, requieren un tiempo que está en el orden de mn, siendo m y n el tamaño (no el valor) de los operandos. Existen algoritmos más eficiente como el algoritmo divide y vencerás explicado en la sección 1.2 del libro de texto, que consigue una eficiencia del orden de $mn^{\lg(3/2)}$, o lo que es lo mismo $mn^{0,59}$, que es preferible a los anteriores.

Cálculo del máximo común divisor.

El algoritmo evidente para calcular el mcd (m,n), se obtiene directamente de la definición:

```
función mcd (m,n)
    i ← min (m,n) + 1
    repetir i ← i-1 hasta que i divide exactamente tanto a m como a n
    devolver i
```

El tiempo requerido por este algoritmo es del orden de la diferencia entre el menor de sus dos argumentos, y su máximo común divisor. (Nótese que hablamos de los valores de los argumentos, no de su tamaño).

Se puede usar otro algoritmo que lo que hace es factorizar primero m y n, y luego tomar el producto de los factores primos comunes a ambos, elevando cada factor a la menor de las potencias de los dos argumentos. Aunque este algoritmo es algo más eficiente, exige factorizar m y n, cosa que no se logra muy eficientemente.

```
función Euclides (m,n)
    mientras m > 0 hacer
    t ← m
    m ← n mod m
    n ← t
devolver n
```

Existe un algoritmo que, aunque es muy antiguo (más de 3.500 años de antigüedad), es mucho más eficiente. Se conoce con el nombre de **Algoritmo de Euclides**. En su versión original utiliza sustracciones sucesivas en lugar de obtener el módulo. Este algoritmo requiere un tiempo del orden del logaritmo de sus argumentos (es decir, del orden de su tamaño).

2.8 ¿Cuándo queda especificado un algoritmo?.

Decíamos al principio que la ejecución de un algoritmo no debe dejar ninguna decisión subjetiva, ni hacer uso de la intuición o de la creatividad. Que se debe demostrar que los algoritmos son correctos en abstracto, ignorando las limitaciones prácticas. Proponíamos que la mayoría de las operaciones aritméticas son elementales.

Todo esto está muy bien, pero...¿qué debemos hacer si se nos obliga a tener en cuenta las limitaciones de la máquina disponible?. No basta en ese caso simplemente con escribir una instrucción como $x \leftarrow y \times z$, dejando que el lector seleccione cualquier técnica que tenga a mano para implementar esa multiplicación. Para implementar completamente el algoritmo, también debemos especificar la forma en que deben ser implementadas las operaciones aritméticas necesarias.

Sin embargo, seguiremos utilizando la palabra ***algoritmo*** para ciertas descripciones incompletas de esta clase.

3 NOTACIÓN ASINTÓTICA

3.1 Introducción.

La eficiencia de los algoritmos puede ayudarnos a elegir uno entre varios algoritmos. Nos contentaremos con expresar el tiempo requerido por un algoritmo, salvo por una constante multiplicativa.

La notación que da título a este capítulo se llama "asintótica" porque trata acerca del comportamiento de funciones en el límite, esto es, para valores suficientemente grandes de su parámetro. De todas formas, un algoritmo asintóticamente mejor suele ser (aunque no siempre) preferible incluso para casos de tamaño moderado.

3.2 Una notación para "EL ORDEN DE".

Consideremos dos funciones de los números Naturales ($\mathbb{N}$) sobre los Reales no negativos ($\mathbb{R}^{\geq 0}$):

$t(n) = 27n^2 + \frac{3}{4}n + 12$
$f(n) = n^2$

Diremos que $t(n)$ está **en el orden de** $f(n)$ si $t(n)$ está acotada superiormente por un múltiplo real positivo de $f(n)$ para todo n suficientemente grande. Matemáticamente esto significa que existe una constante real y positiva c y un entero *umbral* n_0 tal que $t(n) \leq cf(n)$, siempre que $n \geq n_0$.

De esta forma, si la implementación de un algoritmo requiere en el caso peor $27n^2 + \frac{3}{4}n + 12$ microsegundos para resolver un caso de tamaño n, podríamos simplificar diciendo que el tiempo está ***en el orden de n²*** o que requiere un ***tiempo cuadrático***.

Existe un símbolo matemático para representar *el orden de* $\{ O(f(n)) \}$, que es el conjunto de todas las funciones que están en el orden de $f(n)$.

Para el ejemplo anterior,

$t(n) = 27n^2 + \frac{3}{4}n + 12$
$f(n) = n^2$

diremos que $\boxed{t(n) \in O(n^2)}$ o también que $\boxed{t(n) = O(n^2)}$.

Una regla útil para demostrar que una función es del orden de otra es la **regla del máximo**, que dice que:

$$O(f(n) + g(n)) = O(\text{máx}(f(n), g(n))$$

Por ejemplo

$$O(n^2 + n^3 + n \log n) = O(\text{máx}(n^2, n^3, n \log n) = O(n^3)$$

Otra observación útil es que resulta normalmente innecesario especificar la base del logaritmo dentro de la notación asintótica. Esto es debido a que:

$$\log_a n = \log_a b \times \log_b n \Rightarrow \boxed{O(\log_a n) = O(\log_b n)}$$

Es una constante

Sin embargo, la base del logaritmo **no** se puede ignorar cuando:

- ☞ Es más pequeña que 1
- ☞ No es una constante
- ☞ El logaritmo se encuentra en el exponente

La herramienta más potente para demostrar que algunas funciones están en el orden de otras es la **regla del límite**:

1.- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \Re^+$ entonces $f(n) \in O(g(n))$ y $g(n) \in O(f(n))$

2.- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n))$ pero $g(n) \notin O(f(n))$

3.- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ entonces $f(n) \notin O(g(n))$ pero $g(n) \in O(f(n))$

3.3 Otra notación asintótica.

La notación Omega.

Vimos, por ejemplo, que el algoritmo de ordenación *Quicksort* se efectúa en un tiempo en el orden de $O(n \log n)$. Pero resulta sencillo mostrar que $(n \log n) \in O(n^2)$, o incluso a $O(n^3)$. Esto resulta confuso, y es el resultado de que esta notación está pensada para *cotas superiores* sobre cantidades de recursos requeridos. Claramente necesitamos una notación dual para *cotas inferiores*. Esto es la **notación Ω.**

Consideremos dos funciones de los números Naturales ($\mathbb{N}$) sobre los Reales no negativos ($\mathbb{R}^{\geq 0}$):

$t(n) = 27n^2 + \frac{3}{4}n + 12$
$f(n) = n^2$

Diremos que $t(n)$ está **en Omega de** $f(n)$, lo cual se denota como $t(n) \in \Omega(f(n))$, si $t(n)$ está acotada *inferiormente* por un múltiplo real positivo de $f(n)$ para todo n suficientemente grande. Matemáticamente esto significa que existe una constante real y positiva d y un entero *umbral* n_0 tal que $t(n) \geq df(n)$, siempre que $n \geq n_0$.

La notación Theta.

Cuando analizamos el comportamiento de un algoritmo, nos sentimos especialmente felices si su tiempo de ejecución está acotado tanto por encima como por debajo mediante múltiplos reales (posiblemente distintos) de una misma función. Por esta razón, presentamos la notación Θ. Diremos que $t(n)$ está en Theta de $f(n)$, o lo que es lo mismo, que $t(n)$ está en el ***orden exacto*** de $f(n)$, y lo denotaremos como $t(n) \in \Theta(f(n))$, si pertenece tanto a $O(f(n))$ como a $\Omega(f(n))$.

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

La ***regla del umbral*** y la ***regla del máximo*** que formulábamos para la notación O, es aplicable *mutatis mutandis* a la notación Θ.

Sin embargo, la ***regla del límite*** se reformula de la forma siguiente:

1.- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \Re^+$ entonces $f(n) \in \Theta(g(n))$

2.- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ entonces $f(n) \in O(g(n))$ pero $f(n) \notin \Theta(g(n))$

3.- Si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ entonces $f(n) \in \Omega(g(n))$ pero $f(n) \notin \Theta(g(n))$

3.4 Notación asintótica condicional.

Hay algoritmos que resultan más fáciles de analizar si en un principio limitamos nuestra atención a aquellos casos cuyo tamaño satisfaga una cierta condición, tal como por ejemplo, ser una potencia de 2. En este ejemplo, denotamos como:

$t(n) \in \Theta(n^2 \mid n$ es una potencia de 2)

3.5 Notación asintótica con varios parámetros.

Puede suceder cuando se analiza un algoritmo, que su tiempo de ejecución dependa simultáneamente de más de un parámetro del ejemplar en cuestión. Por ejemplo, en los problemas de grafos, el tiempo suele depender tanto del número de nodos como del número de aristas.

Sea $f : \mathbb{N}\times\mathbb{N} \to \mathbb{R}^{\geq 0}$ una función de parejas de números naturales en los reales no negativos, tal como $f(m,n) = m \log n$. Sea t otra función de éstas. Diremos que $t(m,n)$ está en el orden de $f(m,n)$ si $t(m,n)$ está acotada superiormente por un múltiplo real positivo de $f(m,n)$ siempre que **tanto *m* como *n*** sean suficientemente grandes.

3.6 Operaciones sobre notación asintótica.

Para simplificar algunos cálculos, podemos manipular la notación asintótica empleando operadores aritméticos. Por ejemplo $O(f(n)) + O(g(n))$ representa el conjunto de operaciones obtenidas sumando punto a punto toda función de $O(f(n))$ a cualquier función de $O(g(n))$. Intuitivamente este conjunto representa el orden del tiempo requerido por un algoritmo compuesto por una primera fase que requiere un tiempo del orden de $f(n)$ seguido por una segunda fase que requiera un tiempo del orden de $g(n)$.

4 ANÁLISIS DE ALGORITMOS

4.1 Introducción.

Sólo después de haber determinado la eficiencia de los distintos algoritmos, será posible tomar una decisión bien informada. Pero no hay ninguna *fórmula mágica* para analizarlos. En la mayoría de los casos es una cuestión de juicio, intuición y experiencia. Sin embargo, existen algunas técnicas básicas que suelen resultar útiles.

4.2 Análisis de las estructuras de control.

El análisis de algoritmos suele efectuarse *desde dentro hacia fuera*. Es decir, en primer lugar se determina el tiempo requerido por las instrucciones individuales y después se combinan esos tiempos de acuerdo con las ***estructuras de control*** que enlazan las instrucciones del programa.

Secuencias.

Sean P_1 y P_2 dos fragmentos de un algoritmo. Pueden ser instrucciones individuales o subalgoritmos complicados. Sean t_1 y t_2 los tiempos requeridos por P_1 y P_2 respectivamente. La **regla de composición secuencial** dice que el tiempo necesario para calcular "P_1;P_2" es simplemente $t_1 + t_2$. Por la *regla del máximo*, este tiempo está en $\Theta(\text{máx}(t_1,t_2))$.

Bucles "para" (desde).

Los bucles **para** (**desde**) son los más fáciles de analizar. Consideremos el siguiente bucle:

para $i \leftarrow 1$ **hasta** m **hacer** $P(i)$

Adoptaremos el convenio de que si m=0 no existe error, sino que en ese caso el bucle no se efectúa ni una sola vez. Suponiendo que t es el tiempo para ejecutar $P(i)$, el tiempo total será de:

$$\tau = mt$$

El tiempo t, puede ser un valor que no depende de i, o bien el resultado de un algoritmo más extenso que trabaja con ejemplar de tamaño n.

En realidad hemos hecho una simplificación, ya que no hemos tenido en cuenta el tiempo necesario para el *control del bucle*.

El análisis de bucles **para** que empiezan en un valor que no es 1 o que avanzan con pasos mayores, es evidente. Por ejemplo, analicemos el siguiente bucle:

para $i \leftarrow 5$ **hasta** m **paso** 2 **hacer** $P(i)$

Aquí, $P(i)$ se ejecuta $((m-5) \div 2) + 1$ veces siempre que $m \geq 3$.

Llamadas recursivas.

El análisis de algoritmos recursivos suele ser sencillo, al menos hasta cierto punto. Una inspección sencilla del algoritmo suele dar lugar a una ecuación de recurrencia que "remeda" (imita) el flujo de control dentro del algoritmo. Una vez conseguido esto, con las técnicas de resolución de recurrencias, se puede transformar en una ecuación asintótica no recursiva, que es más sencilla.

Por ejemplo, del algoritmo:

```
función Fibrec(n)
    si n<2 entonces devolver n
    si no devolver Fibrec(n-1) + Fibrec(n-2)
```

Podemos deducir fácilmente la función recursiva del tiempo empleado T(n):

$$T(n) = \begin{cases} a & \text{si } n = 0 \text{ ó } n = 1 \\ T(n-1) + T(n-2) + h(n) & \text{en caso contrario} \end{cases}$$

De lo que podremos deducir que *Fibrec*(*n*) requiere un tiempo exponencial en n.

Bucles "mientras" (while) y "repetir" (repeat).

Estos bucles suelen ser más difíciles de analizar que los bucles **para** (**for**), porque no existe una forma evidente *a priori* de saber cuántas veces tendremos que pasar por el bucle. La técnica estándar es hallar una función de las variables implicadas cuyo valor se decremente en cada pasada. Para determinar el número de veces que se repite el bucle, necesitamos saber la forma en que disminuye el valor de esa función. Una aproximación alternativa al análisis del bucle **mientras**, consiste en tratarlo como un algoritmo recursivo.

4.3 Uso de un barómetro.

El análisis de muchos algoritmos se simplifica de forma significativa cuando es posible aislar una instrucción –o comprobación- como *barómetro*. Una ***instrucción barómetro*** es aquella que se ejecuta por lo menos con tanta frecuencia como cualquier instrucción del algoritmo. (No pasa nada si alguna instrucción se ejecuta como mucho un número constante de veces más que el barómetro, ya que la notación asintótica absorberá dicha contribución). El tiempo requerido por el algoritmo completo es del orden exacto del número de veces que se ejecuta la *instrucción barómetro*.

Cuando un algoritmo contiene varios bucles anidados, toda instrucción del bucle más interno puede utilizarse en general como barómetro. Sin embargo hay ocasiones en las que hay que tener en consideración el control implícito del bucle. Esto sucede, típicamente, cuando algunos bucles se ejecutan cero veces, porque estos bucles requieren en realidad, un tiempo aun cuando no se realicen ejecuciones de la instrucción barómetro.

El uso de un barómetro es una herramienta útil para simplificar el análisis de muchos algoritmos, pero esta técnica deberá utilizarse con cuidado.

4.4 Ejemplos adicionales.

Ordenación por selección.

En el algoritmo de ordenación por selección, parece natural utilizar como instrucción barómetro, la comprobación del bucle más interno: "**si** *T[j] < minx* **entonces**", como medida del tiempo total de ejecución del algoritmo, porque ninguno de los bucles se puede ejecutar cero veces (en cuyo caso el control del bucle sería más costoso en términos de tiempo que nuestro barómetro). El número de veces que se ejecuta la comprobación es:

$$\sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \sum_{i=1}^{n-1} (n-i) = \sum_{k=1}^{n-1} k = n(n-1)/2$$

Por tanto el número de veces que se ejecuta la instrucción barómetro está en $\Theta(n^2)$.

```
procedimiento seleccionar (T[1..n])
    para i ← 1 hasta n-1 hacer
        minj ← i; minx ← T[i]
        para j ← i+1 hasta n hacer
            si T[j] < minx entonces
                minj ← j
                minx ← T[j]
            fsi
        fpara
    T[minj] ← T[i]
    T[i] ← minx
```

Ordenación por inserción.

```
procedimiento insertar (T[1..n])
    para i ← 2 hasta n hacer
        x ← T[i]; j ← i-1
        mientras j>0 y x<T[j] hacer
            T[j+1] ← T[j]
            j ← j-1
        fmientras
        T[j+1] ← x
```

En el caso de la ordenación por inserción, seleccionamos como barómetro el número de veces que se comprueba la condición "*j>0* **y** *x<T[j]*" del bucle **mientras**.

A diferencia de la ordenación por selección, aquí debemos analizar *el caso peor*, y siendo así, también está en $\Theta(n^2)$.

4.5 Análisis del caso medio.

Vimos que la ordenación por inserción requiere un tiempo cuadrático en el *caso peor*. Es fácil comprobar que concluye en un tiempo lineal en el *caso mejor*. Es por tanto natural, preguntarse qué ocurre en el ***caso medio***. Esto requiere suponer *a priori* una distribución de probabilidad entre los posibles casos a resolver, y el análisis puede conducir a error si nuestras suposiciones no se corresponden con la realidad. Normalmente supondremos que todos los posibles casos a resolver tienen la misma probabilidad de aparecer. Además, en los casos de ordenación, resulta más sencillo suponer también, que los elementos que hay que ordenar son distintos.

4.6 Análisis amortizado.

En algunas ocasiones, el análisis del caso peor es excesivamente pesimista. Considérese por ejemplo un proceso *P*, en el que dos llamadas sucesivas e idénticas a *P* podrían requerir cantidades de tiempo radicalmente distintas.

Tomando un ejemplo real, consideremos *P* como la tarea de tomar una taza de café. Normalmente sólo hay que *servirse dicha taza*. Pero en ocasiones, nos encontramos la cafetera vacía, y puede que también todas las tazas sucias, por lo que la tarea de tomar una taza de café, implica en este caso un tiempo mucho mayor que incluye el *fregar las tazas* y *preparar una nueva cafetera*. De todas formas, este tiempo invertido, sirve para que las sucesivas llamadas al procedimiento, tengan el coste mínimo de *servirse la taza de café*.

En esta situación, tomar el peor caso produce una respuesta correcta suponiendo que estemos satisfechos con la notación O, por oposición a la notación Θ, pero la respuesta sería pesimista. No podemos considerar (como hacíamos en el *caso medio*), que cualquier llamada al procedimiento es independiente entre sí, sino que los tiempos requeridos tienen una fuerte relación entre sí. Por esto, no tomaremos la media sobre las posibles entradas, sino la media sobre *llamadas sucesivas*.

Un ejemplo clásico de este comportamiento en programación concierne a la reserva de espacio con una necesidad ocasional de "recogida de basura".

Para alcanzar los resultados de un análisis amortizado, hay dos técnicas principales: el enfoque de la función potencial y el truco de contabilidad.

Funciones potenciales.

Supongamos que el proceso a analizar, modifica una base de datos, y que la eficiencia de tal proceso, depende del estado actual de la base de datos. Utilizamos una *función potencial* entera (Φ)de la base de datos, que nos da una idea de lo organizada que está la base de datos. De esta forma ϕ_0 nos dará el valor de Φ en el estado inicial, y un valor ϕ_i nos dará el valor de Φ después de la *i*-ésima llamada.

Se permite que las llamadas al proceso requieran más tiempo que el valor medio, siempre y cuando mejoren la organización de la base de datos. A la inversa, se permite que las llamadas rápidas, desorganicen la base de datos. Esto mismo ocurría con el ejemplo de la taza de café: cuanto más rápida llenemos la taza, más probabilidad de derramarlo, por lo que se necesitará más tiempo cuando toque limpiar.

