Algoritmos

¿Qué es un algoritmo?

Una definición informal (no se considera aquí una definición formal, aunque existe): conjunto finito de reglas que dan una secuencia de operaciones para resolver todos los problemas de un tipo dado. De forma más sencilla, podemos decir que un algoritmo es un conjunto de pasos que nos permite obtener un dato. Además debe cumplir estas condiciones:

- **Finitud**: el algoritmo debe acabar tras un número finito de pasos. Es más, es casi fundamental que sea en un número razonable de pasos.
- **Definibilidad**: el algoritmo debe definirse de forma precisa para cada paso, es decir, hay que evitar toda ambigüedad al definir cada paso. Puesto que el lenguaje humano es impreciso, los algoritmos se expresan mediante un lenguaje formal, ya sea matemático o de programación para un computador.
- **Entrada**: el algoritmo tendrá cero o más entradas, es decir, cantidades dadas antes de empezar el algoritmo. Estas cantidades pertenecen además a conjuntos especificados de objetos. Por ejemplo, pueden ser cadenas de caracteres, enteros, naturales, fraccionarios, etc. Se trata siempre de cantidades representativas del mundo real expresadas de tal forma que sean aptas para su interpretación por el computador.
- **Salida**: el algoritmo tiene una o más salidas, en relación con las entradas.
- **Efectividad**: se entiende por esto que una persona sea capaz de realizar el algoritmo de modo exacto y sin ayuda de una máquina en un lapso de tiempo finito.

A menudo los algoritmos requieren una organización bastante compleja de los datos, y es por tanto necesario un estudio previo de las estructuras de datos fundamentales. Dichas estructuras pueden implementarse de diferentes maneras, y es más, existen algoritmos para implementar dichas estructuras. El uso de estructuras de datos adecuadas pueden hacer trivial el diseño de un algoritmo, o un algoritmo muy complejo puede usar estructuras de datos muy simples.

Uno de los algoritmos más antiguos conocidos es el algoritmo de Euclides. El término algoritmo proviene del matemático Muhammad ibn Musa al-Khwarizmi, que vivió aproximadamente entre los años 780 y 850 d.C. en la actual nación Iraní. El describió la realización de operaciones elementales en el sistema de numeración decimal. De al-Khwarizmi se obtuvo la derivación algoritmo.

Clasificación de algoritmos

- **Algoritmo determinista**: en cada paso del algoritmo se determina de forma única el siguiente paso.
- **Algoritmo no determinista**: deben decidir en cada paso de la ejecución entre varias alternativas y agotarlas todas antes de encontrar la solución.

Todo algoritmo tiene una serie de características, entre otras que requiere una serie de recursos, algo que es fundamental considerar a la hora de implementarlos en una máquina. Estos recursos son principalmente:

- *El tiempo*: período transcurrido entre el inicio y la finalización del algoritmo.
- *La memoria*: la cantidad (la medida varía según la máquina) que necesita el algoritmo para su ejecución.

Obviamente, la capacidad y el diseño de la máquina pueden afectar al diseño del algoritmo.

En general, la mayoría de los problemas tienen un parámetro de entrada que es el número de datos que hay que tratar, esto es, N. La cantidad de recursos del algoritmo es tratada como una función de N. De esta manera puede establecerse un tiempo de ejecución del algoritmo que suele ser proporcional a una de las siguientes funciones:

- **1** : Tiempo de ejecución constante. Significa que la mayoría de las instrucciones se ejecutan una vez o muy pocas.
- **logN** : Tiempo de ejecución logarítmico. Se puede considerar como una gran constante. La base del logaritmo (en informática la más común es la base 2) cambia la constante, pero no demasiado. El programa es más lento cuanto más crezca N, pero es inapreciable, pues logN no se duplica hasta que N llegue a N^2.
- **N** : Tiempo de ejecución lineal. Un caso en el que N valga 40, tardará el doble que otro en que N valga 20. Un ejemplo sería un algoritmo que lee N números enteros y devuelve la media aritmética.
- **N·logN** : El tiempo de ejecución es N·logN. Es común encontrarlo en algoritmos como Quick Sort y otros del estilo divide y vencerás. Si N se duplica, el tiempo de ejecución es ligeramente mayor del doble.
- $\mathbf{N^2}$: Tiempo de ejecución cuadrático. Suele ser habitual cuando se tratan pares de elementos de datos, como por ejemplo un bucle anidado doble. Si N se duplica, el tiempo de ejecución aumenta cuatro veces. El peor caso de entrada del algoritmo Quick Sort se ejecuta en este tiempo.
- $\mathbf{N^3}$: Tiempo de ejecución cúbico. Como ejemplo se puede dar el de un bucle anidado triple. Si N se duplica, el tiempo de ejecución se multiplica por ocho.
- $\mathbf{2^N}$: Tiempo de ejecución exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. El problema de la mochila resuelto por un algoritmo de fuerza bruta -simple backtracking- es un ejemplo. Si N se duplica, el tiempo de ejecución se eleva al cuadrado.

- **Algoritmos polinomiales**: aquellos que son proporcionales a N^k. Son en general factibles.
- **Algoritmos exponenciales**: aquellos que son proporcionales a k^N. En general son infactibles salvo un tamaño de entrada muy reducido.

Notación O-grande

En general, el tiempo de ejecución es proporcional, esto es, multiplica por una constante a alguno de los tiempos de ejecución anteriormente propuestos, además de la suma de algunos términos más pequeños. Así, un algoritmo cuyo tiempo de ejecución sea $T = 3N^2 + 6N$ se puede considerar proporcional a N^2. En este caso se diría que el algoritmo es del orden de N^2, y se escribe $O(N^2)$
Los grafos definidos por matriz de adyacencia ocupan un espacio $O(N^2)$, siendo N el número de vértices de éste.

La notación O-grande ignora los factores constantes, es decir, ignora si se hace una mejor o peor implementación del algoritmo, además de ser independiente de los datos de entrada del algoritmo. Es decir, la utilidad de aplicar esta notación a un algoritmo es encontrar un límite superior del tiempo de ejecución, es decir, el peor caso.

A veces ocurre que no hay que prestar demasiada atención a esto. Conviene diferenciar entre el peor caso y el esperado. Por ejemplo, el tiempo de ejecución del algoritmo Quick Sort es de $O(N^2)$. Sin embargo, en la práctica este caso no se da casi nunca y la mayoría de los casos son proporcionales a $N \cdot \log N$. Es por ello que se utiliza esta última expresión para este método de ordenación.