Definimos el ***tiempo amortizado*** requerido por esa llamada al proceso, de la forma:

$$\tau_i = t_i + \phi_i - \phi_{i-1}$$

Es decir, el tiempo realmente necesario para efectuar la llamada, más el incremento potencial causado por esa llamada.

El reto, cuando uno intenta aplicar esta técnica, consiste en determinar la función potencial adecuada.

El truco de contabilidad.

Esta técnica se puede considerar como una remodelación del enfoque de la función potencial, pero resulta más fácil de aplicar en algunos contextos. Supongamos que se ha conjeturado una cota superior τ del tiempo invertido en el sentido amortizado siempre que se llama al proceso *P*. Para utilizar el truco de contabilidad, es preciso establecer una *cuenta bancaria virtual*, en la que se deposita inicialmente τ fichas cada vez que se llama al proceso *P*. Cada vez que se ejecuta la instrucción barómetro, es preciso pagar una de esas fichas. La regla es que la cuenta nunca esté en números rojos. Esto asegura que sólo se permiten operaciones lentas si se han producido ya número suficiente de operaciones rápidas, las cuales dejan *remanente* en la cuenta.

4.7 Resolución de recurrencias.

Con un poco de experiencia y de intuición, la mayoría de las recurrencias se pueden resolver mediante suposiciones inteligentes. Sin embargo, existe una potente técnica que se puede utilizar para resolver de forma casi automática ciertas clases de recurrencias: la técnica de la *ecuación característica*.

Suposiciones inteligentes.

El enfoque suele desarrollarse en cuatro etapas:

1. Calcular los primeros valores de la recurrencia.
2. Buscar una regularidad
3. Inventar una forma general adecuada
4. Demostrar por inducción matemática que esta forma es correcta.

Por ejemplo, la recurrencia:

$$T(n) = \begin{cases} 0 & \text{si } n = 0 \\ 3T(n \div 2) + n & \text{en caso contrario} \end{cases}$$

Tiene unos primeros valores de:

n	***T*(*n*)**	***T*(*n*) como potencia**
1	1	1
2	5	$3 \times 1 + 2$
2^2	19	$3^2 \times 1 + 3 \times 2 + 2^2$
2^3	65	$3^3 \times 1 + 3^2 \times 2 + 3 \times 2^2 + 2^3$
2^4	211	$3^4 \times 1 + 3^3 \times 2 + 3^2 \times 2^2 + 3 \times 2^3 + 2^4$
2^5	665	$3^5 \times 1 + 3^4 \times 2 + 3^3 \times 2^2 + 3^2 \times 2^3 + 3 \times 2^4 + 2^5$

De lo que deducimos fácilmente que:

$$T(2^k) = \sum_{i=0}^{k} 3^{k-i} 2^i = 3^{k+1} - 2^{k+1}$$

Quedándonos únicamente el demostrar este resultado por inducción matemática.

Ecuación característica.

Recurrencias homogéneas.

Estudiaremos primero las recurrencias homogéneas lineales con coeficientes constantes, esto es, recurrencias de la forma:

$$a_0 t_n + a_1 t_{n-1} + \ldots + a_k t_{n-k} = 0$$

donde los t son los valores que estamos buscando. Esta recurrencia es:

- **Lineal** porque no contiene términos de la forma $t_{n-i} \times t_{n-j}$, t^2_{n-i}, y similares.
- **Homogénea** porque la combinación de los t_{n-i} es igual a cero.
- **Con coeficientes constantes** porque las a_i son constantes.

Estudiaremos este tipo de recurrencias, siguiendo el ejemplo de la ***sucesión de Fibonacci***:

$$f_n = \begin{cases} n & \text{si } n = 0 \text{ o } n = 1 \\ f_{n-1} + f_{n-2} & \text{en caso contrario} \end{cases}$$

Primero escribiremos la recurrencia para que se adapte a la forma de la ecuación general:

$$f_n - f_{n-1} - f_{n-2} = 0$$

Por lo que se corresponde con una *recurrencia homogénea lineal de coeficientes constantes*, con k=2, a_0=1 y a_1=a_2=-1.

La ecuación de grado k en x, se denomina ***ecuación característica*** de la recurrencia:

$$a_0x^k + a_1x^{k-1} + \ldots + a_k = 0$$

Que en nuestro ejemplo sería:

$$x^2 - x - 1 = 0$$

cuyas raíces son:

$$r_1 = \frac{1+\sqrt{5}}{2} \quad \text{y} \quad r_1 = \frac{1-\sqrt{5}}{2}$$

La solución general, por tanto, es de la forma:

$$f_n = c_1 r_1^n + c_2 r_2^n$$

Y de la primera definición de la recurrencia, podemos sacar los valores de la función para n=0 y n=1, de donde:

$$\begin{aligned} c_1 + c_2 &= 0 \\ r_1c_1 + r_2c_2 &= 1 \end{aligned}$$ Resolviendo estas ecuaciones, obtenemos que $c_1 = \frac{1}{\sqrt{5}}$ y $c_2 = -\frac{1}{\sqrt{5}}$

Por tanto,

$$f_n = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

, que es la famosa fórmula de Moivre para la sucesión de Fibonacci.

Recurrencias no homogéneas.

La solución de una recurrencia lineal con coeficientes constantes se vuelve más difícil cuando no es homogénea, es decir, cuando la combinación lineal no es igual a cero.

$$a_0\, t_n + a_1\, t_{n-1} + \ldots + a_k\, t_{n-k} = b^n p(n)$$

En este caso, deberemos hacer las operaciones necesarias para transformar la recurrencia no homogénea, en una recurrencia homogénea.

Por ejemplo, la recurrencia:

$$t_n - 2t_{n-1} = 3^n$$ en este caso, b=3 y $p(n)$=1.

puede transformarse en homogénea de la siguiente manera:

- Primero multiplicamos por 3: $3t_n - 6t_{n-1} = 3^{n+1}$
- Sustituimos n por n-1: $3t_{n-1} - 6t_{n-2} = 3^n$
- Restamos esta ecuación de la primera: $\boxed{t_n - 5t_{n-1} + 6t_{n-2} = 0}$, que ya es homogénea.

Cambios de variable.

A veces es posible resolver recurrencias más complicadas efectuando un cambio de variable.

Transformaciones de intervalo.

Cuando se hace un cambio de variable, se transforma el dominio de la recurrencia. En lugar de hacer esto, puede resultar útil transformar el intervalo para obtener una recurrencia que tenga una forma que sepamos resolver. Ambas transformaciones pueden utilizarse a la vez en algunas ocasiones.

Dos recurrencias típicas.

En algoritmos de tipo divide y vencerás, se producen dos recurrencias típicas, en las que damos la solución asintótica de sus costes.

Reducción del problema mediante sustracción.

Este tipo de recurrencias aparece al calcular el coste de programas recursivos en los que el tamaño del problema decrece en una cantidad constante de una activación recursiva a las siguientes. El aspecto general de la ecuación recurrente es el siguiente:

$$T(n) = \begin{cases} cn^k & , \text{si } 0 \le n < b \\ aT(n-b) + cn^k & , \text{si } n \ge b \end{cases}$$

En este caso, a = número de llamadas recursivas que se realizan en casa pasada del algoritmo.
b = cantidad que se reduce el tamaño del problema de una llamada a la siguiente.

La solución sería:

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

Reducción del problema mediante división.

En este esquema, una llamada al programa con un problema de tamaño n, genera a llamadas recursivas con subproblemas de tamaño n/b.

La ecuación recurrente es la siguiente:

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

En este caso, a = número de llamadas recursivas que se realizan en casa pasada del algoritmo.
b = cantidad por la que se divide el tamaño del problema de una llamada a la siguiente.

La solución sería:

$$T(n) \in \begin{cases} \Theta(n^k) & , \text{si } a < b^k \\ \Theta(n^k \log n) & , \text{si } a = b^k \\ \Theta(n^{\log_b a}) & , \text{si } a > b^k \end{cases}$$

5 ESTRUCTURAS DE DATOS

El uso de unas estructuras de datos bien escogidas suele ser un factor crucial en el diseño de algoritmos eficientes.

5.1 Matrices (Arrays), Pilas y Colas.

Una **matriz** es una estructura de datos que consta de un número fijo de *ítems* (*elementos*) del mismo tipo. En una matriz monodimensional, el acceso a todo *ítem* particular se efectúa especificando un solo *subíndice* o *índice*. Por ejemplo:

tab: **matriz**[1..50] **de** *enteros*

De esta forma, *tab*[1] alude al primer elemento de la matriz y *tab*[50] al último. La propiedad esencial de una matriz es que podemos calcular la posición de un elemento dado, en un tiempo constante. Por ejemplo, si sabemos que la matriz anterior *tab*, comienza en la dirección de memoria 5000 y que las variables enteras ocupan 4 *bytes* de almacenamiento cada una, entonces la dirección del elemento cuyo índice es *k* está dada por: **4996 + 4*k***.

Por otra parte, toda operación que implique a todos los elementos de una matriz, tenderá a requerir más tiempo a medida que crezca el tamaño de la misma. Esas operaciones requieren un tiempo que está en $\Theta(n)$.

Las matrices monodimensionales permiten implementar eficientemente la estructura de datos llamada ***pila***. Es una estructura en la que se añaden elementos, y pueden ser recuperados de forma que el último en llegar es el primero en salir (*LIFO.- Last in, first out*). Se implementa con una matriz del tamaño máximo de la pila y un *contador* de forma que:

- **Vaciar** la pila implica poner *contador* a cero.
- **Apilar** (*push*) un elemento implica incrementar el *contador* y escribir en *pila*[*contador*].
- **Desapilar** (*pop*) un elemento implica leer *pila*[*contador*] y decrementar *contador*.
- Hay que **comprobar** que no se añadan más elementos si la *pila* está llena, y no se lean cuando la *pila* está vacía.

Análogamente se puede implementar la estructura llamada ***cola***. En este caso, el primer dato en salir es el primero en salir (*FIFO.- first in, first out*).

Tanto para pilas como para colas, una desventaja de implementarla con matrices es la necesidad de reservar espacio al principio para el máximo número de elementos que se prevea; si en alguna ocasión el espacio no resultara suficiente, es difícil reservar más, mientras que si se reserva demasiado espacio, es un desperdicio.

El índice de una matriz es casi siempre un entero. Sin embargo, otros tipos denominados "*escalares*" se pueden emplear también. Por ejemplo:

lettab: **matriz**['a'..'z'] **de** *valor*

es una posible forma de declarar 26 valores, indexados mediante las letras de la 'a' a la 'z'. Los índices no pueden ser *números reales* ni estructuras tales como *cadenas* o *conjuntos*.

No solo se pueden declarar matrices unidimensionales, sino que es posible declarar matrices con dos o más índices, de forma similar:

matriz: **matriz**[1..20,1..20] **de** *complejo*

es una matriz cuadrada que contiene 400 (20×20) elementos de tipo *complejo*.

En este caso, una referencia a un elemento requiere de la definición de dos índices (Por ejemplo, *matriz*[5,7]). También aquí es posible calcular la dirección de cualquier elemento en un tiempo constante, y por tanto, una operación que recorra todos los elementos de una matriz de dimensión $n \times n$, requiere un tiempo que está en $\Theta(n^2)$.

5.2 Registros y punteros (apuntadores).

Un ***registro*** también consta de un número fijo de elementos, que suelen llamarse campos, y que son de tipos posiblemente *distintos*. Por ejemplo, si queremos contener la información referente al nombre, edad, peso y sexo de una persona, podemos hacerlo mediante el registro:

tipo *persona* = **registro**
nombre : *cadena*
edad : *entero*
peso : *real*
varon : *booleano*

Entonces, si *Juan* es una variable del tipo *persona*, podemos hacer alusión a los campos de la siguiente forma:

Juan.nombre = "Juan Ruiz"
Juan.edad = 36
Juan.peso = 72,4
Juan.varon = *Cierto*

Las matrices pueden formar parte de los registros en uno de sus campos, y los registros se pueden almacenar en una matriz, declarándola por ejemplo:

clase: **matriz**[1..50] **de** *persona*

En este caso, para referirse por ejemplo, a la edad del 7º elemento, se denotaría como:

clase[7].*edad*.

Hay muchos lenguajes que permiten crear y destruir registros *dinámicamente*. Ésta es una de las razones por las cuales se suelen utilizar los registros en conjunción con los ***punteros***. Una declaración como:

tipo *jefe* = ↑*persona*

dice que *jefe* es un puntero que señala a un registro cuyo tipo es *persona*. es posible crear dinámicamente uno de estos registros mediante una sentencia como:

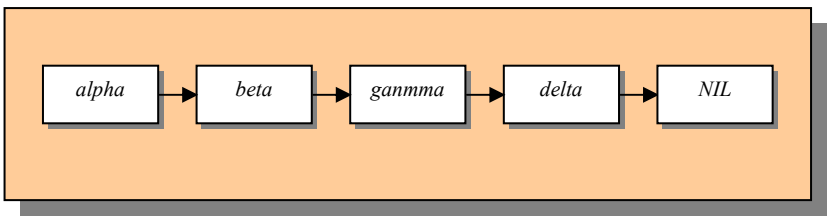
jefe ← **crea** *persona*

Ahora *jefe*↑ (obsérvese que ahora la flecha sigue al nombre) significa "el registro que apunta a *jefe*" Y haremos alusión a sus campos como: *jefe*↑.*nombre*, *jefe*↑.*edad*, y así sucesivamente.
Un puntero que tenga el valor **nulo** (*nil*), no apunta a ningún sitio.

5.3 Listas.

Una ***lista*** es una colección de elementos de información dispuestos en un cierto orden. A diferencia de las matrices y registros, el número de elementos de la lista no suele estar fijado ni suele estar limitado por anticipado. La estructura debería permitirnos determinar por ejemplo, cual es el primero o el último de la lista, o cual es el predecesor o sucesor (si existen) de un elemento dado.

Las listas admiten operaciones como: *insertar un nodo adicional* (quizás en una posición intermedia), *borrar un nodo cualquiera*, *copiar una lista*, *contar el número de elementos*...etc.



Existen diferentes formas de implementación de las listas, con sus particulares características:

En forma de matriz:

tipo *lista* = **registro**
contador : 0..*longmax*
valor : **matriz**[1..*longmax*] **de** *informacion*

Las operaciones de saber el primero o el último, o saber el predecesor o el sucesor, o saber el número de elementos, se calculan fácilmente. Sin embargo, añadir o eliminar un elemento requieren un coste que está en el caso peor, en el orden del tamaño actual de la lista.

Existe un caso especial de *lista*, que es el que ya vimos de ***pila***. En este caso, solo se añaden o borran elementos al final de la pila, por lo que esta implementación es especialmente eficiente, aunque con la pega que ya vimos de la necesidad de reserva anticipada de espacio.

Empleando punteros.

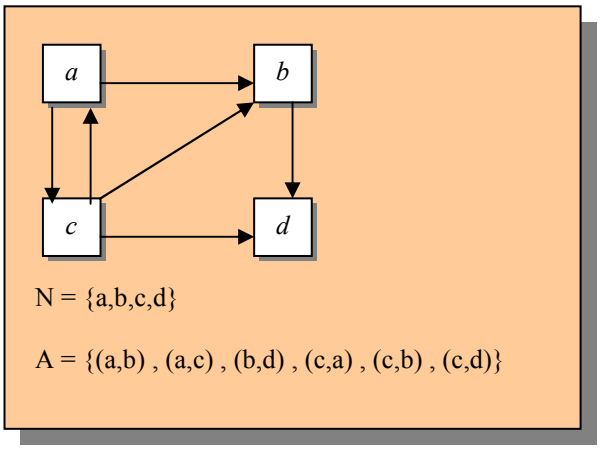
Los nodos suelen ser de una forma similar a:

tipo *lista* = ↑*nodo*
tipo *nodo* = **registro**
    valor : *informacion*
    siguiente : ↑*nodo*

Cada nodo incluye un puntero que señala a su sucesor, excepto el último, que tiene el valor *NIL*.
En este caso, no es necesario la reserva inicial de espacio, porque la lista se va creando o eliminado *dinámicamente*. Aun cuando se utilicen punteros adicionales para saber rápidamente cual es el primer o último elemento, resulta difícil examinar el *k*-ésimo elemento sin tener que recorrer los *k* elementos anteriores. Sin embargo, una vez que se ha llegado a ese elemento *k*, es fácil la inserción de un nuevo nodo, o el borrado de uno existente.
Si añadimos a la estructura un nuevo puntero a cada nodo, que apunte a su predecesor, la lista puede ser recorrida en ambas direcciones. También se pueden construir *listas circulares* haciendo que el último elemento señale al primero.

5.4 Grafos.

Intuitivamente.- Un grafo es un conjunto de *nodos* unidos mediante un conjunto de líneas (para grafos no dirigidos) o flechas (para grafos dirigidos), llamados *aristas*. Cualquier grafo puede contener *caminos* y *ciclos*. Un grafo es *conexo* si desde cualquier nodo, existe un camino hacia cualquier otro (aunque sea recorriendo las flechas en sentido inverso). Un grafo es *fuertemente conexo* si existe un camino para cualesquiera dos nodos, respetando el sentido de las flechas. Nunca hay más de una flecha (a no ser que sean en sentidos opuestos) que unan dos nodos en un grafo dirigido, ni más de una línea en un grafo no dirigido.



Formalmente.- Un grafo es una pareja $G=<N,A>$ en donde N es un conjunto de nodos y A es un conjunto de aristas que se denotan como parejas de nodos: Par ordenado (a,b) para grafos dirigidos, y conjunto $\{a,b\}$ para grafos no dirigidos.

Existen por lo menos dos formas de representar un grafo en una computadora:

Mediante una matriz de adyacencia:

tipo *grafoadya* = **registro**
    valor : **matriz**[1..*numnodos*] **de** *informacion*
    adyacente : **matriz**[1..*numnodos*,1..*numnodos*] **de** *Boolean*

Si una arista (i,j) existe, entonces *grafoadya.adyacente*[*i*,*j*]=*Verdadero*. En caso contrario es *Falso*.
En un grafo no dirigido, la matriz *adyacente* es necesariamente simétrica.
Para esta implementación, buscar si existe o no una arista, o ver el valor de un nodo, tiene coste constante. Pero si deseamos conocer todos los nodos que están conectados a uno dado, necesitamos recorrer una fila completa (para grafos no dirigidos) o una fila y una columna (para dirigidos), lo cual está en $\Theta(numnodos)$. El espacio necesario para almacenar el grafo, es cuadrático con respecto a *numnodos*.

Mediante una lista.

tipo *grafolista* = **matriz**[1..*numnodos*] **de**
registro
valor : *informacion*
vecinos : *lista*

A cada nodo *i*, se le asocia una lista formada por todos los nodos *j* tales que existe la arista (*i*,*j*).
Si el número de aristas es pequeño, esta representación utiliza menos espacio. También se analiza rápidamente todos los vecinos de un nodo. Sin embargo, es menos eficiente el determinar si existe o no una conexión directa entre dos nodos.

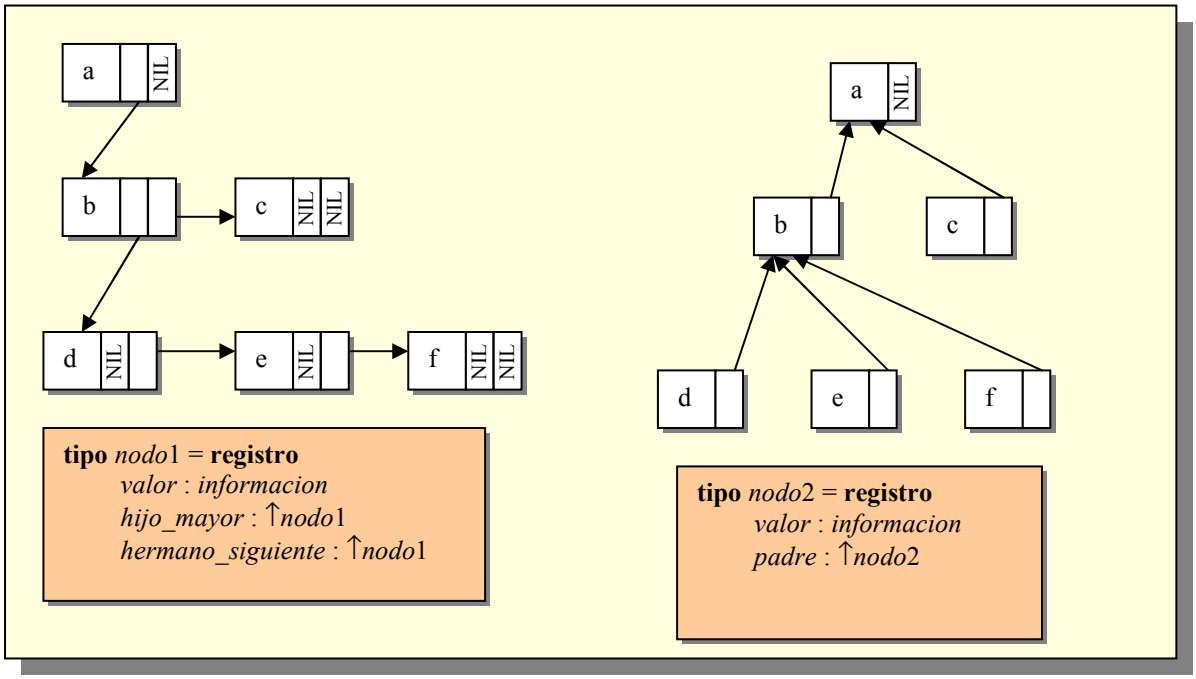
5.5 Árboles.

Un ***árbol*** (en realidad un *árbol libre*) es un grafo acíclico, conexo y no dirigido. Igualmente, se puede definir como un grafo no dirigido en el cual existe exactamente un camino entre todo par de nodos dado.
Por tanto, se pueden utilizar las mismas implementaciones que para los grafos.
Los árboles tienen un cierto número de propiedades:

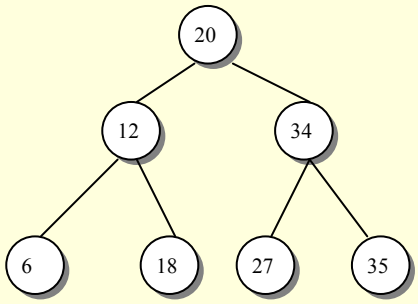
- Un árbol con *n* nodos posee exactamente *n*-1 aristas.
- Si se añade una única arista a un árbol, entonces el grafo resultante contiene un único ciclo.
- Si se elimina una única arista de un árbol, entonces el grafo resultante ya no es conexo.

Nos interesan particularmente, los árboles con raíz, es decir los que tienen un nodo llamado raíz que es especial.
Con el mismo significado que en un árbol genealógico, introducimos los conceptos de ***padre***, ***hijo***, ***hermanos***, ***antepasados***, ***descendientes***.
Una ***hoja***, es un nodo que no tiene *hijos*.

Existen muchas formas de implementar los nodos de un árbol, cada una de ellas con sus ventajas e inconvenientes. Por ejemplo:



Árbol binario es el que cada nodo tiene 0, 1 o 2 hijos. De esta forma definiremos su *hijo_izquierdo* y su *hijo_derecho*.
Un árbol binario es un **árbol de búsqueda**, si el valor de cualquier nodo es mayor o igual que su *hijo_izquierdo* (y por tanto que sus descendientes), y es menor o igual que su *hijo_derecho* (y sus descendientes). Este árbol tiene la ventaja de ser muy eficiente a la hora de hacer búsquedas. En el árbol de la figura siguiente, podemos encontrar el valor 27, con solo 3 comparaciones, a pesar de existir 7 nodos.




```
funcion buscar(x,r)
    si r = nulo entonces {x no está en el árbol}
                devuelve nulo
    sino si x = r↑.valor entonces devolver r
    sino si x < r↑.valor entonces devolver buscar(x, r↑.hijo_izq)
    sino devolver buscar(x, r↑.hijo_der)
```

Busca el valor *x* en el árbol señalado por el puntero *r*.

Resulta sencillo mantener un árbol de búsqueda actualizado. Es decir, mantener la propiedad de árbol de búsqueda aunque se introduzca un nuevo nodo, o se borre uno existente. Sin embargo, es necesario además mantenerlo *equilibrado*, es decir, que no existan muchos nodos que puedan tener un solo hijo, ya que en este caso, se hacen ramas demasiado largas y delgadas, perdiendo la efectividad en la búsqueda. Existe toda una gama de métodos para mantener equilibrado un árbol, y conseguir así que el coste de las búsquedas esté en O(log *n*).

Aclaremos términos que pueden resultar confusos:

- La **altura** de un nodo es el número de aristas que hay en el camino más largo que vaya desde el nodo en cuestión a una hoja.
- La **profundidad** de un nodo es el número de aristas que haya en el camino que va desde el nodo raíz hasta el nodo en cuestión.
- El **nivel** de un nodo es igual a la altura de la raíz menos la profundidad del nodo estudiado.

En el árbol de la página anterior, los valores serán:

Nodo	Altura	Profundidad	Nivel
a	2	0	2
b	1	1	1
c	0	1	1
d	0	2	0
e	0	2	0
f	0	2	0

Finalmente, definiremos la ***altura del árbol***, como la altura de su raíz; ésta es además la profundidad de la hoja más profunda y el nivel de la raíz.

5.6 Tablas asociativas.

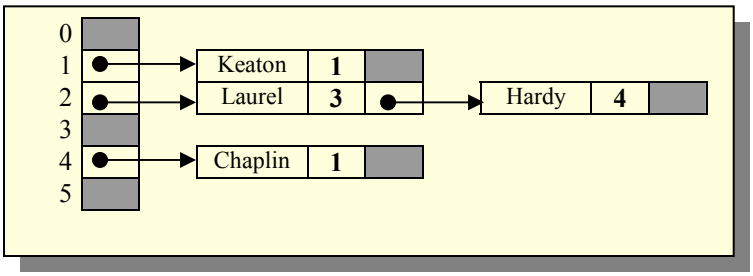
Una ***tabla asociativa*** es como una matriz, salvo que su índice no está restringido a encontrarse entre dos cotas predeterminadas, ni utilizar enteros o escalares. Se pueden usar cadenas como índice. Por ejemplo, se puede usar tanto *T*[1], como *T*["Juan"]. El espacio de almacenamiento no hay que reservarlo con anterioridad, sino que se va creando a medida que se necesita. Esto tiene, naturalmente, un coste; y es que no se garantiza que todas las búsquedas requieran un coste constante. La forma más sencilla de implementarla es usando una lista:

```
tipo lista_tabla = ↑nodo_tabla
tipo nodo_tabla = registro
            indice : tipo_indice
            valor : informacion
            siguiente : ↑nodo_tabla
```

En el caso peor, todas las búsquedas pertenecen a elementos que no están en la tabla, por lo que obliga a recorrerla por completo. Por tanto, una secuencia de *n* accesos con un total de *m* índices, requiere un tiempo que se encuentra en Ω(*nm*). Si hacemos que los índices sean comparables entre sí, se pueden utilizar *árboles equilibrados*, haciendo que el coste esté en O(*n* log *m*).

Prácticamente todos los compiladores utilizan una tabla asociativa para implementar la *tabla de símbolos*. Se guardan en los índices de la tabla, los posibles identificadores del lenguaje, y en el valor,

información relevante de ese identificador. Para mejorar la eficiencia en programas con número elevado de identificadores, se utiliza una técnica conocida con el nombre de *codificación dispersiva* (*hash coding*) o simplemente ***dispersión*** (***hashing***), a pesar de tener un *caso peor* desastroso.



Esta técnica utiliza una *función de dispersión*, la cual debería dispersar eficientemente todos los índices probables: $h(x)$ debería ser diferente de $h(y)$, para la mayoría de los pares $x \neq y$. Por ejemplo, si un valor a, se obtiene del nombre del identificador (por su código ASCII por ejemplo), y N es un número primo, una buena solución sería $h(a) = a \bmod N$.

Cuando $x \neq y$, pero $h(x)=h(y)$, se dice que ha habido una ***colisión*** entre x e y. Existen varias técnicas para abordar la colisión. La más sencilla es la *tabla de dispersión abierta* o *con encadenamiento*. Por ejemplo, la figura muestra la solución para el siguiente ejemplo:

- 4 índices que son: "Laurel", "Chaplin", "Hardy" y "Keaton".
- La información que guardan es:

  T[Laurel] ← 3
  T[Chaplin] ← 1
  T[Hardy] ← 4
  T[Keaton] ← 1

- En este ejemplo, N=6, h(Keaton) = 1, h(Laurel) = h(Hardy) = 2 y h(Chaplin) = 4.

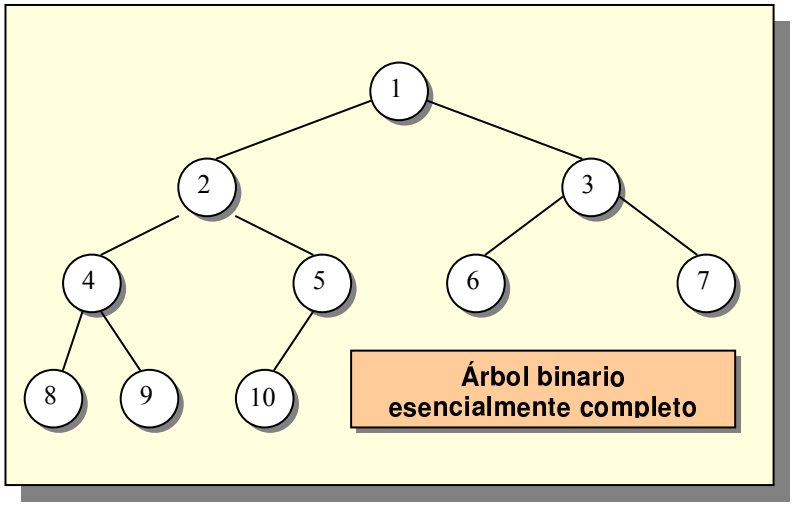
El *factor de carga* de la tabla es m/N, donde m es el número de índices distintos que se han almacenado en la tabla, y N es el tamaño de la matriz que se emplea para implementarla. Si se mantiene el factor de carga entre 0'5 y 1, la tabla ocupa un espacio que está en $\Theta(m)$, lo cual es óptimo.

5.7 Montículos (heaps).

Un ***montículo*** es un tipo especial de árbol con raíz, que se puede implementar eficientemente en una matriz, sin ningún puntero explícito. Esta estructura es interesante, porque se presta a la técnica de ordenación conocida como *ordenación por el método del montículo* (*heapsort*).

Se dice que un árbol binario es *esencialmente completo* si todo nodo interno, con la posible excepción de un nodo especial, tiene exactamente dos hijos. El nodo especial, si existe uno, está en el nivel 1 y posee solamente hijo izquierdo. Además, o bien todas las hojas se encuentran en el nivel 0, o bien están en los niveles cero y uno, y ninguna hoja del nivel 1 está a la izquierda de un nodo interno del mismo nivel.

Intuitivamente es como si los nodos internos se suben en el árbol todo lo posible, con los nodos internos del último nivel empujados a la izquierda; las hojas llenan el último nivel que contiene nodos internos, si es que queda algún espacio, y después se desbordan hacia la izquierda en el nivel cero.



- Si un árbol binario esencialmente completo tiene una altura k, entonces hay un nodo (la raíz) en el nivel k, 2 nodos en el nivel k-1, ...2^{k-1} nodos en el nivel 1, y al menos un nodo, pero no más de 2^k en el nivel 0.
- Si el árbol contiene n nodos en total, se sigue que $2^k \leq n \leq 2^{k+1}$.
- Análogamente, la altura de un árbol que tiene n nodos es $k = \lfloor \lg n \rfloor$.

Esta clase de árbol, se puede representar en una matriz, colocando los nodos en el orden de izquierda a derecha comenzando por la raíz, y descendiendo niveles. El árbol de la figura anterior está numerado tal y como se representaría en una matriz. De esta forma, el padre del nodo representado en $T[i]$, se encuentra en $T[i \div 2]$ (para $i>1$, ya que la raíz $T[1]$ carece de padre), y los hijos del nodo representado en $T[i]$, estarán en $T[2i]$ y $T[2i+1]$ siempre que existan. El sub-árbol cuya raíz está en $T[i]$, también es fácil de identificar.

Un **montículo** es un árbol binario esencialmente completo, cada uno de cuyos nodos incluye un elemento de información denominado *valor del nodo* y que tiene la propiedad consistente en que el valor de cada nodo interno es mayor o igual que los valores de sus hijos. Esto se llama *propiedad del montículo*. Esta propiedad asegura que un nodo tiene un valor superior a cualquiera que esté en los sub-árboles que cuelgan de él. En particular, la raíz es el nodo con mayor valor del árbol.

Mantener la propiedad del montículo es fácil:

- ☞ Si un nodo aumenta de valor de forma que se hace mayor que su padre, solo hay que intercambiar los valores de padre e hijo, y continuar esta misma operación hacia arriba. Diremos que el nodo modificado, ha ***flotado*** hasta su nueva posición.
- ☞ Si por el contrario, el valor decrece por debajo de alguno de sus hijos, basta con intercambiarlo con el mayor de sus hijos y continuar hacia abajo. Diremos que el nodo modificado, se ha ***hundido*** hasta su nueva posición.

En las páginas 187 a 190 del libro, podemos ver los procedimientos para manejar un montículo que está implementado en una matriz *T*.

```
procedimiento modificar-montículo(T[1..n],i,v)
   { T[1..n] es un montículo. T[i] recibe
   el valor v, y se vuelve a establecer
   la propiedad del montículo}

procedimiento hundir(T[1..n],i)
   { hunde el nodo i}

procedimiento flotar(T[1..n],i)
   { flota el nodo i}

funcion buscar-max(T[1..n])
   { proporciona el mayor valor del
   montículo, que evidentemente es T[1]}

procedimiento borrar-max(T[1..n])
   { borra el mayor valor, y restaura la
   propiedad del montículo}

procedimiento añadir-nodo(T[1..n],v)
   { Añade un elemento cuyo valor es v al
   montículo T[1..n] y restaura la
   propiedad del montículo en T[1..n+1]}

procedimiento crear-monticulo(T[1..n])
   { Transforma la matriz T[1..n] en un
   montículo}
```

Éstas son exactamente las operaciones que se necesitan para implementar eficientemente las ***listas de prioridad dinámica***. Esto resulta especialmente útil en las simulaciones por computadora y en el diseño de planificadores para sistemas operativos. Sin embargo, los montículos no tienen una buena manera de buscar un cierto elemento dentro de él. Tampoco hay forma eficiente de fusionar dos montículos.

Williams inventó el montículo para que sirviera como estructura de datos subyacente al siguiente algoritmo de ordenación:

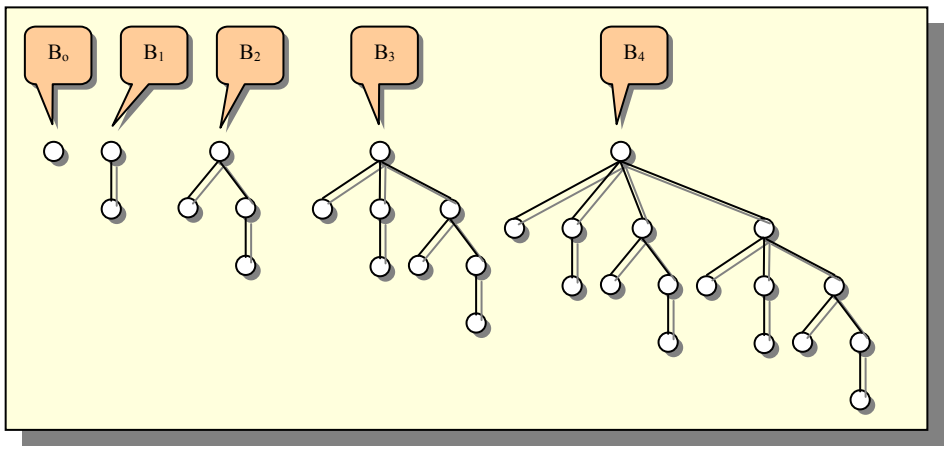
```
procedimiento ordenacion_por_monticulo(T[1..n])
    {T es la matriz que hay que ordenar}
        crear-monticulo(T)
        para i←n bajando hasta 2 hacer
            intercambiar T[1] y T[i]
            hundir(T[1..i-1]1)
```

Este algoritmo requiere un tiempo que está en O(*n* log *n*) para ordenar *n* elementos.

5.8 Montículos binominales.

En un *montículo ordinario* que contenga *n* elementos, buscar el mayor de ellos requiere un tiempo que está en O(1). Borrar el mayor elemento, o insertar un nuevo elemento, requiere un tiempo que está en O(log *n*). Sin embargo, fusionar dos montículos que tengan entre los dos *n* elementos, requiere un tiempo que está en O(*n*).

En un ***montículo binominal***, la búsqueda del mayor elemento sigue necesitando un tiempo que está en O(1), y el borrado del mayor elemento sigue requiriendo un tiempo en O(log *n*). Sin embargo, la fusión de dos de estos montículos sólo requiere un tiempo en O(log *n*), y la inserción de un nuevo elemento –siempre y cuando se considere el *tiempo amortizado*, y no el coste en sí de cada operación– solamente requiere un tiempo en O(1).