Una definición rigurosa de esta notación es la siguiente:
Una función g(N) pertenece a O(f(N)) si y sólo si existen las constantes c_0 y N_0 tales que: $|g(N)| <= |c_0 \cdot f(N)|$, para todo $N >= N_0$.

Estimación de la O grande

Reglas para calcular el tiempo de ejecución en el peor caso de los distintos tipos de instrucciones

Regla1: Instrucciones elementales

Se llama instrucción elemental a las operaciones de entrada/salida, asignaciones y expresiones aritméticas en las que no está involucrada ninguna variable que dependa del tamaño de la entrada del algoritmo.

$$[I \equiv \text{instrucción elemental}] \Rightarrow \mathbf{T}_I \in \mathbf{O}(1)$$

Regla 2: Secuencia de instrucciones

$$\left[\begin{array}{l} I \equiv I_1; I_2 \\ T_{I_1} \in \mathbf{O}(f_1) \\ T_{I_2} \in \mathbf{O}(f_2) \end{array}\right] \Rightarrow \mathbf{T}_I \in \mathbf{O}(max(f_1, f_2))$$

Regla 3: Instrucciones de selección

$$\left[\begin{array}{l} I \equiv \text{SI } B \text{ ENTONCES } I_1 \text{ SI NO } I_2 \\ T_B \in \mathbf{O}(f_B) \\ T_{I_1} \in \mathbf{O}(f_1) \\ T_{I_2} \in \mathbf{O}(f_2) \end{array}\right] \Rightarrow \mathbf{T}_I \in \mathbf{O}(max(f_B, f_1, f_2))$$

$$\left[\begin{array}{l} I \equiv \text{CASO } E \text{ EN caso-1: } I_1 \ldots \text{ caso-k: } I_k \\ T_E \in \mathbf{O}(f_E) \\ T_{I_i} \in \mathbf{O}(f_i), i \in \{1, \ldots, k\} \end{array}\right] \Rightarrow \mathbf{T}_I \in \mathbf{O}(max(f_E, f_i))$$

Regla 4: Instrucciones de repetición

$$\left[\begin{array}{l} I \equiv \text{MIENTRAS } B \text{ HACER } J \\ f_{iter} \equiv \text{núm. iteraciones (peor caso)} \\ T_{B;J} \in \mathbf{O}(f_i) \text{ en la iteración } i \end{array}\right] \Rightarrow \mathbf{T}_I \in \mathbf{O}\left(\sum_{i=1}^{f_{iter}} f_i\right)$$

Nota: la complejidad de un bucle

PARA i DESDE a1 HASTA a2 HACER J

se calcula igual que la de un bucle MIENTRAS salvo que en el caso PARA el coste de la condición B se puede ignorar puesto que es siempre constante.

Regla 5: Llamadas a subprogramas

En el caso de instrucciones que sean llamadas a otros algoritmos (subprogramas), el coste de la instrucción será el coste del subprograma:

$$[\ I \equiv \text{llamada a un subprograma } P\ ] \Rightarrow \mathbf{T}_I = \mathbf{T}_P$$

Regla 6: Sí el algoritmo es recursivo

Para calcular la complejidad de un algoritmo recursivo es necesario realizar los dos siguientes pasos:

1. Plantear la ecuación recurrente asociada con el tiempo de ejecución T del subprograma.
2. Resolver la ecuación recurrente anterior (es decir, encontrar una expresión no recursiva para el valor de T).

Las familias de recurrencias más habituales son:

1. Ecuaciones recurrentes obtenidas mediante sustracción
2. Ecuaciones recurrentes obtenidas mediante división

Clasificación de problemas

Los problemas matemáticos se pueden dividir en primera instancia en dos grupos:

- **Problemas indecidibles**: aquellos que no se pueden resolver mediante un algoritmo.
- **Problemas decidibles**: aquellos que cuentan al menos con un algoritmo para su cómputo.

Sin embargo, que un problema sea decidible no implica que se pueda encontrar su solución, pues muchos problemas que disponen de algoritmos para su resolución son inabordables para un computador por el elevado número de operaciones que hay que realizar para resolverlos. Esto permite separar los problemas decidibles en dos:

- **intratables**: aquellos para los que no es factible obtener su solución.
- **tratables**: aquellos para los que existe al menos un algoritmo capaz de resolverlo en un tiempo razonable.

Los problemas pueden clasificarse también atendiendo a su **complejidad**. Aquellos problemas para los que se conoce un algoritmo polinómico que los resuelve se denominan **clase P**. Los algoritmos que los resuelven son deterministas. Para otros problemas, sus mejores algoritmos conocidos son no deterministas. Esta clase de problemas se denomina **clase NP**. Por tanto, los problemas de la clase **P** son un subconjunto de los de la clase **NP**, pues sólo cuentan con una alternativa en cada paso.

Algoritmos de ordenación

Su finalidad es organizar ciertos datos en un orden creciente o decreciente mediante una regla prefijada (numérica, alfabética...). Atendiendo al tipo de elemento que se quiera ordenar puede ser:

Los principales algoritmos de ordenación son:

Selección:

Este método consiste en buscar el elemento más pequeño del array y ponerlo en primera posición; luego, entre los restantes, se busca el elemento más pequeño y se coloca en segudo lugar, y así sucesivamente hasta colocar el último elemento. Por ejemplo, si tenemos el array {40,21,4,9,10,35}, los pasos a seguir son:

{4,21,40,9,10,35} <-- Se coloca el 4, el más pequeño, en primera posición : se cambia el 4 por el 40.
{4,9,40,21,10,35} <-- Se coloca el 9, en segunda posición: se cambia el 9 por el 21.
{4,9,10,21,40,35} <-- Se coloca el 10, en tercera posición: se cambia el 10 por el 40.
{4,9,10,21,40,35} <-- Se coloca el 21, en tercera posición: ya está colocado.
{4,9,10,21,35,40} <-- Se coloca el 35, en tercera posición: se cambia el 35 por el 40.

Si el array tiene N elementos, el número de comprobaciones que hay que hacer es de N*(N-1)/2, luego el tiempo de ejecución está en $O(n^2)$

Burbuja

Consiste en comparar pares de elementos adyacentes e intercambiarlos entre sí hasta que estén todos ordenados. Con el array anterior, {40,21,4,9,10,35}:

Primera pasada:
{21,40,4,9,10,35} <-- Se cambia el 21 por el 40.
{21,4,40,9,10,35} <-- Se cambia el 40 por el 4.