Estos montículos se basan en los árboles de su mismo nombre. El *i*-ésimo ***árbol binominal*** B_i, con $i \geq 0$, se define recursivamente como aquel que consta de un nodo raíz con *i* hijos, en donde el *j*-ésimo hijo, $1 \leq j \leq i$, es a su vez un árbol binominal B_{j-1}. En la figura se muestra desde B_0 a B_4.

5.9 Estructuras de conjuntos disjuntos (partición).

Supongamos que se tienen *N* objetos numerados de 1 a *N*. deseamos agrupar estos objetos en ***conjuntos disjuntos***, de tal forma que en todo momento cada objeto se encuentre exactamente en un conjunto. En cada conjunto, se selecciona un miembro que servirá como *rótulo* (por ejemplo, el elemento más pequeño). Inicialmente, esos *N* objetos se encuentran en *N* conjuntos diferentes, cada uno de los cuales contiene exactamente un objeto. Posteriormente, se ejecuta una secuencia de operaciones que pueden ser de dos tipos:

- Dado un objeto, **buscar** el conjunto que lo contiene.
- Dados dos rótulos, **fusionar** el contenido de los dos conjuntos correspondientes y seleccionar un rótulo para el conjunto combinado.

Un conjunto es pues, una colección de elementos de información sin orden alguno. Se utilizan cuando la pertenencia de un elemento a un conjunto, es la información más relevante de la colección de elementos que estamos considerando. No es aconsejable su uso si se necesita darles algún orden, o si necesitamos referenciar a un elemento en una posición dada.

Implementaciones:

- **Lista enlazada**.- Permite que el conjunto tenga un tamaño ilimitado, pero algunas operaciones tienen un elevado coste, al tener que recorrer toda la lista.
- **Vectores**.- Necesitamos conocer el número máximo de elementos albergados. Tiene los mismos problemas que la implementación anterior.
- **Vector característico**.- Para cada elemento, existe un bit que dice si el elemento existe o no en el conjunto. Limita el tamaño del conjunto, pero permite operaciones mucho más eficientes.

6 ALGORITMOS VORACES

Los algoritmos voraces suelen ser los más fáciles. El enfoque que aplican *es miope*, es decir, toman decisiones basándose en la información de que disponen de modo inmediato, sin tener en cuenta los efectos. Por tanto, resultan fáciles de inventar, fáciles de implementar, y cuando funcionan, son eficientes. Sin embargo, no sirven para todos los problemas.

Un algoritmo voraz funciona seleccionando el arco o la tarea que parezca más prometedora en un determinado instante, y nunca reconsidera su decisión.

6.1 Dar la vuelta (1).

Supongamos que disponemos de las siguientes monedas: 100 pesetas, 25 pesetas, 10 pesetas, 5 pesetas y 1 peseta. Nuestro problema es diseñar un algoritmo para pagar una cierta cantidad a un cliente utilizando el menor número posible de monedas.

Intuitivamente, utilizamos un algoritmo voraz: empezamos por nada, y en cada fase vamos añadiendo a las monedas seleccionadas, la mayor posible sin sobrepasar la cantidad que nos resta por pagar. Por ejemplo, para pagar 289, usaremos: 100, 100, 25, 25, 25, 10, 1, 1, 1, 1.

El algoritmo será por tanto:

```
funcion devolver_cambio(n): conjunto de monedas
    {Da el cambio de n unidades. La constante C
    especifica las monedas disponibles}
    const C = {100,25,10,5,1}
    S ← ∅ {S.- conjunto solución}
    s ← 0
    mientras s ≠ n hacer
            x ← el mayor elemento de C tal que s + x ≤ n
            si no existe ese elemento entonces
                    devolver "no encuentro la solución"
            S ← S ∪ {una moneda de valor x}
            s ← s + x
    devolver S
```

Es fácil convencerse (aunque es difícil probarlo formalmente), que este algoritmo siempre produce una solución óptima. Sin embargo, con una serie de valores diferentes, o si el suministro de alguna de las monedas está limitado, el algoritmo voraz puede no funcionar.

Este algoritmo es "voraz", porque en cada paso selecciona la mayor moneda, sin preocuparse de lo correcto de esta decisión a la larga.

6.2 Características generales de los algoritmos voraces.

Los algoritmos voraces:

- Se aplican a problemas de *optimización* (se necesita una forma óptima de resolver el problema).
- Consisten en ir seleccionando elementos de un conjunto de candidatos que se van incorporando a la solución.
- Existe una función que comprueba si un cierto conjunto de candidatos constituye una *solución*.
- Hay una segunda función que comprueba si un cierto conjunto de candidatos es *factible*, esto es, si es posible o no completar el conjunto añadiendo otros candidatos.
- Hay otra función más, la *función de selección*, que indica en cualquier momento cuál es el elemento más prometedor de los candidatos restantes. Es crucial determinar la función de selección apropiada que nos asegure que la solución obtenida es óptima.
- Se caracterizan porque nunca se deshace una decisión ya tomada: los candidatos desechados no vuelven a ser considerados y los incorporados a la solución permanecen en ella hasta el final del algoritmo.

En notación algorítmica puede escribirse así:

```
fun voraz(C:conjunto) dev (S:conjunto)
    S ← ∅
    mientras ¬solucion(S) ∧ C ≠ ∅ hacer
        x ← seleccionar(C)
        C ← C \ {x}
        si completable(S ∪ {x}) entonces
            S ← S ∪ {x}
        fsi
    fmientras
    dev S
ffun
```

Las funciones que hay que particularizar para cada problema serán:

1. *solucion*(*S*).- Decide si el conjunto *S*, es o no una solución.
2. *seleccionar*(*C*).- De entre los candidatos restantes, elige el más prometedor según el criterio que recoge la función *objetivo*. Esta función es la que determina el comportamiento del algoritmo.
3. *completable*.- Determina si un conjunto de elementos (pasado como argumento), puede llegar a una solución o no. Esta función también suele llamarse *factible*.

Está clara la razón por la que este esquema se llama *voraz*: A cada paso, el procedimiento selecciona el mejor bocado que pueda tragar.

Volviendo al ejemplo *dar vuelta*, visto anteriormente:

- *C* = {100,25,10 5 1}, con tantas monedas de cada valor que nunca las agotamos.
- *solución*(*S*).- comprueba si el valor de las monedas seleccionadas es *exactamente* el valor que hay que pagar.
- *completable*, o *factible*.- Nos dice si el valor de las monedas de un conjunto *no sobrepasa* la cantidad que hay que pagar.
- *seleccionar*(*C*).- Toma la moneda de valor más alto que quede en el conjunto de candidatos.

En general, para determinar los aspectos de la resolución de un problema, es recomendable seguir ciertos pasos, que particularizados para el esquema ***voraz*** son:

Elección del esquema.

Para aplicar el esquema voraz, el problema ha de ser de optimización. Se debe distinguir un conjunto de candidatos y que la solución pase por ir escogiéndolos o rechazándolos.

La función de selección debe asegurar que la solución alcanzada es la óptima. Si no, puede que hayamos confundido el problema con uno de exploración de grafos. Por ejemplo, en el problema de las monedas, la solución es óptima tomando el conjunto de monedas españolas (cuando se utilizaban pesetas, no euros como ahora), es decir, *C* = {100,25,10 5 1}. Sin embargo, si tomamos las monedas inglesas en peniques *C* = {30,24,12,6,3,1}, para un cambio de 48 peniques, nuestro algoritmo nos daría un conjunto solución *S* = {30,12,6}, pero la solución óptima sería *S* = {24,24}.

Identificación con el problema.

Es necesario determinar quienes son los candidatos, la función de selección, la función solución y la función factible.

Estructuras de datos.

Hay que especificar al menos, la estructura de datos correspondiente a los elementos (candidatos) que componen el problema. Los candidatos se pueden representar normalmente en un conjunto. La solución normalmente también, aunque en ocasiones es necesario conocer el orden en el que se han introducido en la solución. En este caso, es mejor utilizar una lista.

Algoritmo completo.

En principio, casi siempre es posible dar el algoritmo completo, simplemente con especificar las funciones *solución*, *seleccionar* y *factible*.

Estudio del coste.

Los algoritmos voraces son iterativos, por lo que evaluar su coste es sencillo. Normalmente el coste del bucle voraz es el que determina el coste global del algoritmo.

Aquí es importante decidir la implementación correcta de las estructuras de datos, ya que elegir el candidato correcto, puede ser más o menos costoso utilizando conjuntos, montículos, vectores...etc

Probablemente, la mejor opción sea el montículo, ya que seleccionar el mayor (o menor) elemento, tiene coste constante; y retirarlo del conjunto y restaurar el montículo tiene coste logarítmico.

Demostración de optimalidad.

Una argumentación clara debe ser suficiente. Es interesante tener presentes las técnicas de demostración *reducción al absurdo* e *inducción matemática*.

6.3 Grafos: Árboles de recubrimiento mínimo.

Planteamiento del problema:

> Sea $G=<N,A>$ un grafo conexo no dirigido. Cada arista de A, posee un *coste* no negativo. El problema consiste en hallar un subconjunto T de las aristas de G tal que, utilizando solamente las aristas de T, todos los nodos deben quedar conectados. Además, el coste total debe ser el menor posible. Es decir, la suma de todos los costes de T, debe ser mínima. En caso de varias soluciones del mismo coste, se prefiere la de menor número de aristas.

Un grafo conexo con n nodos, debe tener al menos n-1 aristas. Si tiene más, contiene al menos un ciclo, por lo que es posible eliminar alguna arista (siempre que la arista eliminada forme parte de un ciclo). Tomaremos por tanto, que la solución óptima tendrá n-1 aristas (siendo n el número de nodos) y será por tanto un árbol.

El grafo G' se denomina ***árbol de recubrimiento mínimo*** para el grafo G. Este problema tiene muchas aplicaciones. Por ejemplo, si los nodos son ciudades, y el coste de las aristas representa el coste de instalar una línea telefónica, podremos hallar la red telefónica más barata que da servicio a todas las ciudades.

En principio, podemos seguir dos líneas de actuación, que nos conducirán a dos algoritmos distintos, y que ambos funcionan correctamente. Una técnica es comenzar con un conjunto T vacío, al que vamos añadiendo la arista que en cada momento sea la más "corta", y no haya sido rechazada. Otra técnica es seleccionar un nodo, y construir un árbol a partir de él, seleccionando en cada etapa la arista más corta posible que pueda seguir extendiendo el árbol hacia un nodo adicional.

La forma de aplicar ambos algoritmos al esquema general es:

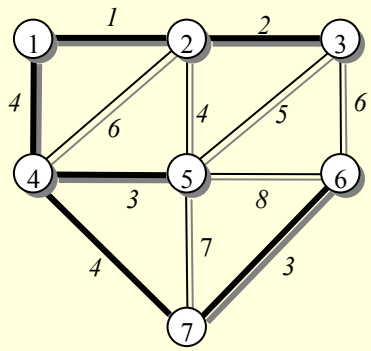
- Los candidatos son las aristas de G.
- Un conjunto de aristas es solución si constituye un árbol de recubrimiento para los nodos de N.
- Un conjunto de aristas es factible (completable) si no contiene ningún ciclo.

Algoritmo de Kruskal.

Inicialmente, T está vacío. Mientras no se encuentre la solución, el grafo parcial está formado por los nodos de G y las aristas de T y consta de varias componentes conexas (al principio, contiene tantas componentes conexas como nodos hay en el grafo). Examinamos las aristas de G por orden creciente. Si una arista une dos componentes conexas distintas, entonces se añade a T. Las dos componentes conexas se unen formando una única componente conexa. Si la arista elegida no une dos componentes conexas distintas, se rechaza (forma parte de un ciclo). El algoritmo se detiene cuando solo hay una única componente conexa, es decir, cuando se han tomado n-1 aristas.

Podemos seguir el algoritmo con el ejemplo de la figura, en el que se muestra el estado del conjunto T en cada uno de los pasos del bucle voraz.

```
funcion Kruskal(G = <N,A> : grafo ; longitud : A → ℝ+) : conjunto de aristas
    { INICIALIZACIÓN }
    Ordenar A por longitudes crecientes
    n ← el número de nodos que hay en N
    T ← ∅
    Iniciar n conjuntos, cada uno con un elemento distinto de N.
    {BUCLE VORAZ}
    repetir
        e ← {u,v} ← arista más corta, aún no considerada.
        compu ← buscar(u)
        compv ← buscar(v)
        si compu ≠ compv entonces
            fusionar(compu,compv)
            T ← T ∪ {e}
    hasta que T contenga n-1 aristas
devolver T
```



Paso	Arista considerada	Componentes conexas
Iniciación	--	{1} {2} {3} {4} {5} {6} {7}
1	{1,2}	{1,2} {3} {4} {5} {6} {7}
2	{2,3}	{1,2,3} {4} {5} {6} {7}
3	{4,5}	{1,2,3} {4,5} {6} {7}
4	{6,7}	{1,2,3} {4,5} {6,7}
5	{1,4}	{1,2,3,4,5} {6,7}
6	{2,5}	rechazado
7	{4,7}	{1,2,3,4,5,6,7}

Para implementar el algoritmo, tenemos que manejar un cierto número de conjuntos (los nodos de cada componente conexa). Por tanto, es preciso efectuar rápidamente dos operaciones: *buscar*(*x*), que nos dice en qué componente conexa se encuentra el nodo *x*, y *fusionar*(*A*,*B*) para fusionar dos componentes conexas. Para este algoritmo, es preferible representar el grafo como un vector de aristas con sus longitudes asociadas, y no como una matriz de distancias.

El tiempo total para el algoritmo está en $\Theta(a \log n)$, considerando un grafo con n nodos y a aristas.

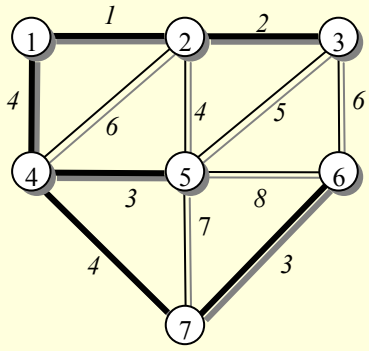
Algoritmo de Prim.

En el algoritmo de Kruskal, se obtiene un bosque de árboles, que crece al azar, hasta que finalmente todas las componentes del bosque se fusionan en un único árbol. En el ***algoritmo de Prim***, el árbol de recubrimiento mínimo crece de forma natural comenzando por una raíz arbitraria. En cada fase, se añade una nueva rama al árbol ya construido; el algoritmo se detiene cuando se han alcanzado todos los nodos.

```
funcion Prim(G = <N,A> : grafo ; longitud : A → ℝ+) : conjunto de aristas
    { INICIALIZACIÓN }
    T ← ∅
    B ← {un miembro arbitrario de N}
    mientras B≠N hacer
        buscar e = {u,v} de longitud mínima tal que
            (u ∈ B) y (v ∈ N\B)
        T ← T ∪ {e}
        B ← B ∪ {v}
    devolver T
```

Sea B un conjunto de nodos, y sea T un conjunto de aristas. Inicialmente, B contiene un único nodo arbitrario, y T está vacío. En cada paso busca la arista más

corta posible, de forma que un extremo esté en algún nodo de *B*, y el otro extremo "salga" de *B*. Es decir, una arista {*u*,*v*} tal que (*u* ∈ *B*) y (*v* ∈ *N**B*). Una vez encontrada esta arista, se añade *v* a *B*, y {*u*,*v*} a *T*.



Paso	{*u*,*v*}	*B*
Iniciación	--	{1}
1	{1,2}	{1,2}
2	{2,3}	{1,2,3}
3	{1,4}	{1,2,3,4}
4	{4,5}	{1,2,3,4,5}
5	{4,7}	{1,2,3,4,5, 7}
6	{7,6}	{1,2,3,4,5,6,7}

En el ejemplo anterior, cuando se detiene el algoritmo, T contiene las aristas seleccionadas: {1,2}, {2,3}, {1,4}, {4,5}, {4,7} y {7,6}.

El algoritmo de Prim requiere un tiempo que está en **$\Theta(n^2)$**.

Vimos que el algoritmo de Kruskal estaba en $\Theta(a \log n)$, pero la elección de uno u otro algoritmo, dependerá de la configuración del grafo:

- ☞ En un grafo denso, *a* tiende a $n(n\text{-}1)/2$. En este caso, el algoritmo de Prim puede ser mejor, ya que el de Kruskal se encuentra en $\Theta(n^2 \log n)$.
- ☞ Si el grafo es disperso, *a* tiende a *n*, y entonces es preferible el de Kruskal, porque requiere un tiempo en $\Theta(n \log n)$.

6.4 Grafos: Caminos mínimos.

Planteamiento del problema:

> Sea *G*=<*N*,*A*> un grafo dirigido. Cada arista de *A*, posee una *longitud* no negativa. Se toma uno de los nodos como *origen*. El problema consiste en determinar la longitud del camino mínimo que va desde el *origen* a cada uno de todos los demás nodos del grafo.

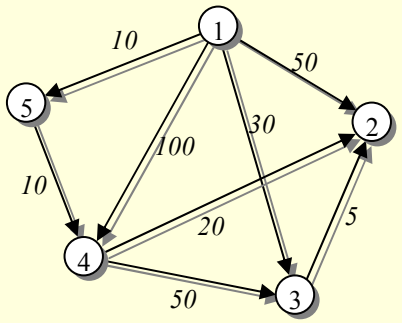
Este problema se resuelve mediante el algoritmo voraz que recibe frecuente el nombre de **algoritmo de Dijkstra**. Utiliza dos conjuntos de nodos, *S* y *C*. En todo momento, *S* contiene los nodos que ya han sido seleccionados y de los que ya conocemos su distancia al origen. El conjunto *C* contiene los demás nodos. En todo momento, tenemos la propiedad invariante *N* = *S* ∪ *C*. Al principio, *S* solo contiene el nodo origen, y en cada paso, se selecciona aquel nodo de *C* cuya distancia al origen sea mínima, y se lo añadimos a *S*.

Diremos que un camino desde el origen hasta algún nodo, es *especial*, si todos los nodos intermedios a lo largo del camino, pertenecen a *S*. Hay una matriz D, que contiene la longitud del camino especial más corto hasta cada nodo.

Suponemos que todos los nodos están numerados de 1 a *n*. Suponemos también, sin pérdida de generalidad, que el *nodo origen* es el nodo 1. Suponemos también, que la matriz *L*, da la longitud de todas las aristas dirigidas (*infinito* si la arista no existe en el grafo).

```
funcion Dijkstra(L[1..n,1..n]) : matriz [2..n]
    matriz D[2..n]
    { INICIALIZACIÓN }
        C ← {2,3,...,n} {S = N\C   SOLO EXISTE IMPLÍCITAMENTE}
        para i ← 2 hasta n hacer D[i] ← L[1,i]
    {BUCLE VORAZ}
        repetir n-2 veces
            v ← algún elemento de C que minimiza D[v]
            C ← C\{v}  {IMPLÍCITAMENTE S ← S∪{v}}
            para cada w ∈ C hacer
                D[w] ← mín(D[w] , D[v]+L[v,w])
devolver D
```

Vemos un ejemplo de funcionamiento en la siguiente figura:



Paso	*v*	*C*	*B*
Iniciación	--	{2,3,4,5}	[50,30,100,10]
1	5	{2,3,4}	[50,30,20,10]
2	4	{2,3}	[40,30,20,10]
3	3	{2}	[35,30,20,10]

Para determinar no solamente la longitud de los caminos, sino también por dónde pasan, es necesario otra matriz, en la que vamos anotando para cada nodo, cual es su predecesor en el camino al nodo origen.