{21,4,9,40,10,35} <-- Se cambia el 9 por el 40.
{21,4,9,10,40,35} <-- Se cambia el 40 por el 10.
{21,4,9,10,35,40} <-- Se cambia el 35 por el 40.

Segunda pasada:
{4,21,9,10,35,40} <-- Se cambia el 21 por el 4.
{4,9,21,10,35,40} <-- Se cambia el 9 por el 21.
{4,9,10,21,35,40} <-- Se cambia el 21 por el 10.

Ya están ordenados, pero para comprobarlo habría que acabar esta segunda comprobación y hacer una tercera.

Si el array tiene N elementos, para estar seguro de que el array está ordenado, hay que hacer N-1 pasadas, por lo que habría que hacer (N-1)*(N-i-1) comparaciones, para cada i desde 1 hasta N-1. El número de comparaciones es, por tanto, N(N-1)/2, lo que nos deja un tiempo de ejecución, al igual que en la selección, en $O(n^2)$.

Inserción directa

En este método lo que se hace es tener una sublista ordenada de elementos del array e ir insertando el resto en el lugar adecuado para que la sublista no pierda el orden. La sublista ordenada se va haciendo cada vez mayor, de modo que al final la lista entera queda ordenada. Para el ejemplo {40,21,4,9,10,35}, se tiene:

{40,21,4,9,10,35} <-- La primera sublista ordenada es {40}.

Insertamos el 21:
{40,40,4,9,10,35} <-- aux=21;
{21,40,4,9,10,35} <-- Ahora la sublista ordenada es {21,40}.

Insertamos el 4:
{21,40,40,9,10,35} <-- aux=4;
{21,21,40,9,10,35} <-- aux=4;
{4,21,40,9,10,35} <-- Ahora la sublista ordenada es {4,21,40}.

Insertamos el 9:
{4,21,40,40,10,35} <-- aux=9;
{4,21,21,40,10,35} <-- aux=9;
{4,9,21,40,10,35} <-- Ahora la sublista ordenada es {4,9,21,40}.

Insertamos el 10:
{4,9,21,40,40,35} <-- aux=10;
{4,9,21,21,40,35} <-- aux=10;
{4,9,10,21,40,35} <-- Ahora la sublista ordenada es {4,9,10,21,40}.

Y por último insertamos el 35:
{4,9,10,21,40,40} <-- aux=35;
{4,9,10,21,35,40} <-- El array está ordenado.

En el peor de los casos, el número de comparaciones que hay que realizar es de N*(N+1)/2-1, lo que nos deja un tiempo de ejecución en $O(n^2)$. En el mejor caso (cuando la lista ya estaba ordenada), el número de comparaciones es N-2. Todas ellas son falsas, con lo que no se produce ningún intercambio. El tiempo de ejecución está en O(n).

El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Vemos que cuanto más ordenada esté inicialmente más se acerca a O(n) y cuanto más desordenada, más se acerca a $O(n^2)$.

El peor caso es igual que en los métodos de burbuja y selección, pero el mejor caso es lineal, algo que no ocurría en éstos, con lo que para ciertas entradas podemos tener ahorros en tiempo de ejecución.

Inserción binaria

Es el mismo método que la inserción directa, excepto que la búsqueda del orden de un elemento en la sublista ordenada se realiza mediante una búsqueda binaria, lo que en principio supone un ahorro de tiempo. No obstante, dado que para la inserción sigue siendo necesario un desplazamiento de los elementos, el ahorro, en la mayoría de los casos, no se produce, si bien hay compiladores que realizan optimizaciones que lo hacen ligeramente más rápido.

Shell

Es una mejora del método de inserción directa, utilizado cuando el array tiene un gran número de elementos. En este método no se compara a cada elemento con el de su izquierda, como en el de inserción, sino con el que está a un cierto número de lugares (llamado salto) a su izquierda. Este salto es constante, y su valor inicial es N/2 (siendo N el número de elementos, y siendo división entera). Se van dando pasadas hasta que en una pasada no se intercambie ningún elemento de sitio. Entonces el salto se reduce a la mitad, y se vuelven a dar pasadas hasta que no se intercambie ningún elemento, y así sucesivamente hasta que el salto vale 1.

Por ejemplo, lo pasos para ordenar el array {40,21,4,9,10,35} mediante el método de Shell serían:

Salto=3:
  Primera pasada:
  {9,21,4,40,10,35} <-- se intercambian el 40 y el 9.
  {9,10,4,40,21,35} <-- se intercambian el 21 y el 10.
Salto=1:
  Primera pasada:
  {9,4,10,40,21,35} <-- se intercambian el 10 y el 4.
  {9,4,10,21,40,35} <-- se intercambian el 40 y el 21.

{9,4,10,21,35,40} <-- se intercambian el 35 y el 40.
Segunda pasada:
{4,9,10,21,35,40} <-- se intercambian el 4 y el 9.

Con sólo 6 intercambios se ha ordenado el array, cuando por inserción se necesitaban muchos más.

Ordenación rápida (quicksort)

Este método se basa en la táctica "divide y vencerás", que consiste en ir subdividiendo el array en arrays más pequeños, y ordenar éstos. Para hacer esta división, se toma un valor del array como pivote, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha. A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el array.

Normalmente se toma como pivote el primer elemento de array, y se realizan dos búsquedas: una de izquierda a derecha, buscando un elemento mayor que el pivote, y otra de derecha a izquierda, buscando un elemento menor que el pivote. Cuando se han encontrado los dos, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran.Por ejemplo, para dividir el array {21,40,4,9,10,35}, los pasos serían:

{21,40,4,9,10,35} <-- se toma como pivote el 21. La búsqueda de izquierda a derecha encuantra el valor 40, mayor que pivote, y la búsqueda de derecha a izquierda encuentra el valor 10, menor que el pivote. Se intercambian:
{21,10,4,9,40,35} <-- Si seguimos la búsqueda, la primera encuentra el valor 40, y la segunda el valor 9, pero ya se han cruzado, así que paramos. Para terminar la división, se coloca el pivote en su lugar (en el número encontrado por la segunda búsqueda, el 9, quedando:
{9,10,4,21,40,35} <-- Ahora tenemos dividido el array en dos arrays más pequeños: el {9,10,4} y el {40,35}, y se repetiría el mismo proceso.