Para un grafo con n nodos y a aristas, el tiempo requerido por esta versión del algoritmo, está en $\Theta(n^3)$. Si $a<<n^2$, podríamos introducir algunas mejoras en el algoritmo, al tener la esperanza de que muchas de las aristas de la matriz, tendrían como valor ∞, es decir, no existirían, por lo que nos evitaríamos el tener que examinarlas en el bucle **para** interno.

6.5 El problema de la mochila (1).

Planteamiento del problema:

Nos dan n objetos y una mochila. Para i=1,2,...,n, el objeto i tiene un peso positivo w_i, y un valor positivo v_i. La mochila puede llevar un peso que no sobrepase W. Nuestro objetivo es llenar la mochila de forma que maximicemos el valor total de los objetos. No es necesario meter objetos completos, sino que se permite meter una fracción x_i de cada objeto. De esta forma, el objeto i contribuye con $x_i w_i$ al peso total, y con $x_i v_i$ al valor total.

Se puede plantear el problema como:

$$\text{maximizar } \sum_{i=1}^{n} x_i v_i \text{ con la restricción } \sum_{i=1}^{n} x_i w_i \le W$$

Utilizaremos un algoritmo voraz en el que los *candidatos* son los diferentes objetos, la solución es un vector (x_1,..., x_n), que nos dice qué fracción de cada objeto introducimos en la mochila. La *función objetivo* es el valor total de los objetos, y nos queda por definir cuál será la *función de selección*.

Podríamos optar por tres posibilidades:

1. Escoger el objeto con más valor. (max (v_i)).
2. Escoger el objeto con menos peso.(min (w_i)).
3. Escoger el objeto con mayor valor por unidad de peso. (max(v_i/w_i)).

Se puede demostrar que la tercera opción obtiene un resultado óptimo.

Si los objetos se encuentran ordenados por orden creciente de v/w, entonces el tiempo está en **O(*n*)**. Si incluimos la ordenación de los elementos, se encuentra por tanto en **O(*n* log *n*)**.

```
funcion mochila(w[1..n], v[1..n], W) : matriz [1..n]
    {INICIALIZACIÓN}
        para i = 1 hasta n hacer x[i] ← 0
        peso ← 0
    {BUCLE VORAZ}
        mientras peso < W hacer
            i ← el mejor objeto restante (mejor relación v/w)
            si peso + w[i] ≤ W entonces
                x[i] ← 1
                peso ← peso + w[i]
            sino x[i] ← (W-peso)/ w[i]
                peso ← W
devolver x
```

6.6 Planificación.

En esta ocasión, se presentan dos problemas que conciernen a la forma óptima de planificar tareas en una sola máquina. En el primero, el problema consiste en minimizar el tiempo medio que interviene cada tarea en el sistema. En el segundo, las tareas tienen un plazo fijo de ejecución, y cada una aporta unos ciertos beneficios sólo si se acaba en el plazo previsto; nuestro objetivo es maximizar la rentabilidad.

Minimización del tiempo en el sistema.

Los ejemplos reales serían un servidor (procesador, surtidor de gasolina, cajero automático...etc) que tiene que dar servicio a n clientes, conociendo de antemano el tiempo requerido por cada cliente. Mientras se da servicio a un cliente, los que no han sido atendidos esperan.

Se trata de minimizar $T = \sum_{i=1}^{n}$ (tiempo en el sistema para el cliente i)

Por ejemplo, si tenemos 3 clientes para servir, los cuales requieren un tiempo t_1=5, t_2=10 y t_3=3; tendremos seis posibilidades de orden en el servicio (123, 132, 213, 231, 312, 321). Calculando los diferentes tiempos, vemos que el menor corresponde a:

Cliente 3 = 3.
Cliente 1 = 3+5 = 8.
Cliente 2 = 8+10 = 18.
⟶ Haciendo un total de 3+8+18 = 29

La planificación óptima se obtiene cuando se sirve a los clientes por orden creciente de tiempos de servicio.

La implementación de este algoritmo es trivial. En esencia, se ordenan los clientes por orden de tiempo no decreciente, lo cual requiere un tiempo que está en **O(*n* log *n*)**.

El problema se puede generalizar para s servidores.

Planificación con plazo fijo.

Tenemos que ejecutar un conjunto de n tareas, cada una de las cuales requiere un tiempo unitario. En cualquier instante T=1,2,3..., podemos ejecutar únicamente una tarea. La tarea i nos produce unos beneficios $g_i > 0$ sólo en el caso de que sea ejecutada en un instante anterior a d_i.

Por ejemplo, con n=4, y los siguientes datos:

i	1	2	3	4
g_i	50	10	15	30
d_i	2	1	2	1

Las diferentes planificaciones con sus beneficios correspondientes serían:

secuencia	beneficio
1	50
2	10
3	15
4	30
1,3	65
2,1	60
2,3	25
3,1	65
4,1	80
4,3	45

El resto de combinaciones no aparecen por no ser posibles, es decir, por ejemplo, la secuencia 3,2 no es considerada porque la tarea 2 se ejecutaría fuera de plazo. La más beneficiosa sería efectuar la tarea 4, y después la 1.

Para implementar el algoritmo, supondremos que las tareas están organizadas de forma que $g_1 \geq g_2 \geq \ldots \geq g_n$. Supondremos además que $n > 0$ y también que todos los $d_i > 0$. Utilizaremos también un espacio adicional al comienzo de los vectores d y j (plazos y solución), que utilizaremos como *centinela*.

El peor caso de este algoritmo es cuando se han de considerar todas las tareas, estando el tiempo en **Ω(n^2)**.

```
funcion secuencia(d[0..n]) : k,matriz[1..k]
    matriz j[0..n] {LA SOLUCIÓN SE VA CONSTRUYENDO EN j. k NOS DICE CUANTAS
                    TAREAS HAY YA PLANIFICADAS}
    d[0] ← j[0] ← 0  {CENTINELAS}
    k ← j[1] ← 1   {LA TAREA 1 SIEMPRE SE SELECCIONA}
    {BUCLE VORAZ}
    para i ← 2 hasta n hacer   {ORDEN DESCENDIENTE DE g}
        r ← k
        mientras d[j[r]] > máx(d[i],r) hacer r ← r-1
        si d[i] > r entonces
        para m ← k paso –1 hasta r+1 hacer j[m+1] ← j[m]
        j[r+1] ← i
        k ←k+1
devolver k , j[1..k]
```

Se obtiene un algoritmo más eficiente cuando se utiliza una técnica distinta para verificar si un conjunto de tareas es factible. Se empieza por una planificación vacía, se considera cada tarea sucesivamente, y se añade a la planificación que se está construyendo en el momento más tardío posible, pero no antes de su fecha final. Si no se puede planificar una tarea antes de su plazo, entonces el conjunto *J* no es factible.

El primer algoritmo consiste en ir añadiendo tareas a la solución, en orden decreciente de valor. Cada solución parcial es una secuencia compacta de tareas. De esta forma, la primera tarea (la de más valor) siempre se añadirá a la secuencia resultado, independientemente de cuándo caduque. ¿Y qué se hace con el resto de las tareas (de las 2ª en adelante, por orden decreciente de valor)?: se le busca una posición en la secuencia resultado; si tal hueco o posición existe, se añade a dicha secuencia; si no existe, dicho elemento es descartado. Para buscar tal posición, lo que se hace es recorrer la secuencia que llevamos hasta el momento (que inicialmente contiene al menos el elemento de mayor valor), de derecha a izquierda. Si la secuencia que llevamos calculada en un momento determinado tiene, p.ej. 3 elementos, y el que vamos a añadir (Tarea 8, p.ej.) caduca en tiempo 5, pues se puede añadir sencillamente al final de dicha secuencia, en la posición 4 (ya que caduca después). Si, p.ej. caduca en tiempo 2, pues no es posible añadirla al final, pero ¿se podrá añadir por en el medio, desplazando algunas de las tareas ya añadidas una unidad temporal a la derecha en su secuencia?. Pues depende; supongamos el caso de la figura:

Posición	1	2	3	4	5	6
Tareas	5	3	1			
Caducidad	1	3	4			

En un caso como este, si la tarea a añadir caduca en tiempo 2, si desplazamos las tareas situadas a la derecha de la secuencia que caducan después que ella (caducidad > 2), vemos que no hay problema, pues la tarea 1 caduca en 4 y ahora mismo se ejecuta en 3, por lo que si la desplazamos hacia la derecha se ejecutará en 4, que sigue estando en los límites de su tiempo de ejecución. Lo mismo ocurre con la tarea 3, luego es posible abrir un hueco en 2 para albergar la nueva tarea. Nótese que la tarea 5 no puede desplazarse a la derecha, pues ya se ejecuta en su límite de tiempo: el 1. De esta forma, quedaría:

Posición	1	2	3	4	5	6
Tareas	5	8	3	1		
Caducidad	1	2	3	4		

Por el contrario, si en nuestro proceso de búsqueda de hueco, necesitamos desplazar a la derecha una tarea, pero ésta ya está en su límite de tiempo, concluimos en que dicho hueco no se puede crear para la nueva tarea y ésta no puede añadirse a la secuencia.

El segundo algoritmo, parece mucho más enrevesado, pero en realidad su lógica es muy sencilla: las tareas se ordenan por orden decreciente de valor, y se añaden a la secuencia lo más tarde posible. P.ej., suponiendo las tareas de la figura 6.9 en la pág. 237, la secuencia quedaría:

i=1

Posición	1	2	3	4	5	6
Tareas			1			
Caducidad			3			

i=2

Tareas	2		1			
Caducidad	1		3			

i=3 (se queda igual)

Tareas	2		1			
Caducidad	1		3			

i=4

Tareas	2	4	1			
Caducidad	1	3	3			

7 DIVIDE Y VENCERÁS

Es una técnica que consiste en descomponer el problema en un cierto número de subcasos más pequeños de ese problema, que resolveremos sucesiva e independientemente. Después combinaremos todas las soluciones obtenidas para obtener la solución del problema original.

7.1 Introducción: Multiplicación de enteros muy grandes.

Para la multiplicación, vimos en el primer capítulo dos métodos, que nos daban unos costes del orden de n^2. Adelantábamos un método de divide y vencerás, que es el que pasamos a describir. Utilizaremos el mismo ejemplo que en el capítulo 1, es decir, multiplicar 981×1234.

En primer lugar, añadiremos un cero a la izquierda del primer número, para que tengan las mismas cifras, y dividiremos cada número en dos, obteniendo w, x, y, z.

$$w=09,\ x=81; \qquad y=12,\ z=34$$

Podemos entonces transformar los números originales en:

$$981 = 10^2 w + x \qquad 1234 = 10^2 y + z$$

Por lo tanto, el producto requerido se puede expresar como:

$$981 \times 1234 = (10^2 w + x) \times (10^2 y + z) = 10^4 wy + 10^2(wz+xy) + xz$$

Se reduciría por tanto a 4 multiplicaciones de números de dos cifras. Pero ¿podríamos obtener la solución con solo tres multiplicaciones?. La solución sería:

$(wz+xy) = (r - p - q)$, siendo:

$p = wy = 09 \times 12 = 108$
$q = zx = 81 \times 34 = 2754$
$r = (w+x) \times (y+z) = 90 \times 46 = 4140$

Y como p y q , ya las teníamos que calcular de todas formas, solo hay que hacer una multiplicación más

$981 \times 1234 = 10^4 p + 10^2(r-p-q) + q = 1.080000 + 127.800 + 2.754 =$ **1.210.554**

De esta manera, el producto se puede reducir a 3 multiplicaciones de números de 2 cifras, junto con un cierto número de desplazamientos (multiplicaciones por potencias de 10), adiciones y sustracciones. Entonces, ¿merece la pena hacer cuatro sumas más para evitar una multiplicación?. La respuesta es que para números pequeños no, pero sí cuanto mayores sean los números a multiplicar.

7.2 El caso general.

El caso general de los algoritmos divide y vencerás es:

```
fun divide_y_venceras (problema)
    si suficientemente_simple (problema)
        entonces dev solucion_simple (problema)
    sino hacer
        {p1..pk} ← descomposicion (problema)
        para cada pi hacer
            si ← divide_y_venceras (pi)
        fpara
        dev combinacion (s1..sk)
    fsi
ffun
```

Las funciones que han de particularizarse son:

1. *suficientemente_simple*. - Decide si un problema está por debajo de un tamaño umbral o no.
2. *solucion_simple*.- Algoritmo utilizado para resolver los casos más sencillos por debajo del tamaño umbral.
3. *descomposicion*.- Descompone el problema en *k* subproblemas de tamaño menor.
4. *combinacion*.- Algoritmo que combina las soluciones a los subproblemas en la solución al problema del que provienen.

El número de subejemplares, *k*, suele ser pequeño e independiente del caso en particular que haya que resolver. Cuando *k*=1, no tiene mucho sentido justificar el nombre de *divide y vencerás* que se da a esta técnica. En estos casos, la técnica recibe el nombre de *reducción* (*simplificación*).

Para que *divide y vencerás* merezca la pena, tiene que ser posible descomponer el ejemplar en subejemplares, y tiene que ser posible componer las diferentes soluciones de forma bastante eficiente. También los ejemplares deben ser aproximadamente del mismo tamaño. En la mayoría de los algoritmos, se obtienen subejemplares de tamaño n/b, siendo n el tamaño original, y b una constante.

En general, para determinar los aspectos de la resolución de un problema, es recomendable seguir ciertos pasos, que particularizados para el esquema ***divide y vencerás*** son:

Elección del esquema.

Es fundamental ver claramente cómo un problema puede dividirse en subproblemas idénticos a los que pueda aplicarse exactamente el mismo procedimiento. Además, se debe tener una idea clara de cómo combinar las soluciones a los subproblemas.

Identificación con el problema.

Hay que especificar cómo se divide en subproblemas, y cómo se combinan las soluciones. Además hay que decidir cual será el tamaño umbral y qué función solucionará los problemas por debajo de ese umbral.

También hay que comentar cuál es el *preorden bien fundado* para los problemas, es decir, hay que garantizar que la forma de dividir en subproblemas conduce, efectivamente, a problemas más sencillos y que siempre termina en el tamaño trivial.

Estructuras de datos.

Hay que especificar en qué consiste un problema, y en su caso, los elementos del problema.

Algoritmo completo.

Casi siempre se reduce a especificar las funciones *suficientemente_simple, solucion_simple, descomposicion y combinacion*.

Estudio del coste.

Al tratarse de algoritmos recursivos, es necesario plantear la ecuación de recurrencia. Al estar dividiendo el problema, normalmente se podrá aplicar la fórmula para reducciones por división:

$$T(n) = \begin{cases} cn^k & , \text{si } 1 \le n < b \\ aT(n/b) + cn^k & , \text{si } n \ge b \end{cases} \Longrightarrow \quad T(n) \in \begin{cases} \Theta(n^k) & , \text{si } a < b^k \\ \Theta(n^k \log n) & , \text{si } a = b^k \\ \Theta(n^{\log_b a}) & , \text{si } a > b^k \end{cases}$$

Por lo tanto, hay que plantear la ecuación de recurrencia para $T(n)$ e identificar en ella las constantes a, b y k.

7.3 Búsqueda binaria.

En esencia es el algoritmo que usamos intuitivamente para buscar una palabra en un diccionario o un nombre en la guía telefónica. Probablemente se trate de la aplicación más sencilla de *divide y vencerás*. Tan sencilla que en realidad es más bien un caso de *reducción* (*simplificación*): la solución a un caso grande, se reduce a un único caso más pequeño (en esta ocasión de tamaño mitad).

Planteamiento del problema:

> Sea $T[1..n]$ una matriz ordenada por orden no decreciente; esto es, $T[i] \le T[j]$ siempre que $1 \le i \le j \le n$. Sea x un elemento. El problema consiste en buscar x en la matriz T, si es que está.
>
> Formalmente: buscamos i tal que $1 \le i \le n+1$ y $T[i-1] < x \le T[i]$, con la convención lógica de que $T[0] = -\infty$ y $T[n+1] = \infty$.

Una aproximación sencilla a este problema, es examinar secuencialmente todos los elementos de T hasta que encontremos x, o hasta que lleguemos al final, o bien hasta que encontremos un elemento que no sea menor que x.

Este algoritmo requiere un tiempo que está en $\Theta(r)$, en donde r es el índice que se devuelve. Esto está en $\Omega(n)$ en el caso peor, y en $O(1)$ en el caso mejor.

```
funcion secuencial(T[1..n],x)
    para i ← 1 hasta n hacer
        si T[i] ≥ x entonces devolver i
devolver n+1
```

```
funcion busquedabin(T[1..n] , x)
    si n = 0 o x > T[n] entonces
        devolver n+1
    sino
        devolver binrec(T[1..n] , x)


funcion binrec(T[i..j] , x)
{BÚSQUEDA BINARIA DE x EN LA SUBMATRIZ t[i..j]
  CON LA SEGURIDAD DE QUE T[i-1]< x ≤ T[j]}
    si i = j entonces
        devolver i
    k ← (i+j)÷2
    si x ≤ T[k] entonces
        devolver binrec(T[i..k] , x)
    sino
        devolver binrec(T[k+1..j] , x)
```

Para acelerar la búsqueda, deberíamos buscar en la primera mitad de la matriz o en la segunda. Y para saber en cual de las mitades buscar, compararemos x con uno de los elementos de la matriz.

Sea $k = \lceil n/2 \rceil$. Si $x \le T[k]$, podremos reducir la búsqueda a $T[1..k]$, en caso contrario, buscaremos en $T[k+1..n]$.

Utilizando las ecuaciones de resolución de recursividades, obtenemos que el tiempo que necesita este algoritmo, está en **$\Theta(\log m)$** aun en el caso peor, siendo $m = j-i+1$

Es fácil realizar una versión iterativa de este mismo algoritmo, la cual tienen también el mismo coste. Se usan dos índices (i,j), que van acotando las mitades de búsqueda.

```
funcion biniter(T[1..n] , x)
    si x > T[n] entonces devolver n+1
    i ← 1 ; j ← n
    mientras i < j hacer
        k ← (i+j)÷2
        si x ≤ T[k] entonces j ← k
        sino i ← k+1
devolver i
```

7.4 Ordenación.

Ya vimos dos soluciones para este problema que requerían un tiempo cuadrático (Ordenación por *selección* y ordenación por *inserción*), y un caso más que requería un tiempo que está en $\Theta(n \log n)$ (ordenación por *montículo*). Veremos ahora dos métodos que siguen el esquema de *divide y vencerás*, que son la ***ordenación por fusión***, y la ***ordenación rápida*** (***quicksort***).

Ordenación por fusión.

Se trata de dividir la matriz en dos partes (de tamaño lo más parecido posible), ordenar recursivamente esas mitades, y fusionarlas teniendo cuidado de mantener el orden en la fusión. Necesitamos un algoritmo eficiente para fusionar dos matrices *U* y *V*, en una matriz *T* cuya longitud sea la suma de las otras dos.

Es necesario disponer de un espacio adicional al final de las matrices, que nos servirá de centinela, y en el que colocaremos un valor que sea mayor que cualquiera de la matriz, el cual lo representaremos por ∞.

```
procedimiento fusionar(U[1..m+1] , V[1..n+1] , T[1..m+n])
    i , j ← 1
    U[m+1] , V[n+1] ← ∞
    para k ← 1 hasta m+n hacer
        si U[i] < V[i] entonces
            T[k] ← U[i] ; i ← i+1
        sino
            T[k] ← U[j] ; j ← j+1


procedimiento ordenarporfusion(T[1..n])
    si n es suficientemente pequeño entonces insertar(T)
    sino
        matriz U[1..1+⌊n/2⌋], V[1.. 1+⌊n/2⌋ ]
        U[1..⌊n/2⌋ ] ← T[1..⌊n/2⌋ ]
        V[1..⌈n/2⌉] ← T[1+⌊n/2⌋ ..n]
        ordenarporfusion(U[1..⌊n/2⌋ ])
        ordenarporfusion(V[1..⌈n/2⌉])
        fusionar(U,V,T)
```

Utilizamos la ordenación por inserción, como subalgoritmo básico para resolver los ejemplares de tamaño menor que el umbral. Para mejorar la eficiencia, se pueden hacer *U* y *V*, variables globales.

Analizando la correspondiente ecuación de recurrencia, vemos que se particulariza con los valores *a*=2 (dos llamadas recursivas), *b*=2 (el tamaño se divide por dos en cada llamada), y *k*=1 (el coste de las operaciones adicionales: separación de *T* en dos matrices, y *fusionar*, requieren un tiempo lineal). Por tanto, el coste está en **Θ(*n* log *n*)**.

A pesar de tener el mismo coste que la *ordenación por montículo*, tiene la desventaja de tener que utilizar un espacio significativamente mayor, mientras que la *ordenación por montículo*, puede ordenar *in situ*.

Es muy importante el hacer los dos subejemplares lo más parecidos en tamaño. Olvidarnos de equilibrar los tamaños de los subcasos puede ser desastroso para la eficiencia de un algoritmo obtenido por *divide y vencerás*.

```
procedimiento pivote(T[i..j] ; var l)
        {PERMUTA LOS ELEMENTOS DE LA MATRIZ T[i..j] Y PROPORCIONA UN
        VALOR l TAL QUE, AL FINAL:
                    i ≤ l ≤ j
                    T[k] ≤ p PARA TODO i ≤ k <l
                    T[l] = p
                    T[k] > p PARA TODO l < k ≤ j
              EN DONDE p ES EL VALOR INICIAL DE T[i]}
    p ← T[i]
    k ← i ; l ← j+1
    repetir k ← k+1 hasta que T[k] > p o k ≥ j
    repetir l ← l-1 hasta que T[l] ≤ p
    mientras k < l hacer
        intercambiar T[k] y T[l]
        repetir k ← k+1 hasta que T[k] > p
        repetir l ← l-1 hasta que T[l] ≤ p
    intercambiar T[i] y T[l]


procedimiento quicksort(T[i..j])
    si j-i es suficientemente pequeño entonces insertar(T[i..j])
    sino
        pivote(T[i..j] , l)
        quicksort(T[i..l-1])
        quicksort(T[l+1..j])
```

Ordenación rápida (Quicksort).

El algoritmo de ordenación inventado por **Hoare**, también sigue el esquema de *divide y vencerás*, pero a diferencia de *ordenar por fusión*, la mayor parte del trabajo no recursivo se utiliza para construir los subcasos y no para combinar sus soluciones.

En primer lugar, se selecciona como ***pivote*** uno de los elementos de la matriz. Se parte a ambos lados del *pivote*, y se desplazan los elementos para que queden los más pequeños que el *pivote* a su izquierda, y el resto a la derecha. Ambas partes se ordenan recursivamente en sucesivas llamadas.

Lo ideal sería utilizar como *pivote*, la **mediana** de la matriz, pero averiguar cual es, nos costaría mucho tiempo, así que seleccionamos un elemento arbitrario y confiamos en la suerte.

Primeramente, el procedimiento *pivote*, lo que hace es descomponer la submatriz *T*[*i..j*] empleando como pivote *p*=*T*[*i*]. Consiste en explorar la matriz por la izquierda hasta encontrar un valor mayor que el pivote, y por la derecha hasta encontrar un valor menor o igual al *pivote*. Entonces se intercambian los valores encontrados. Ese proceso sigue hasta que las dos búsquedas (izquierda y derecha) se encuentran. Para finalizar, se coloca el *pivote* en el punto de encuentro.

Para ordenar una matriz completa, hay que llamar al procedimiento *quicksort*(*T*[1..*n*]).

Quicksort es ineficiente si sucede que de forma sistemática, en la mayoría de las llamadas recursivas, los subejemplares están fuertemente desequilibrados. En el peor caso, se puede estar llamando sucesivamente a un subejemplar de tamaño 0, junto con otro de tamaño *n*-1, en cuyo caso requiere un tiempo que está en $\Omega(n^2)$. En término medio, los subejemplares no están demasiado desequilibrados, y la solución estaría en **O(*n* log *n*)**.

En la práctica, la constante oculta es más pequeña que las asociadas en *ordenación por montículo* o en *ordenación por fusión*.

7.5 Búsqueda de la mediana.

Sea *T*[1..*n*] una matriz de enteros, y sea *s* un entero entre 1 y *n*. Se define el *s*-ésimo menor elemento de *T* como aquel elemento que se encontraría en la posición *s*-ésima si se ordenara *T* en orden no decreciente. El problema de encontrar un *s* dado, se conoce como el *problema de selección*. En particular, se defina la **mediana** como su $\lceil n/2 \rceil$-ésimo elemento. Por ejemplo, la mediana de [3, 1, 4, 1, 5, 9, 2, 6, 5] es 4, puesto que hay cuatro elementos más pequeños que la mediana, y cuatro elementos mayores que ella.

Utilizaremos un algoritmo que halla la mediana en un tiempo lineal en el peor caso:

```
funcion pseudomediana(T[1..n])
    si n ≤ 5 entonces devolver medianadhoc(T)
    z ← ⌊n/5⌋
    matriz Z[1..z]
    para i ← 1 hasta z hacer Z[i] ← medianadhoc(T[5i-4..5i])
    devolver selección(Z,⌈z/2⌉)
```

Aquí, *medianadhoc* es un algoritmo diseñado especialmente para hallar la mediana de un máximo de 5 elementos, lo cual se puede hacer en un tiempo limitado superiormente por una constante, y *selección*(*Z*,$\lceil z/2 \rceil$) determina la mediana exacta de la matriz *Z*.

7.6 Multiplicación de matrices.

La definición de la multiplicación de dos matrices *A* y *B* de tamaño $n \times n$:

$$C_{ij} = \sum_{k=1}^{n} A_{ik} B_{kj}$$

nos da idea del algoritmo clásico para esta operación. Cada entrada de *C* se calcula en un tiempo que está en $\Theta(n)$, suponiendo que la suma y la multiplicación de escalares sean operaciones elementales. Como hay que calcular n^2 entradas, el algoritmo clásico estará en **$\Theta(n^3)$**.

A final de los 60, Strassen mejoró este algoritmo. La idea básica, es similar a la mejora que se obtuvo al multiplicar enteros grandes. Viendo el ejemplo con matrices de 2×2:

$$A = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \text{ y } B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix}$$

Consideremos las siguientes operaciones que necesitan sólo una única multiplicación cada una:

$m_1 = (a_{21}+a_{22}-a_{11})\ (b_{22}-b_{12}+b_{11})$
$m_2 = a_{11}\ b_{11}$
$m_3 = a_{12}\ b_{21}$
$m_4 = (a_{11}-a_{21})\ (b_{22}-b_{12})$
$m_5 = (a_{21}+a_{22})\ (b_{12}-b_{11})$
$m_6 = (a_{12}-a_{21}+a_{11}-a_{22})\ b_{22}$
$m_7 = a_{22}\ (b_{11}+b_{22}-b_{12}-b_{21})$

Se puede comprobar, que el producto matricial *AB*, está dado por la siguiente matriz, obtenida a partir de los valores *m* calculados anteriormente:

$$C = \begin{pmatrix} m_2 + m_3 & m_1 + m_2 + m_5 + m_6 \\ m_1 + m_2 + m_4 - m_7 & m_1 + m_2 + m_4 + m_5 \end{pmatrix}$$

Por tanto, es posible multiplicar dos matrices 2×2, empleando solamente siete multiplicaciones escalares y algunas sumas y restas. Evidentemente, al aumentar el tamaño de las matrices a multiplicar, la ventaja se vuelve mayor. Conseguimos así, un tiempo del algoritmo, que está en $O(n^{\lg 7}) = O(n^{2,8})$

7.7 Exponenciación.

Sean *a* y *n* dos enteros. Deseamos calcular $x=a^n$. si *n* es pequeño, el algoritmo evidente es:

```
funcion exposec(a,n)
    r ← a
    para i ← 1 hasta n-1 hacer r ← a × r
devolver r
```

Este algoritmo requiere un tiempo que está en Θ(*n*), siempre que consideremos las multiplicaciones como *operación elemental*, pero esto no está en la realidad, ya que a poco que aumentemos los operandos, las operaciones de multiplicación desbordan la palabra del ordenador, pasando a ser operaciones *no elementales*. Si tomamos *m*, como el tamaño de *a*, este algoritmo se ejecuta en un tiempo que está en $\Theta(m^2n^2)$. Y si utilizamos para las multiplicaciones, el algoritmo de divide y vencerás, obtenemos un tiempo en $\Theta(m^{lg3}n^2)$.

Una mejora del algoritmo, se obtiene al observar que $a^n = (a^{n/2})^2$ cuando *n* es par. Esto es interesante, porque $a^{n/2}$ se puede calcular cuatro veces más deprisa que a^n con *exposec*, y solo falta una elevación al cuadrado (una multiplicación) para obtener el resultado. El algoritmo sería:

```
funcion expoDV(a,n)
    si n=1 entonces devolver a
    si n es par entonces devolver [expoDV(a,n/2)]²
    devolver a × expoDV(a,n-1)
```

Con este algoritmo, y utilizando para las multiplicaciones, el algoritmo de divide y vencerás, obtenemos un tiempo en $\Theta(m^{lg3}n^{lg3})$.

Existe una versión iterativa del algoritmo:

```
funcion expoiter(a,n)
    i ← n; r ← 1; x ← a
    mientras i > 0 hacer
        si i es impar entonces r ← rx
        x ← x²
        i ← i ÷ 2
    devolver r
```

7.8 Ensamblando todas las piezas: Introducción a la criptografía.

En la aritmética modular, es razonable contar todas las multiplicaciones como de un mismo coste. Por ejemplo, $a^n \bmod z$. Si *x* e *y* son dos enteros entre 0 y *z*-1, y si *z* es un entero de tamaño *m*, entonces, la multiplicación modular *xy* mod *z*, necesita una multiplicación entera ordinaria de dos números de tamaño *m* como máximo, dando lugar a un número entero de tamaño 2*m* como máximo, seguida por división del producto por *z* (un entero de tamaño *m*), para calcular el resto de la división.

Utilizando las siguientes propiedades de la aritmética modular, podemos modificar el algoritmo de la sección anterior, para calcular $a^n \bmod z$.

- ☞ $xy \bmod z = [(x \bmod z) \times (y \bmod z)] \bmod z$

☞ $(x \bmod z)^y \bmod z = x^y \bmod z$

```
funcion expomod(a,n,z)
    {CALCULA an mod z}
    mientras i > 0 hacer
        si i es impar entonces r ← a mod z
        x ← x² mod z
        i ← i ÷ 2
devolver r
```

Este algoritmo requiere un número de multiplicaciones modulares que está en $\Theta(\log n)$. Por ejemplo, para calcular $a^n \bmod z$, suponiendo que a, n y z sean números de 200 dígitos, y suponiendo también que dos números de ese tamaño, se pueden multiplicar módulo z en un milisegundo, entonces, *expomod* lo calcula en menos de un segundo, mientras que esa misma operación, pero sin módulo z, por el algoritmo *exposec*, requeriría aproximadamente **¡179 veces la edad del universo!**, es decir, muchísimo.

Todo esto es impresionante, pero, ¿para qué diablos puede interesarle a alguien trabajar en aritmética modular?. La **criptografía** (*el arte y la ciencia de la comunicación secreta a través de canales poco seguros*), tiene la respuesta.

Pondremos un ejemplo:

> Alicia desea enviar un mensaje secreto m a Benito. Para evitar que nadie lea el mensaje, lo convierte en un *mensaje cifrado* c, que es un mensaje que depende del mensaje original m, y de una clave k, que se supone sólo conocen Alicia y Benito. Es necesario por tanto, que ambos comunicantes hayan establecido de antemano, y en secreto, esa clave k. Este sistema confía en que el *escucha espía*, pueda interceptar c, pero no conozca k, para que no pueda deducir m.

¿Pueden comunicarse secretamente Alicia y Benito, si no han convenido de antemano la clave?. La era de la *criptografía de clave pública* nació por una idea desarrollada por *Diffie*, *Hellman* y *Merkle*, y una solución encontrada por *Rivest*, *Shamir* y *Adleman*, que se conoce como ***sistema criptográfico RSA***, en alusión a sus inventores.

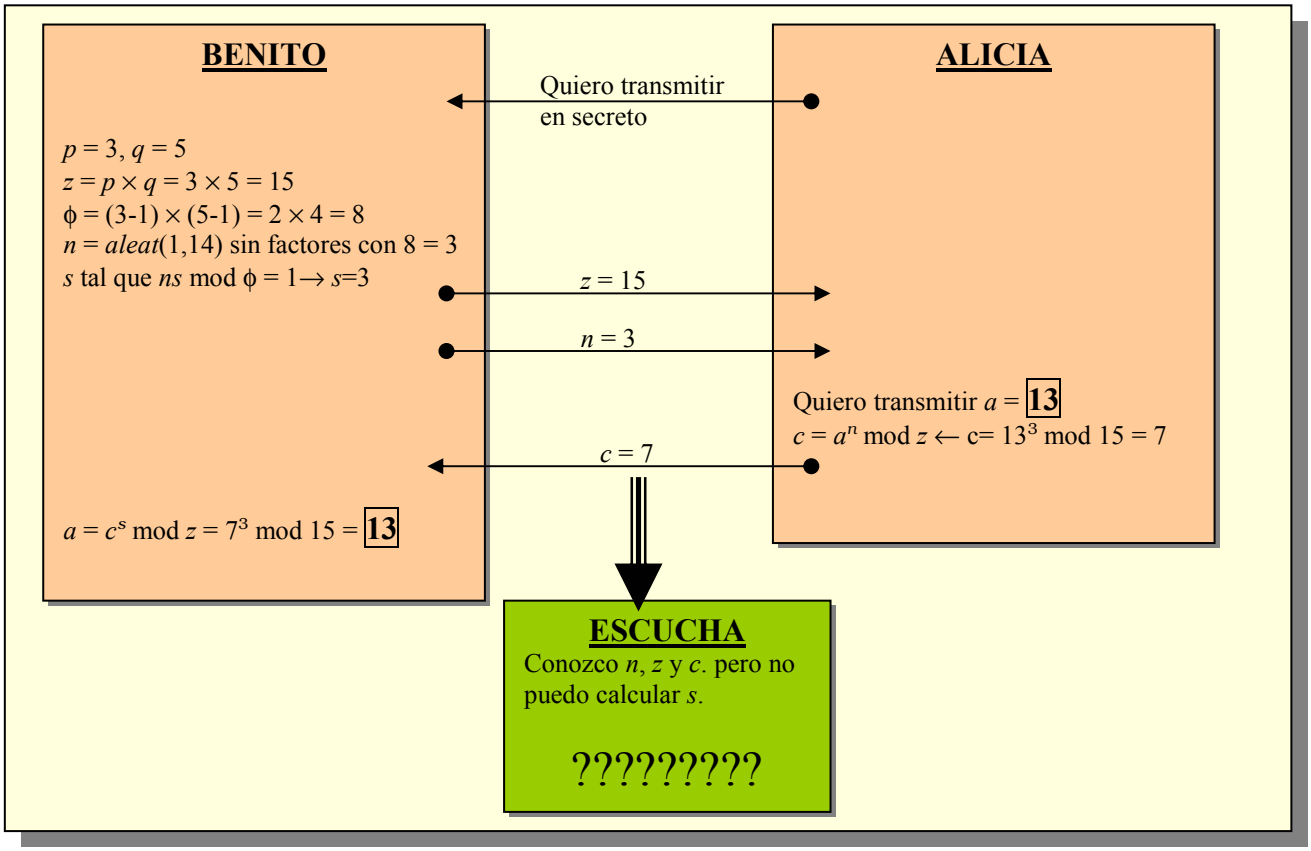
- ☞ Consideremos dos números primos p y q de 100 dígitos, seleccionados aleatoriamente por Benito (existe un algoritmo eficiente para comprobar la primalidad de números tan grandes). **{[Vamos a seguir el ejemplo con números pequeños para poder ver las operaciones a realizar. En nuestro caso tomaremos p=3 y q=5]}.**
- ☞ Sea z el producto de p y q **{[z= 3×5 = 15]}**. Benito puede calcular z eficientemente, sin embargo, no existe un algoritmo conocido, que pueda recalcular p y q a partir de z, dentro de la duración del tiempo del Universo.
- ☞ Sea ϕ igual a $(p-1)(q-1)$. **{[ϕ = 2×4 = 8]}.**
- ☞ Sea n un entero seleccionado aleatoriamente por Benito entre 1 y z-1, que no tenga factores comunes con ϕ **{[Decidimos que n es 3, que no tiene factores comunes con 8]}.**
- ☞ La teoría elemental de números, nos dice que existe un único entero s entre 1 y z-1 tal que se cumple que $ns \bmod \phi = 1$. Además, s es fácil de calcular a partir de n y ϕ, y su existencia es la comprobación de que n y ϕ no tienen factores comunes **{[Para nuestro ejemplo, s = 3, ya que $ns \bmod \phi$ = 3×3 mod 8 = 9 mod 8 = 1]}.**
- ☞ El teorema clave de este proceso es que:

  $a^n \bmod z = a$, siempre que $0 \le a < z$ y $x \bmod \phi = 1$
- ☞ Para permitir que Alicia (o alguien más) se comunique privadamente con él, Benito hace público los valores de z y n, pero mantiene en secreto el valor de s. **{[Hace público z=15 y n=3. Mantiene en secreto s=3]}**
- ☞ Supongamos que Alicia quiere transmitir un mensaje m, que puede ser transformado según su código ASCII, en un entero, que debe estar entre 0 y z-1 (en caso contrario, se trocea el mensaje). Suponemos que ese número entero obtenido del código ASCII es a. **{[Por ejemplo, a=13]}.**
- ☞ Alicia calcula con el algoritmo *expomod*, $c = a^n \bmod z$. y transmite c. **{[Alicia transmite c , que se calcula como $13^3 \bmod 15 = 7$]}.**
- ☞ Como Benito conoce s, le resulta fácil recalcular el mensaje original, mediante una llamada a *expomod*(c,s,z)

  $c^s \bmod z = (a^n \bmod z)^s \bmod z = (a^n)^s \bmod z = a^{ns} \bmod z = a$

  {[$c^s \bmod z = 7^3 \bmod 15 = 13$, que es el mensaje original]}.
- ☞ Un escucha, puede interceptar z, n y c. Su objetivo sería determinar el mensaje original a, que es el único número entre 0 y z-1 tal que $c = a^n \bmod z$., pero para eso necesitaría factorizar z

como producto de p y q, y factorizar un número de 200 dígitos, está fuera del alcance de la tecnología actual. **{[Naturalmente, en el ejemplo que hemos seguido sería fácil, ya que al conocer z=15, averiguamos fácilmente que p y q son 3 y 5. Esto no es tan fácil con números muy grandes]}**. Por tanto, la ventaja de Benito es que conoce esos factores p y q, que son necesarios para calcular ϕ y s.