Intercalación

no es propiamente un método de ordenación, consiste en la unión de dos arrays ordenados de modo que la unión esté también ordenada. Para ello, basta con recorrer los arrays de izquierda a derecha e ir cogiendo el menor de los dos elementos, de forma que sólo aumenta el contador del array del que sale el elemento siguiente para el array-suma. Si quisiéramos sumar los arrays {1,2,4} y {3,5,6}, los pasos serían:

Inicialmente: i1=0, i2=0, is=0.
Primer elemento: mínimo entre 1 y 3 = 1. Suma={1}. i1=1, i2=0, is=1.
Segundo elemento: mínimo entre 2 y 3 = 2. Suma={1,2}. i1=2, i2=0, is=2.
Tercer elemento: mínimo entre 4 y 3 = 3. Suma={1,2,3}. i1=2, i2=1, is=3.
Cuarto elemento: mínimo entre 4 y 5 = 4. Suma={1,2,3,4}. i1=3, i2=1, is=4.
Como no quedan elementos del primer array, basta con poner los elementos que quedan del segundo array en la suma:
Suma={1,2,3,4}+{5,6}={1,2,3,4,5,6}

Fusión (merge sort):

Consta de dos partes, una parte de intercalación de listas y otra de divide y vencerás.

Primera parte

¿Cómo intercalar dos listas ordenadas en una sola lista ordenada de forma eficiente?

Suponemos que se tienen estas dos listas de enteros ordenadas ascendentemente:

lista 1: 1 -> 3 -> 5 -> 6 -> 8 -> 9
lista 2: 0 -> 2 -> 6 -> 7 -> 10

Tras mezclarlas queda:

lista: 0 -> 1 -> 2 -> 3 -> 5 -> 6 -> 6 -> 7 -> 8 -> 9 -> 10

Esto se puede realizar mediante un único recorrido de cada lista, mediante dos punteros que recorren cada una. En el ejemplo anterior se insertan en este orden -salvo los dos 6 que puede variar según la implementación-: 0 (lista 2), el 1 (lista 1), el 2 (lista 2), el 3, 5 y 6 (lista 1), el 6 y 7 (lista 2), el 8 y 9 (lista 1), y por llegar al final de la lista 1, se introduce directamente todo lo que quede de la lista 2, que es el 10.

Segunda parte divide y vencerás.

Se separa la lista original en dos trozos del mismo tamaño (salvo listas de longitud impar) que se ordenan recursivamente, y una vez ordenados se fusionan obteniendo una lista ordenada. Como todo algoritmo basado en divide y vencerás tiene un caso base y un caso recursivo.

* Caso base: cuando la lista tiene 1 ó 0 elementos (0 se da si se trata de ordenar una lista vacía). Se devuelve la lista tal cual está.

* Caso recursivo: cuando la longitud de la lista es de al menos 2 elementos. Se **divide** la lista en dos trozos del mismo tamaño que se ordenan recursivamente. Una vez ordenado cada trozo, se fusionan y se devuelve la lista resultante.

El esquema es el siguiente:

```
Ordenar(lista L)
inicio
  si tamaño de L es 1 o 0 entonces
    devolver L
  si tamaño de L es >= 2 entonces
    separar L en dos trozos: L1 y L2.
    L1 = Ordenar(L1)
    L2 = Ordenar(L2)
    L = Fusionar(L1, L2)
    devolver L
fin
```

El algoritmo funciona y termina porque llega un momento en el que se obtienen listas de 2 ó 3 elementos que se dividen en dos listas de un elemento (1+1=2) y en dos listas de uno y dos elementos (1+2=3, la lista de 2 elementos se volverá a dividir), respectivamente. Por tanto se vuelve siempre de la recursión con listas ordenadas (pues tienen a lo sumo un elemento) que hacen que el algoritmo de fusión reciba siempre listas ordenadas.

Se incluye un ejemplo explicativo donde cada sublista lleva una etiqueta identificativa.

Dada: 3 -> 2 -> 1 -> 6 -> 9 -> 0 -> 7 -> 4 -> 3 -> 8 (lista original)
se **divide** en:
3 -> 2 -> 1 -> 6 -> 9 (lista 1)
0 -> 7 -> 4 -> 3 -> 8 (lista 2)
· se ordena recursivamente cada lista:
· 3 -> 2 -> 1 -> 6 -> 9 (lista 1)
· · se **divide** en:
· · 3 -> 2 -> 1 (lista 11)
· · 6 -> 9 (lista 12)
· · se ordena recursivamente cada lista:
· · 3 -> 2 -> 1 (lista 11)
· · · se **divide** en:
· · · 3 -> 2 (lista 111)
· · · 1 (lista 112)
· · · se ordena recursivamente cada lista:
· · · 3 -> 2 (lista 111)
· · · se **divide** en:
· · · 3 (lista 1111, que no se divide, caso base). Se devuelve 3
· · · 2 (lista 1112, que no se divide, caso base). Se devuelve 2
· · · se fusionan 1111-1112 y queda:
· · · 2 -> 3. Se devuelve 2 -> 3
· · · 1 (lista 112)
· · · 1 (lista 1121, que no se divide, caso base). Se devuelve 1
· · · se fusionan 111-112 y queda:
· · · 1 -> 2 -> 3 (lista 11). Se devuelve 1 -> 2 -> 3
· · 6 -> 9 (lista 12)
· · · se **divide** en:
· · · 6 (lista 121, que no se divide, caso base). Se devuelve 6
· · · 9 (lista 122, que no se divide, caso base). Se devuelve 9
· · · se fusionan 121-122 y queda:
· · · 6 -> 9 (lista 12). Se devuelve 6 -> 9
· · se fusionan 11-12 y queda:
· · 1 -> 2 -> 3 -> 6 -> 9. Se devuelve 1 -> 2 -> 3 -> 6 -> 9
· 0 -> 7 -> 4 -> 3 -> 8 (lista 2)
· ... tras repetir el mismo procedimiento se devuelve 0 -> 3 -> 4 -> 7 -> 8
· se fusionan 1-2 y queda:
· 0 -> 1 -> 2 -> 3 -> 3 -> 4 -> 6 -> 7 -> 8 -> 9, que se devuelve y se termina.

Este es un buen algoritmo de ordenación, pues no requiere espacio para una nueva lista y sólo las operaciones recursivas consumen algo de memoria. Es por tanto un **algoritmo ideal** para ordenar listas.

La complejidad es la misma en todos los casos, ya que no influye cómo esté ordenada la lista inicial -esto es, no existe ni mejor ni peor caso-, puesto que la intercalación de dos listas ordenadas siempre se realiza de una única pasada. La complejidad es proporcional a N·logN, característica de los algoritmos "Divide y Vencerás". Para hacer más eficiente el algoritmo es mejor realizar un primer recorrido sobre toda la lista para contar el número de elementos y añadir como parámetro a la función dicho número.

Algoritmos de búsqueda

La búsqueda de un elemento dentro de un array es una de las operaciones más importantes en el procesamiento de la información, y permite la recuperación de datos previamente almacenados. El tipo de búsqueda se puede clasificar como interna o externa, según el lugar en el que esté almacenada la información (en memoria o en dispositivos externos). Todos los algoritmos de búsqueda tienen dos finalidades:

- Determinar si el elemento buscado se encuentra en el conjunto en el que se busca.
- Si el elemento está en el conjunto, hallar la posición en la que se encuentra.

En este apartado nos centramos en la búsqueda interna. Como principales algoritmos de búsqueda en arrays tenemos la búsqueda secuencial, la binaria y la búsqueda utilizando tablas de hash.

Búsqueda secuencial

Consiste en recorrer y examinar cada uno de los elementos del array hasta encontrar el o los elementos buscados, o hasta que se han mirado todos los elementos del array.

```
for(i=j=0;i<N;i++)
  if(array[i]==elemento)
  {
    solucion[j]=i;
    j++;
  }
```

Este algoritmo se puede optimizar cuando el array está ordenado, en cuyo caso la condición de salida cambiaría a:

```
for(i=j=0;array[i]<=elemento;i++)
```

o cuando sólo interesa conocer la primera ocurrencia del elemento en el array:

```
for(i=0;i<N;i++)
  if(array[i]==elemento)
    break;
```

En este último caso, cuando sólo interesa la primera posición, se puede utilizar un centinela, esto es, dar a la posición siguiente al último elemento de array el valor del elemento, para estar seguro de que se encuentra el elemento, y no tener que comprobar a cada paso si seguimos buscando dentro de los límites del array:

```
array[N]=elemento;
for(i=0;;i++)
  if(array[i]==elemento)
    break;
```

Si al acabar el bucle, i vale N es que no se encontraba el elemento. El número medio de comparaciones que hay que hacer antes de encontrar el elemento buscado es de (N+1)/2.

Búsqueda binaria o dicotómica

Para utilizar este algoritmo, el array debe estar ordenado. La búsqueda binaria consiste en dividir el array por su elemento medio en dos subarrays más pequeños, y comparar el elemento con el del centro. Si coinciden, la búsqueda se termina. Si el elemento es menor, debe estar (si está) en el primer subarray, y si es mayor está en el segundo.

Por ejemplo, para buscar el elemento 3 en el array {1,2,3,4,5,6,7,8,9} se realizarían los siguientes pasos:

> Se toma el elemento central y se divide el array en dos:
> {1,2,3,4}-5-{6,7,8,9}
> Como el elemento buscado (3) es menor que el central (5), debe estar en el primer subarray: {1,2,3,4}
> Se vuelve a dividir el array en dos:
> {1}-2-{3,4}
> Como el elemento buscado es mayor que el central, debe estar en el segundo subarray: {3,4}
> Se vuelve a dividir en dos:
> {}-3-{4}
> Como el elemento buscado coincide con el central, lo hemos encontrado.
> Si al final de la búsqueda todavía no lo hemos encontrado, y el subarray a dividir está vacio {}, el elemento no se encuentra en el array

En general, este método realiza log(2,N+1) comparaciones antes de encontrar el elemento, o antes de descubrir que no está. Este número es muy inferior que el necesario para la búsqueda lineal para casos grandes.

Este método también se puede implementar de forma recursiva, siendo la función recursiva la que divide al array en dos más pequeños.

Búsqueda mediante transformación de claves (hashing)

Es un método de búsqueda que aumenta la velocidad de búsqueda, pero que no requiere que los elementos estén ordenados. Consiste en asignar a cada elemento un índice mediante una transformación del elemento. Esta correspondencia se realiza mediante una función de conversión, llamada función hash. La correspondencia más sencilla es la identidad, esto es, al número 0 se le asigna el índice 0, al elemento 1 el índice 1, y así sucesivamente. Pero si los números a almacenar son demasiado grandes esta función es inservible. Por ejemplo, se quiere guardar en un array la información de los 1000 usuarios de una empresa, y se elige el núemro de DNI como elemento identificativo. Es inviable hacer un array de 100.000.000 elementos, sobre todo porque se desaprovecha demasiado espacio. Por eso, se realiza una transformación al número de DNI para que nos de un número menor, por ejemplo coger las 3 últimas cifras para guardar a los empleados en un array de 1000 elementos. Para buscar a uno de ellos, bastaría con realizar la transformación a su DNI y ver si está o no en el array.

La función de hash ideal debería ser biyectiva, esto es, que a cada elemento le corresponda un índice, y que a cada índice le corresponda un elemento, pero no siempre es fácil encontrar esa función, e incluso a veces es inútil, ya que puedes no saber el número de elementos a almacenar. La función de hash depende de cada problema y de cada finalidad, y se pueden utilizar con números o cadenas, pero las más utilizadas son:

Restas sucesivas

esta función se emplea con claves numéricas entre las que existen huecos de tamaño conocido, obteniéndose direcciones consecutivas. Por ejemplo, si el número de expediente de un alumno universitario está formado por el año de entrada en la universidad, seguido de un número identificativo de tres cifras, y suponiendo que entran un máximo de 400 alumnos al año, se le asignarían las claves:

1998-000 --> 0 = 1998000-1998000
1998-001 --> 1 = 1998001-1998000
1998-002 --> 2 = 1998002-1998000
...
1998-399 --> 399 = 1998399-1998000
1999-000 --> 400 = 1999000-1998000+400
...
yyyy-nnn --> N = yyyynnn-1998000+(400*(yyyy-1998))

Aritmética modular

el índice de un número es resto de la división de ese número entre un número N prefijado, preferentemente primo. Los números se guardarán en las direcciones de memoria de 0 a N-1. Este método tiene el problema de que cuando hay N+1 elementos, al menos un índice es señalado por dos elementos (teorema del palomar). A este fenómeno se le llama colisión, y es tratado más adelante. Si el número N es el 13, los números siguientes quedan transformados en:

13000000 --> 0
12345678 --> 7

13602499 --> 1
71140205 --> 6
73062138 --> 6

Mitad del cuadrado

consiste en elevar al cuadrado la clave y coger las cifras centrales. Este método también presenta problemas de colisión:

123*123=15129 --> 51
136*136=18496 --> 84
730*730=532900 --> 29
301*301=90601 --> 06
625*625=390625 --> 06

Truncamiento:

consiste en ignorar parte del número y utilizar los elementos restantes como índice. También se produce colisión. Por ejemplo, si un número de 8 cifras se debe ordenar en un array de 1000 elementos, se pueden coger la primer, la tercer y la última cifras para formar un nuevo número:

13000000 --> 100
12345678 --> 138
13602499 --> 169
71140205 --> 715
73162135 --> 715

Plegamiento

consiste en dividir el número en diferentes partes, y operar con ellas (normalmente con suma o multiplicación). También se produce colisión. Por ejemplo, si dividimos los número de 8 cifras en 3, 3 y 2 cifras y se suman, dará otro número de tres cifras (y si no, se cogen las tres últimas cifras):

13000000 --> 130=130+000+00
12345678 --> 657=123+456+78
71140205 --> 118 --> 1118=711+402+05
13602499 --> 259=136+024+99
25000009 --> 259=250+000+09

Tratamiento de colisiones

Pero ahora se nos presenta el problema de qué hacer con las colisiones, qué pasa cuando a dos elementos diferentes les corresponde el mismo índice. Pues bien, hay tres posibles soluciones:

Cuando el índice correspondiente a un elemento ya está ocupada, se le asigna el primer índice libre a partir de esa posición. Este método es poco eficaz, porque al nuevo elemento se le asigna un índice que podrá estar ocupado por un elemento posterior a él, y la búsqueda se ralentiza, ya que no se sabe la posición exacta del elemento.

También se pueden reservar unos cuantos lugares al final del array para alojar a las colisiones. Este método también tiene un problema: ¿Cuánto espacio se debe reservar? Además, sigue la lentitud de búsqueda si el elemento a buscar es una colisión.

Lo más efectivo es, en vez de crear un array de número, crear un array de punteros, donde cada puntero señala el principio de una lista enlazada. Así, cada elemento que llega a un determinado índice se pone en el último lugar de la lista de ese índice. El tiempo de búsqueda se reduce considerablemente, y no hace falta poner restricciones al tamaño del array, ya que se pueden añadir nodos dinámicamente a la lista.

Algoritmos de backtracking

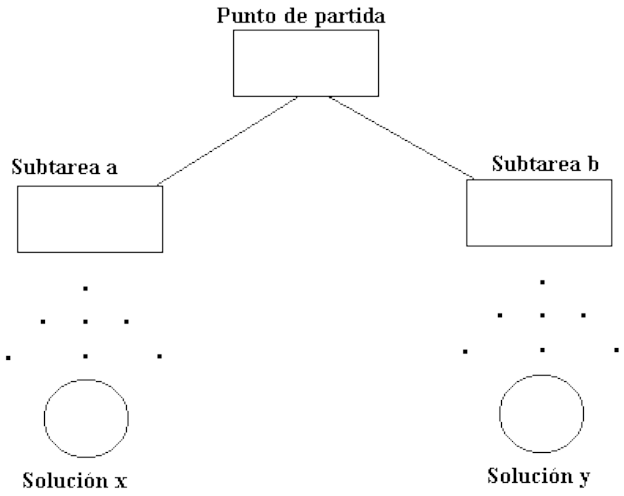
Los algoritmos de backtracking se utilizan para encontrar soluciones a un problema. No siguen unas reglas para la búsqueda de la solución, simplemente una búsqueda sistemática, que más o menos viene a significar que hay que probar todo lo posible hasta encontrar la solución o encontrar que no existe solución al problema. Para conseguir este propósito, se separa la búsqueda en varias búsquedas parciales o subtareas. Asimismo, estas subtareas suelen incluir más subtareas, por lo que el tratamiento general de estos algoritmos es de naturaleza recursiva.

¿Por qué se llaman algoritmos de backtracking?. Porque en el caso de no encontrar una solución en una subtarea se retrocede a la subtarea original y se prueba otra cosa distinta (una nueva subtarea distinta a las probadas anteriormente).

Puesto que a veces nos interesa conocer múltiples soluciones de un problema, estos algoritmos se pueden modificar fácilmente para obtener una única solución (si existe) o todas las soluciones posibles (si existe más de una) al problema dado.

Estos algoritmos se asemejan al recorrido en profundidad dentro de un grafo (ver sección de grafos, estructuras de datos, y recorrido de grafos, algoritmos), siendo cada subtarea un nodo del grafo. El caso es que el grafo no está definido de forma explícita (como lista o matriz de adyacencia), sino de forma implícita, es decir, que se irá creando según avance el recorrido. A menudo dicho grafo es un árbol, o no contiene ciclos, es decir, al buscar una solución es, en general, imposible llegar a una misma solución **x** partiendo de dos subtareas distintas **a** y **b**; o de la subtarea **a** es imposible llegar a la subtaréa **b** y viceversa.

Gráficamente se puede ver así:



A menudo ocurre que el árbol o grafo que se genera es tan grande que encontrar una solución o encontrar la mejor solución entre varias posibles es computacionalmente muy costoso. En estos casos suelen aplicarse una serie de restricciones, de tal forma que se puedan podar algunas de las ramas, es decir, no recorrer ciertas subtareas. Esto es posible si llegado a un punto se puede demostrar que la solución que se obtendrá a partir de ese punto no será mejor que la mejor solución obtenida hasta el momento. Si se hace correctamente, la poda no impide encontrar la mejor solución.

A veces, es imposible demostrar que al hacer una poda no se esté ocultando una buena solución. Sin embargo, el problema quizás no pida la mejor solución, sino una que sea razonablemente buena y cuyo coste computacional sea bastante reducido. Esa es una buena razón para aumentar las restricciones a la hora de recorrer un nodo. Tal vez se pierda la mejor solución, pero se encontrará una aceptable en un tiempo reducido.