El sistema secreto que acabamos de describir, goza de amplia aceptación como una de las mejores invenciones de la historia de la criptografía.

8 EXPLORACIÓN DE GRAFOS

8.1 Grafos y juegos: Introducción.

Vamos a utilizar un ejemplo de juego:

Se trata de una de las muchas variantes de Nim, conocido también como Marienbad. Inicialmente hay un montón (por lo menos 2) de cerillas en la mesa, entre dos jugadores. El primero en jugar puede coger las cerillas que quiera, con tal de coger una por lo menos, y dejar una por lo menos. A partir de entonces, cada jugador puede tomar un número de cerillas que esté entre una y el doble de las que tomó el contrincante en la anterior jugada. Gana el que coja la última cerilla. No hay empates.

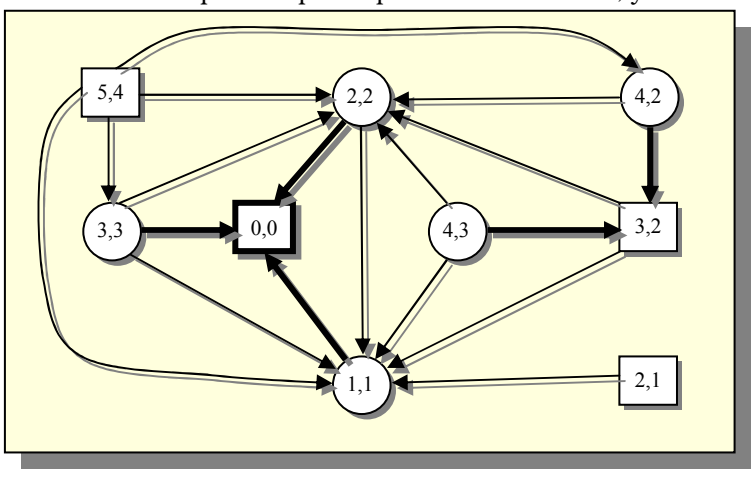
En cada fase del juego tenemos una situación distinta. Vamos a considerar que estamos en la siguiente situación:

> *Tenemos en la mesa 5 cerillas, y nuestro contrincante tomó dos cerillas, por lo que nosotros tenemos la posibilidad de coger 1, 2, 3 o 4 cerillas.*

Normalmente, valoraremos lo que puede ocurrir en las distintas posibilidades que se nos presentan:

- Cogiendo 4, le dejo una y gana
- Cogiendo 3, él puede tomar entre 1 y 6, por lo que puede coger las dos que quedan, y ganarme.
- Cogiendo 2, puede tomar entre 1 y 4, por lo que puede coger las 3 que quedan y ganarme.
- Cogiendo 1, puede coger entre 1 y 2, por lo que todavía no me gana. **Esta será mi mejor opción**.

En un ejemplo más complicado, quizás deberíamos haber tenido en cuenta una fase más del juego, es decir, de cada movimiento nuestro, los posibles contraataques de que dispone el contrincante, y de cada contraataque, nuestras nuevas posibilidades. Y así sucesivamente.



Este proceso de pensamiento anticipatorio, se puede formalizar mediante un grafo dirigido, en el que cada nodo del grafo se corresponde con una situación del juego, y cada arista en una posible jugada que nos lleva de una situación a otra. En nuestro caso, los nodos (las situaciones del juego), no sólo deben indicar las cerillas que hay en la mesa, sino también el tope de las que podemos coger. No es necesario realizar un grafo distinto para cada jugador, puesto que ambos tienen las mismas reglas y metas, y pueden utilizar el mismo grafo. Existen juegos en los que esto no ocurre, como el de *cacos y polis*, en el que los dos jugadores tienen objetivos distintos.

En la representación como grafo, los nodos redondos denotan situaciones de victoria y los cuadrados de derrota. Las aristas gruesas corresponden a jugadas victoriosas (desde una situación de victoria, ganaremos si seguimos las aristas gruesas). Evidentemente, en las situaciones de derrota, no hay aristas gruesas. Con este grafo, vemos que en ejemplo anterior, tenemos la partida perdida, y nuestra única opción es ir al nodo (4,2) esperando que el contrincante "falle", y nos deje ganar. Pero si el contrincante utiliza el método correcto, no podemos vencerle.

Estos mismos principios se pueden aplicar a otros juegos de estrategia. Por sencillez, supondremos que el juego se va turnando entre dos jugadores, que las reglas son las mismas para ambos (el juego es *simétrico*), y que la casualidad no interviene en el resultado (el juego es *determinista*). Algunas situaciones del juego, no ofrecen jugadas válidas, y por tanto su nodo asociado, no posee sucesores. Supondremos también que nunca el juego puede tener duración infinita, y en cada paso, las posibilidades de cada jugador son finitas.

Para etiquetar los nodos del grafo asociado, se siguen las siguientes reglas:

1. Se etiquetan las posiciones terminales. Estas etiquetas dependen del juego en cuestión. Por ejemplo, en el caso del ajedrez, una situación terminal por bloqueo mutuo, da lugar a tablas. Sin embargo, otras situaciones terminales son situaciones de victoria o derrota, según las reglas del juego en cuestión.
2. Una situación no terminal:
   - ☞ Es una situación de **victoria** si al menos uno de los sucesores es situación de *derrota*, porque el jugador que toma el turno, puede llevar al contrincante a esa situación de *derrota*.
   - ☞ Es una situación de **derrota**, si todos sus sucesores son situaciones de *victoria*, porque el jugador al que corresponda el turno, no puede evitar brindar en bandeja la *victoria* a su oponente.
   - ☞ Cualquier otra situación, da lugar a **tablas**. Entre los sucesores se encontrarán nodos de *tablas* y puede que nodos de *victoria*. El jugador puede evitar dejar al oponente en situación de *victoria*, pero no puede forzarle a la *derrota*.

En principio, esta técnica se puede aplicar a juegos tan complejos como el ajedrez. En este caso, cada nodo no solo describe la situación del tablero, sino que necesitaremos conocer a quién corresponde jugar, cuáles son los caballos y reyes que se han movido desde el principio (para saber si es legal *enrocar*) y si algún peón acaba de avanzar dos casillas (para saber si es posible una *captura al paso*). Podríamos construir así el grafo para una partida perfecta. Desgraciadamente (o por suerte para el ajedrez), este grafo es tan grande que es materialmente inmanejable. Nos tendremos que limitar a construir la parte del grafo "cercana" a la situación actual del juego. Tendremos en este caso, dos grafos distintos:

- ☞ Un grafo, como estructura de *bytes* y *punteros*, que se almacena en la memoria del computador, y que representa una parte (o todo) el grafo implícito.
- ☞ Un *grafo implícito*, que representa el juego total, pero que puede no existir completamente en la memoria del ordenador.

8.2 Recorridos de árboles.

En los árboles binarios, se suelen emplear tres técnicas:

	Exploramos en este orden:		
☞ En **preorden**.-	Nodo	Subárbol izquierdo	Subárbol derecho.
☞ En **orden infijo**.-	Subárbol izquierdo	Nodo	Subárbol derecho
☞ En **postorden**.-	Subárbol izquierdo	Subárbol derecho	Nodo

Estas tres técnicas exploran el nodo de izquierda a derecha. Hay otras tres técnicas análogas, que lo hacen de derecha a izquierda. Para cada una de las 6 técnicas descritas, el tiempo necesario para explorar un árbol con n nodos está en $\Theta(n)$.

Precondicionamiento.

Si tenemos que resolver varios casos parecidos con el mismo problema, puede merecer la pena invertir una cierta cantidad de tiempo en calcular resultados auxiliares que puedan ser utilizados en el futuro para acelerar la resolución de cada caso. Sea a el tiempo necesario para cada caso si no se dispone de esa información auxiliar. Sea b en tiempo necesario conociendo dicha información. Sea p el tiempo para calcular esa información auxiliar. Por tanto, para resolver n casos, se necesita un tiempo na, sin precondicionamiento, y un tiempo $p + nb$ si se emplea precondicionamiento. De esta forma, (siempre que b sea menor que a), resultará ventajoso utilizar el precondicionamiento si:

$$\boxed{n > \frac{p}{(a-b)}}$$

En el ejemplo de la devolución de monedas (sección 6.1), si construimos una matriz con las monedas a devolver para cada valor, una vez calculados los valores c_{nj} necesarios, podemos dar rápidamente el cambio siempre que sea necesario.

Otro ejemplo es del de la función que nos dice si un nodo es antecesor de otro. (Todo nodo es su antecesor, y la raíz es antecesor de todos). Es decir, dado un par de nodos (v,w), determinar si v es o no antecesor de w. Esto tiene un coste que está en $\Omega(n)$ en el peor caso. Pero se puede hacer un precondicionamiento que nos cuesta $\Theta(n)$, y a partir de entonces, el coste de la función es constante:

Si de cada nodo, calculamos el puesto que haría en un recorrido en preorden (*prenum*[*a*]), y también recorriéndolo en postorden (*postnum*[*a*]), podremos decir que:

$$v \text{ es antecesor de } w \Leftrightarrow prenum[v] \le prenum[w] \wedge postnum[v] \ge postnum[w]$$

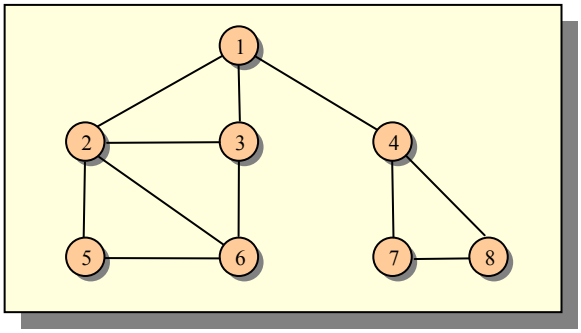
Una vez que los valores de *prenum* y *postnum* se han calculado en $\Theta(n)$, la condición requerida se puede comprobar en un tiempo que está en $\Theta(1)$.

8.3 Recorrido en profundidad: Grafos no dirigidos.

Para hacer un recorrido en profundidad, de un grafo $G = <N,A>$, se selecciona cualquier nodo v como punto de partida y se marca como *visitado*. Si hay algún nodo adyacente a v, que no haya sido *visitado*, se invoca recursivamente el procedimiento sobre dicho nodo. Al volver de la llamada recursiva, si hay otro nodo adyacente a v que no haya sido visitado, se vuelve a aplicar el procedimiento. Si no lo hay, se aplica el procedimiento a algún nodo del grafo que quede sin visitar (si es *no conexo*, quedarán nodos sin visitar). Se sigue así hasta que no queden nodos sin visitar.

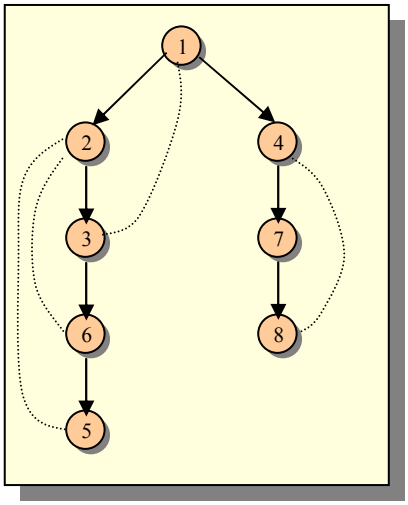
```
procedimiento recorridop(G)
    para cada v ∈ N hacer marca[v] ← no visitado
    para cada v ∈ N hacer
        si marca[v] ≠ visitado entonces rp(v)

procedimiento rp(v)
    {EL NODO v NO HA SIDO VISITADO ANTERIORMENTE}
    marca[v] ← visitado
    para cada nodo w adyacente a v hacer
        si marca[w] ≠ visitado entonces rp(w)
```

La recursión sólo se detiene cuando se ve bloqueada y no puede seguir. En ese momento "retrocede" para que sea posible estudiar posibilidades alternativas en niveles más elevados. Por ejemplo, en el grafo de la figura, y comenzando por en nodo 1, el orden de visita de los nodos sería 1, 2, 3, 6, 5, 4, 7, 8.



El algoritmo requiere un tiempo que está en $\Theta(n)$ para las llamadas al procedimiento, y un tiempo que está en $\Theta(a)$ para inspeccionar las marcas de visitado o no visitado, siendo a el número de aristas del grafo. Por tanto, el tiempo de ejecución está en $\boldsymbol{\Theta(máx(a,n))}$.

El recorrido en profundidad, asocia al grafo un árbol T de recubrimiento, o un bosque de árboles, en caso de que el grafo no sea conexo. En el ejemplo que estamos siguiendo, el árbol sería el de la figura siguiente, en el que las líneas discontinuas representan las aristas del grafo original que no han sido necesarias para el recorrido en profundidad del mismo.



Un recorrido en profundidad, también ofrece una manera de numerar los nodos del grafo que se está visitando. En otras palabras, los nodos del árbol asociado, se numeran en *preorden*. Por supuesto, el árbol y la numeración generados por un recorrido en profundidad, dentro de un grafo, no son únicos, sino que dependen del punto de partida seleccionado y del orden en que se visiten los nodos vecinos o adyacentes.

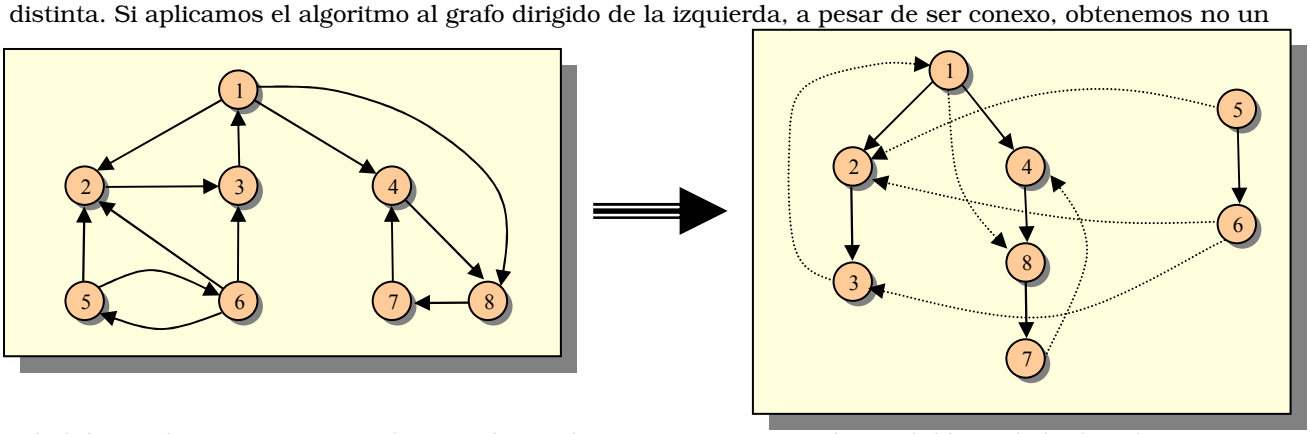
Puntos de articulación.

Un nodo v de un grafo conexo es un **punto de articulación**, si el subgrafo que se obtiene borrando v y todas sus aristas, ya no es conexo. Por ejemplo, en el grafo anterior, el nodo 1 es un *punto de articulación*, porque si lo borramos, nos aparece un grafo con dos componentes conexas.

Un grafo se llama **biconexo** (o bien *no articulado*) si es *conexo*, y carece de *puntos de articulación*. Un grafo se llama **bicoherente** (o bien *carente de istmos*, o 2-*arista conexo*) si todo punto de articulación está unido mediante al menos dos aristas a cada componente del subgrafo restante. Estos conceptos son importantes en la práctica. Por ejemplo, el grafo de una red telefónica, si es *biconexo*, nos asegura que se puede seguir comunicando, aunque falle uno de los nodos.

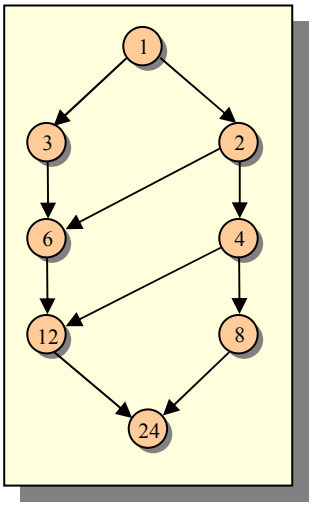
8.4 Recorrido en profundidad: Grafos dirigidos.

Los algoritmos presentados en la sección anterior, son igualmente válidos aquí, ya que el mecanismo es el mismo, con la única diferencia de que hay que tener en cuenta la dirección de las aristas para averiguar si un nodo es o no adyacente de otro. El algoritmo se comporta por tanto de manera distinta. Si aplicamos el algoritmo al grafo dirigido de la izquierda, a pesar de ser conexo, obtenemos no un



árbol de recubrimiento, sino un bosque de recubrimiento, representado en el dibujo de la derecha. En esta figura, también se representan con puntos, las aristas no utilizadas para el recorrido en profundidad.

El tiempo que requiere este algoritmo, está también en **$\Theta(máx(a,n))$**.



Grafos acíclicos: Ordenación topológica.

Los **grafos dirigidos acíclicos** se pueden utilizar para representar un cierto número de relaciones interesantes. Esta clase incluye los *árboles*, pero es menos general que la clase de todos los *grafos dirigidos*. Por ejemplo, podrían servir para representar la estructura de una expresión aritmética que contenga subexpresiones repetidas.

Estos grafos suelen utilizarse para especificar la forma en que se desarrolla un proyecto. Los nodos son las diferentes fases, y las aristas corresponden a las actividades a realizar para pasar de una fase a otra. Se puede realizar un recorrido *en profundidad* (debidamente modificado), para determinar una **ordenación topológica**. Esta ordenación, se hace de forma que si existe la arista (i,j), entonces el nodo i, precede al nodo j (no se puede realizar una fase del proyecto, sin haber terminado las anteriores). Por ejemplo, en el grafo acíclico de la figura, podrían existir varias formas de *orden natural*, como por ejemplo:

(1, 3, 2, 6, 4, 12, 8, 24) o bien, (1, 3, 6, 2, 4, 12, 8, 24)

8.5 Recorrido en anchura.

Un *recorrido en profundidad* llega a un nodo v, intenta a continuación visitar algún vecino de v, después algún vecino del vecino, y así sucesivamente. Sin embargo, un **recorrido en anchura**, llega al nodo v, y visita primero a todos los vecinos de v, y sólo después de esto, visita nodos más alejados.

A diferencia del recorrido en profundidad, el recorrido en anchura no es naturalmente recursivo. Para poder comparar ambos algoritmos, convertiremos primero el algoritmo del recorrido en profundidad *rp*, en un algoritmo iterativo *rp2* mediante el uso de una pila (LIFO).

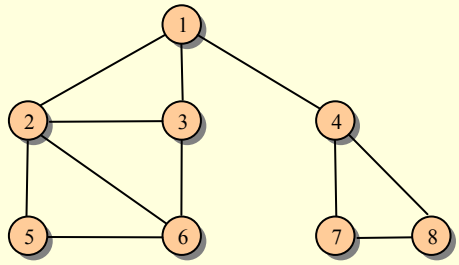
Después, lo podremos comparar con el procedimiento iterativo para el recorrido en anchura *ra*, que utiliza una cola (FIFO) en lugar de una pila (LIFO).

```
procedimiento rp2(v)
    P ← pila vacía
    marca[v] ← visitado
    mientras P no está vacía hacer
        mientras exista un nodo w adyacente a cima(P)
                tal que marca[w] ≠ visitado
                hacer marca[w] ← visitado
                apilar w en P {AHORA w ES LA NUEVA cima(P)}
        desapilar P
```

En ambos casos, necesitamos un programa principal que dé comienzo al recorrido poniendo todas las marcas de los nodos como *no visitado*, y que aplique el procedimiento *rp2* o *ra*, a cada nodo no visitado (por si hay más de una componente conexa)

```
procedimiento ra(v)
    Q ← cola vacía
    marca[v] ← visitado
    poner v en Q
    mientras Q no esté vacía hacer
        u ← primero(Q)
        quitar u de Q
        para cada nodo w adyacente a u hacer
            si marca[w] ≠ visitado entonces marca[w] ← visitado
            poner w en Q
```

Si aplicamos la búsqueda en anchura para el mismo ejemplo que seguimos en la sección 8.3 para un recorrido en profundidad, en este caso, los nodos se visitan en el orden reflejado en la siguiente tabla, en la cual se representa también el contenido de la cola *Q*, en las diferentes fases del proceso.



Fase	Nodo visitado	Q
1	1	2,3,4
2	2	3,4,5,6
3	3	4,5,6
4	4	5,6,7,8
5	5	6,7,8
6	6	7,8
7	7	8
8	8	

Igualmente, podemos asociar un árbol (o un bosque si no es conexo) al recorrido en anchura. También el coste está en **Θ(máx(*a*,*n*))**.

Teniendo cuidado en aplicar correctamente el concepto "*adyacente*", este algoritmo es posible aplicarlo también a grafos dirigidos.

El recorrido en anchura se emplea especialmente cuando haya que efectuar una exploración parcial de un grafo infinito (o tan grande que no resulte manejable).

8.6 Vuelta atrás.

Hay una cantidad enorme de problemas que se pueden formular en términos de una búsqueda ciega en grafos. Es decir, la solución es uno de los nodos, pero no hay ningún orden o procedimiento establecido para llegar directamente a ese nodo: es necesario explorar exhaustivamente el grafo hasta encontrarlo. Si el grafo contiene un número elevado de nodos (o es infinito), es inabordable construirlo explícitamente en memoria. Usaremos entonces un *grafo implícito*, en el que construimos partes relevantes del mismo, y vamos descartando los nodos ya examinados, para dejar espacio libre para otros fragmentos del grafo.

```
fun vuelta-atrás(ensayo)
    si valido(ensayo)
        entonces dev ensayo
    sino para cada hijo ∈ compleciones(ensayo) hacer
      si condiciones-de-poda(hijo) entonces vuelta-atrás(hijo) fsi
    fsi
ffun
```

El esquema de **backtracking** o **vuelta atrás**, realiza una exploración en profundidad dentro de un grafo dirigido mediante un algoritmo recursivo. El grafo a explorar suele ser un árbol (o al menos no contiene ciclos). Para cada nodo, *vuelta atrás* genera sus **compleciones** (es decir, sus hijos directos) y aplica de nuevo *vuelta atrás* sobre cada uno de ellos. Si en algún punto del recorrido se dan condiciones que hagan inútil seguir el camino tomado (**condiciones de poda**) se abandona esa rama, retrocediendo a la última decisión.

Para un problema dado, es necesario especificar:

- Qué es un **ensayo**, es decir, un nodo del grafo.
- La función ***válido***, que determina si un nodo es solución al problema o no.
- La función ***compleciones***, que genera los hijos de un nodo dado.
- La función ***condiciones-de-poda*** que verifica si puede descartarse de antemano una rama del árbol, aplicando los criterios de poda sobre el nodo origen de esa rama. Esta función devuelve *Cierto* si ha de explorarse el nodo, y *Falso* si puede abandonarse la búsqueda de ese nodo.

La exploración en profundidad que realiza implícitamente el esquema de vuelta atrás puede también realizarse mediante un algoritmo iterativo, utilizando una pila para mantener los nodos generados y aún no visitados:

```
fun busqueda-en-profundidad(ensayo)
    p ← pila vacía
    aplilar(ensayo,p)
    mientras ¬vacía(p) hacer
        nodo ← desaplilar(p)
        si válido(nodo) entonces dev nodo
        sino
            para cada hijo en compleciones(nodo) hacer
                si condiciones-de-poda(hijo) entonces apilar(hijo,p) fsi
            fpara
        fsi
    fmientras
ffun
```

Nótese que *compleciones*, *condiciones-de-poda* y *válido* son funciones idénticas a la versión recursiva, por lo que para pasar de una implementación a otra, solo hay que cambiar el cuerpo del programa.

8.7 Ramificación y poda.

Igual que antes, esta técnica sirve para explorar un grafo dirigido implícito. Una vez más, este grafo es normalmente acíclico o incluso un árbol. Pero en ocasiones, se requiere que la solución alcanzada sea óptima. En ese caso, se calcula para cada nodo una cota del posible valor de aquellas soluciones que pudieran encontrarse a partir de ese nodo en el grafo. Si la cota muestra que cualquier solución que vayamos a encontrar por esa rama será necesariamente peor que la mejor encontrada hasta el momento, entonces no es necesario seguir explorando por esa parte del grafo.

La cota calculada se puede utilizar como una condición más de poda, o se puede utilizar también para guiar la búsqueda: en lugar de buscar *en profundidad* o *en anchura*, seleccionamos en cada momento el nodo más prometedor de entre los que nos quedan por explorar. Veíamos que un recorrido *en profundidad* usaba una *pila*, y un recorrido *en anchura* utilizaba una *cola*. Pues bien, para *ramificación y poda*, se usa por ejemplo un *montículo*, ordenado según esa cota calculada, por lo que los nodos se exploran en orden a su probabilidad de éxito.

```
fun ramificacion-y-poda-en-anchura(ensayo)
    p ← cola vacía
    c ← coste mínimo
    solución ← solución vacía
    encolar(ensayo,p)
    mientras ¬vacía(p) hacer
        nodo ← desencolar(p)
        si válido(nodo) entonces hacer
            si coste(nodo) < c entonces hacer
                solución ← nodo
                c ← coste(nodo)
            fsi
        sino
            para cada hijo en compleciones(nodo) hacer
                si condiciones-de-poda(hijo) Y cota(hijo) < c
                    entonces encolar(hijo.p)
                fsi
            fpara
        fsi
    fmientras
ffun
```

```
fun ramificación-y-poda(ensayo)
    m ← montículo-vacío
    cota-superior ← inicializar-cota-superior
    solución ← solución vacía
    añadir-nodo(ensayo,m)
    mientras ¬vacío(m) hacer
        nodo ← extraer-raiz(m)
        si válido(nodo) entonces hacer
            si coste(nodo) < cota-superior entonces hacer
                solución ← nodo
                cota-superior ← coste(nodo)
            fsi
        sino
            si cota-inferior(nodo) ≥ cota-superior entonces
                dev solución
            sino
                para cada hijo en compleciones(nodo) hacer
                    si condiciones-de-poda(hijo) Y cota-inferior(hijo)< cota-superior
                        entonces añadir-nodo(hijo,m)
                    fsi
                fpara
            fsi
        fsi
    fmientras
ffun
```

Este esquema, podría aplicarse a ejemplos como:

El problema de la asignación.

En el problema de la asignación, hay que asignar *n* tareas a *n* agentes, de forma que cada agente realice exactamente una tarea. El problema consiste en asignar las tareas de forma que se minimice el coste total de ejecutar las *n* tareas.

Por ejemplo:

> Una empresa de mensajería dispone de tres motoristas en distintos puntos de la ciudad, y tiene que atender a tres clientes en otros tres puntos. Se puede estimar el tiempo que tardaría cada motorista en atender a cada uno de los clientes según la tabla siguiente.

	Moto 1	Moto 2	Moto 3
Cliente 1	30	40	70
Cliente 2	60	20	10
Cliente 3	40	90	20

- Para obtener una *cota-superior-inicial*, hay varias soluciones. Por ejemplo se puede utilizar una de las diagonales (que las tres motos atiendan a los clientes 1,2 y 3 respectivamente), lo que tendría un coste de 30+20+20 = 70 minutos. Evidentemente esta puede ser una cota superior inicial, ya que la solución al problema será mejor o igual a ésta.
- Para obtener el *coste* de un nodo final, no hay más que sumar los costes de cada moto.
- Para obtener la *cota-inferior* de un nodo no final pondremos un ejemplo: si tenemos asignada la moto 2 al cliente 1, con un coste de 40 minutos, cualquier solución obtenida a partir de esa asignación parcial, tendrá un coste igual o peor a 40+10+40 = 90 minutos. Al llegar a esta rama, podemos ver que no es necesario seguir, ya que $\neg(\textit{cota-inferior} < \textit{cota-superior}) \Leftrightarrow \neg(90 < 70)$. Es decir, por esta rama, nunca obtendremos una solución mejor que la ya encontrada hasta el momento.

Consideraciones generales.

Resulta casi imposible dar una idea precisa de lo bien que se va a comportar esta técnica en un problema dado, empleando una cota dada. Hay que llegar a un compromiso en la calidad de la cota calculada. Una buena cota, nos evitará examinar nodos inútilmente, y llegaremos antes a la solución óptima, pero en contrapartida, puede ser una cota difícil de calcular, y que nos haga perder mucho tiempo en calcularla para cada nodo. Sin embargo, en la práctica, casi siempre es rentable invertir tiempo en calcular una buena cota.

8.8 El principio de MINIMAX.

Para juegos como el ajedrez, queda descartado el poder analizar todo el grafo implícito, por lo que nos conformaremos con examinar parte del grafo a partir de una posición dada.

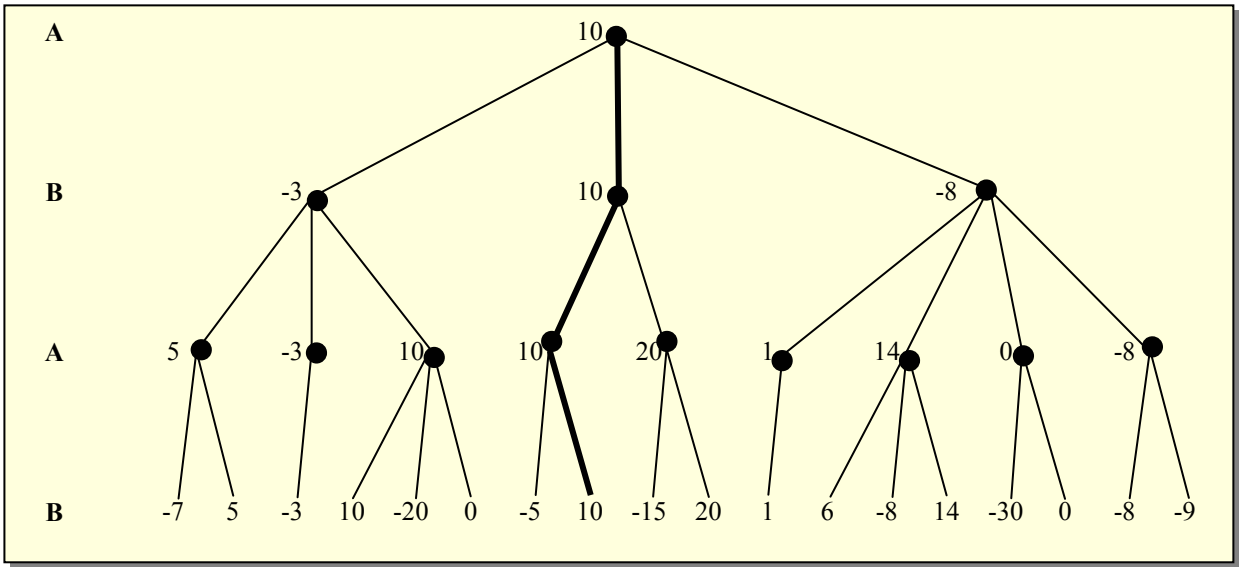
El primer paso es establecer una función estática ***eval***, que atribuya un valor a cada situación del tablero. Idealmente, el valor de *eval*(*u*), debiera crecer cuanto más favorable sea la situación para que ganemos (que ganen las blancas). Habitualmente, se dan valores próximos a cero si la ventaja no es clara para ninguno de los dos, valores positivos si vamos ganando (van ganando las blancas), y valores negativos si nos acercamos a nuestra derrota (ganan las negras). Debe existir un compromiso entre lo preciso de dicha función, y el coste de calcularla en cada situación del tablero.

Por ejemplo, una función sencilla de calcular, aunque evidentemente no perfecta, sería la de dar un valor a cada pieza, sumar las blancas que haya en el tablero, y restar las negras. Con esta función, nuestra mejor jugada sería la que **maximice** el valor del *eval* para el siguiente tablero. Naturalmente, hemos de esperar que en la siguiente jugada, nuestro adversario elegirá una jugada que **minimice** *eval*.

Para añadir aspectos más dinámicos a la evaluación estática que proporciona *eval*, resulta preferible examinar varias jugadas por anticipado.

La razón por la que esta técnica se llama **MINIMAX**, es porque las negras intentan **MINI**mizar la ventaja que se les permite a las blancas, y las blancas, por su parte, intentan **MAX**imizar la ventaja que se obtiene de cada jugada.

En el ejemplo de la figura, suponemos que se trata de un juego en el que el jugador A intenta maximizar la función *eval*, y el jugador B, la intenta minimizar. Después de obtener la parte del grafo representado, se aplica *eval* a los tableros finales, y en nuestro ejemplo obtenemos –7, 5, -3, 10, -20,...etc.



De los dos tableros inferiores de la izquierda, el jugador A elegirá evidentemente el de valor 5 (el que maximiza *eval*). Del siguiente, no tiene otra posibilidad que elegir –3. De los tres siguientes, elegirá el de valor 10. Pero de estas tres opciones, hemos de esperar que el jugador B, elija la menor (minimice *eval*), es decir –3. Y así sucesivamente. Analizando por tanto la parte del grafo de que disponemos, nuestra mejor jugada es elegir la que nos lleva a un tablero con valor 10, ya que en la siguiente jugada, nuestro contrincante elegirá dicha jugada. Es decir, el camino mostrado en grueso en la figura.

La técnica minimax, puede mejorarse, por ejemplo, explorando con más profundidad los movimientos más prometedores. También abandonando ramas, si se dispone de información suficiente para concluir que no pueden llegar a tener influencia en el valor de los nodos pertenecientes a zonas más altas del árbol. Este segundo tipo de mejora, se conoce con el nombre de **poda alfa-beta**.

En general, para determinar los aspectos de la resolución de un problema, es recomendable seguir ciertos pasos, que particularizados para el esquema de ***búsqueda ciega en grafos*** son:

Elección del esquema.

Hay muchos problemas que pueden resolverse con esquemas de *búsqueda ciega* en un árbol, ya que no es más que una versión sofisticada de "*prueba y error*". Es por tanto, que suelen tener un coste elevado, y debemos restringirlos a los casos en que ninguno de los demás esquemas (*voraz* y *divide y vencerás*) sea aplicable.

Una vez desechados el esquema *voraz* y *divide y vencerás*, debemos tener claro en qué consiste el árbol, cuáles son sus nodos y cómo pasar de un nodo a sus hijos. Se deberá elegir entre la búsqueda en *anchura*, en *profundidad*, *recursiva* o *iterativa*. Normalmente, un algoritmo de *vuelta atrás*, suele ser el más intuitivo. De todas formas, esta elección ya no es tan comprometida, y las funciones a especificar, no suelen cambiar de un esquema a otro.

Si además de encontrar una solución, ésta ha de ser óptima, entonces será necesario aplicar el esquema de *ramificación y poda*.

Identificación con el problema.

Hay que identificar los *nodos*, cómo se decide que un nodo es *solución*, cómo se hallan los *hijos* de un nodo (sus *compleciones*), cuáles son las *condiciones de poda* que interrumpen la búsqueda en una rama. Si se trata de *ramificación y poda*, hay que explicar cómo se estima la *cota* para un nodo dado.

Estructuras de datos.

Suele ser suficiente con dar la estructura de los *nodos* (o *ensayos*). Si además, se trata de una búsqueda *iterativa*, necesitaremos de una estructura de *pila* (búsqueda en *profundidad*), *cola* (búsqueda en *anchura*) o *montículo* (*ramificación y poda*).

A veces es necesario guardar los nodos visitados para llegar a la solución, para lo que una *lista* suele ser lo más adecuado.

Algoritmo completo.

Hay que especificar las funciones *compleciones*, *solución* y *condiciones–de-poda*.
En ramificación y poda, es necesario también las funciones *cota* y *coste*.

Estudio del coste.

En la mayoría de las ocasiones, no se puede dar de forma exacta el coste de una búsqueda ciega. Una buena estimación suele ser el tamaño del árbol de búsqueda, en función del tamaño del problema.