Los algoritmos de backtracking tienen un esquema genérico, según se busque una o todas las soluciones, y puede adaptarse fácilmente según las necesidades de cada problema. A continuación se exponen estos esquemas, extraídos de Wirth. Los bloques se agrupan con **begin** y **end**, equivalentes a los corchetes de C, además están tabulados.

esquema para una solución:

```
procedimiento ensayar (paso : TipoPaso)
 repetir
 |  seleccionar_candidato
 |  if aceptable then
 |  begin
 |    anotar_candidato
 |    if solucion_incompleta then
 |    begin
 |      ensayar(paso_siguiente)
 |      if no acertado then borrar_candidato
 |    end
 |    else begin
 |      anotar_solucion
 |      acertado <- cierto;
 |  end
 hasta que (acertado = cierto) o (candidatos_agotados)
fin procedimiento
```

esquema para todas las soluciones:

```
procedimiento ensayar (paso : TipoPaso)
 para cada candidato hacer
 |  seleccionar candidato
 |  if aceptable then
 |  begin
 |    anotar_candidato
 |    if solucion_incompleta then
 |      ensayar(paso_siguiente)
 |    else
 |      almacenar_solucion
 |    borrar_candidato
 |  end
 hasta que candidatos_agotados
fin procedimiento
```

Concepto de divide y vencerás

La técnica de diseño de algoritmos llamada "divide y vencerás" (divide and conquer) consiste en descomponer el problema original en varios sub-problemas más sencillos, para luego resolver éstos mediante un cálculo sencillo. Por último, se combinan los resultados de cada sub-problema para obtener la solución del problema original. El pseudocódigo sería:

```
funcion divide_y_venceras_1(problema)
{
  descomponer el problema en n subproblemas más pequeños;
  para i=1 hasta n hacer
    resolver el subproblema k;
  combinar las n soluciones;
 }
```

Un ejemplo de "divide y vencerás" es la ordenación rápida, o quicksort, utilizada para ordenar arrays. En ella, se dividía el array en dos sub-arrays, para luego resolver cada uno por separado, y unirlos. El ahorro de tiempo es grande: el tiempo necesario para ordenar un array de elementos mediante el método de la burbuja es cuadrático: kN^2. Si dividimos el array en dos y ordenamos cada uno de ellos, el tiempo necesario para resolverlo es ahora $k(N/2)^2+k(N/2)^2=(kN^2)/2$. El tiempo necesario para ordenarlo es la mitad, pero sigue siendo cuadrático.

Pero ahora, si los subproblemas son todavía demasiado grandes, ¿por qué no utilizar la misma táctica con ellos, esto es, dividirlos a ellos también, utilizando un algoritmo recursivo que vaya dividiendo más el sub-problema hasta que su solución sea trivial?

Un algoritmo del tipo:

```
funcion divide_y_venceras(problema)
{
  si el problema es trivial
    entonces resolver el problema;
  si no es trivial
  {
    descomponer el problema en n subproblemas más pequeños;
    para i=1 hasta n hacer
      divide_y_venceras(subproblema_k);
    combinar las n soluciones;
  }
}
```

Si aplicamos este método al quicksort, el tiempo disminuye hasta ser logarítmico, con lo que el tiempo ahorrado es mayor cuanto más aumenta N.

Tiempo de ejecución

El tiempo de ejecución de un algoritmo de divide y vencerás, T(n), viene dado por la suma de dos elementos:

- El tiempo que tarda en resolver los A subproblemas en los que se divide el original, $A \cdot T(n/B)$, donde n/B es el tamaño de cada sub-problema.
- El tiempo necesario para combinar las soluciones de los sub-problemas para hallar la solución del original; normalmente es $O(n^k)$
- Por tanto, el tiempo total es: $T(n) = A \cdot T(n/B) + O(n^k)$. La solución de esta ecuación, si A es mayor o igual que 1 y B es mayor que 1, es:

  si $A>B^k$, $T(n) = O(n^{\log_B A})$
  si $A=B^k$, $T(n) = O(n^k \cdot \log n)$
  si $A<B^k$, $T(n) = O(n^k)$

Determinación del umbral

Uno de los aspectos que hay que tener en cuenta en los algoritmos de divide y vencerás es dónde colocar el umbral, esto es, cuándo se considera que un sub-problema es suficientemente pequeño como para no tener que dividirlo para resolverlo. Normalmente esto es lo que hace que un algoritmo de divide y vencerás sea efectivo o no. Por ejemplo, en el algoritmo de ordenación quicksort, cuando se tiene un array de longitud 3, es mejor ordenarlo utilizando otro algoritmo de ordenación (con 6 comparaciones se puede ordenar), ya que el quicksort debe dividirlo en dos sub-arrays y ordenar cada uno de ellos, para lo que utiliza más de 6 comparaciones.

Estructuras de datos

Para procesar información en un computador es necesario hacer una abstracción de los datos que tomamos del mundo real -abstracción en el sentido de que se ignoran algunas propiedades de los objetos reales, es decir, se simplifican-. Se hace una selección de los datos más representativos de la realidad a partir de los cuales pueda trabajar el computador para obtener unos resultados.

Cualquier lenguaje suministra una serie de tipos de datos simples, como son los números enteros, caracteres, números reales. En realidad suministra un subconjunto de éstos, pues la memoria del ordenador es finita. Los punteros (si los tiene) son también un tipo de datos. El tamaño de todos los tipos de datos depende de la máquina y del compilador sobre los que se trabaja.

En principio, conocer la representación interna de estos tipos de datos no es necesaria para realizar un programa, pero sí puede afectar en algunos casos al rendimiento.

¿Qué es una estructura de datos?

Se trata de un conjunto de variables de un determinado tipo agrupadas y organizadas de alguna manera para representar un comportamiento. Lo que se pretende con las estructuras de datos es facilitar un esquema lógico para manipular los datos en función del problema que haya que tratar y el algoritmo para resolverlo. En algunos casos la dificultad para resolver un problema radica en escoger la estructura de datos adecuada. Y, en general, la elección del algoritmo y de las estructuras de datos que manipulará estarán muy relacionadas.

Según su comportamiento durante la ejecución del programa distinguimos estructuras de datos:

- Estáticas: su tamaño en memoria es fijo. Ejemplo: arrays.
- Dinámicas: su tamaño en memoria es variable. Ejemplo: listas enlazadas con punteros, ficheros, etc.

Las estructuras de datos que trataremos aquí son los arrays, las pilas y las colas, los árboles, y algunas variantes de estas estructuras. La tabla que se encuentra al comienzo de esta página agrupa todas las estructuras de datos que emplearán los algoritmos explicados en esta web.

Tipos Abstractos de Datos

Los tipos abstractos de datos (TAD) permiten describir una estructura de datos en función de las operaciones que pueden efectuar, dejando a un lado su implementación.

Los TAD mezclan estructuras de datos junto a una serie de operaciones de manipulación. Incluyen una especificación, que es lo que verá el usuario, y una implementación (algoritmos de operaciones sobre las estructuras de datos y su representación en un lenguaje de programación), que el usuario no tiene necesariamente que conocer para manipular correctamente los tipos abstractos de datos.

Se caracterizan por el encapsulamiento. Es como una caja negra que funciona simplemente conectándole unos cables. Esto permite aumentar la complejidad de los programas pero manteniendo una claridad suficiente que no desborde a los desarrolladores. Además, en caso de que algo falle será más fácil determinar si lo que falla es la caja negra o son los cables.

Por último, indicar que un TAD puede definir a otro TAD. Por ejemplo, en próximos apartados se indicará como construir pilas, colas y árboles a partir de arrays y listas enlazadas. De hecho, las listas enlazadas también pueden construirse a partir de arrays y viceversa.

Recursividad

Se dice que algo es recursivo si se define en función de sí mismo o a sí mismo. También se dice que nunca se debe incluir la misma palabra en la definición de ésta. El caso es que las definiciones recursivas aparecen con frecuencia en matemáticas, e incluso en la vida real. Un ejemplo: basta con apuntar una cámara al monitor que muestra la imagen que muestra esa cámara. El efecto es verdaderamente curioso, en especial cuando se mueve la cámara alrededor del monitor.

En matemáticas, tenemos múltiples definiciones recursivas:

- Números naturales:
  (1) 1 es número natural.
  (2) el siguiente número de un número natural es un número natural
- El factorial: n!, de un número natural (incluido el 0):
  (1) si n = 0 entonces: 0! = 1
  (2) si n > 0 entonces: $n! = n \cdot (n-1)!$

Asimismo, puede definirse un programa en términos recursivos, como una serie de pasos básicos, o **paso base** (también conocido como condición de parada), y un **paso recursivo**, donde vuelve a llamarse al programa. En un computador, esta serie de pasos recursivos debe ser finita, terminando con un paso base. Es decir, a cada paso recursivo se reduce el número de pasos que hay que dar para terminar, llegando un momento en el que no se verifica la condición de paso a la recursividad. Ni el paso base ni el paso recursivo son necesariamente únicos.

Por otra parte, la recursividad también puede ser indirecta, si tenemos un procedimiento P que llama a otro Q y éste a su vez llama a P. También en estos casos debe haber una condición de parada.

Existen ciertas estructuras cuya definición es recursiva, tales como los árboles, y los algoritmos que utilizan árboles suelen ser en general recursivos.

¿Qué pasa si se hace una llamada recursiva que no termina?

Cada llamada recursiva almacena los parámetros que se pasaron al procedimiento, y otras variables necesarias para el correcto funcionamiento del programa. Por tanto si se produce una llamada recursiva infinita, esto es, que no termina nunca, llega un momento en el que no quedará memoria para almacenar más datos, y en ese momento se abortará la ejecución del programa. Para probar esto se puede intentar hacer esta llamada en el programa factorial definido anteriormente: factorial(-1);

Por supuesto no hay que pasar parámetros a una función que estén fuera de su dominio, pues el factorial está definido solamente para números naturales, pero es un ejemplo claro.

¿Cuándo utilizar la recursión?

Para empezar, algunos lenguajes de programación no admiten el uso de recursividad, como por ejemplo el ensamblador o el FORTRAN. Es obvio que en ese caso se requerirá una solución no recursiva (iterativa). Tampoco se debe utilizar cuando la solución iterativa sea clara a simple vista. Sin embargo, en otros casos, obtener una solución iterativa es mucho más complicado que una solución recursiva, y es entonces cuando se puede plantear la duda de si merece la pena transformar la solución recursiva en otra iterativa. Posteriormente se explicará como eliminar la recursión, y se basa en almacenar en una pila los valores de las variables locales que haya para un procedimiento en cada llamada recursiva. Esto reduce la claridad del programa. Aún así, hay que considerar que el compilador transformará la solución recursiva en una iterativa, utilizando una pila, para cuando compile al código del computador.
Por otra parte, casi todos los algoritmos basados en los esquemas de vuelta atrás y divide y vencerás son recursivos, pues de alguna manera parece mucho más natural una solución recursiva.

Aunque parezca mentira, es en general mucho más sencillo escribir un programa recursivo que su equivalente iterativo. Si el lector no se lo cree, posiblemente se deba a que no domine todavía la recursividad. Se propondrán diversos ejemplos de programas recursivos de diversa complejidad para acostumbrarse a la recursión.

Ejemplo

La famosa sucesión de Fibonacci puede definirse en términos de recurrencia de la siguiente manera:

(1) Fib(1) = 1 ; Fib(0) = 0
(2) Fib(n) = Fib(n-1) + Fib(n-2) si n >= 2

Arrays

Un array es un tipo de estructura de datos que consta de un número fijo de elementos del mismo tipo. En una máquina, dichos elementos se almacenan en posiciones contiguas de memoria. Estos elementos pueden ser variables o estructuras. Para definirlos se utiliza la expresión:

```
tipo_de_elemento nombre_del_array[número_de_elementos_del_array];
int mapa[100];
```

Cada uno de los elementos de los que consta el array tiene asignado un número (índice). El primer elemento tiene índice 0 y el último tiene índice número_de_elementos_del_array-1. Para acceder a ese elemento se pone el nombre del array con el índice entre corchetes:

```
nombre_del_array[índice]
mapa[5]
```

Los elementos no tienen por qué estar determinados por un solo índice, pueden estarlo por dos (por ejemplo, fila y columna), por tres (p.e. las tres coordenadas en el espacio), o incluso por más. A estos arrays definidos por más de un índice se le llaman arrays multidimensionales o matrices, y se definen:

```
tipo_de_elemento nombre_del_array[número1] [número2]... [númeroN];
int mapa[100][50][399];
```

Y para acceder a un elemento de índices i1,i2,...,iN, la expresión es similar:

```
nombre_del_array[i1][i2]...[iN]
mapa[34][2][0]
```

Hay que tener cuidado con no utilizar un índice fuera de los límites, porque dará resultados inesperados (tales como cambio del valor de otras variables o finalización del programa, con error "invalid memory reference").

Listas

Una lista es una estructura de datos secuencial.

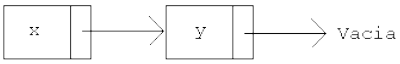
Una manera de clasificarlas es por la forma de acceder al siguiente elemento:

- *lista densa*: la propia estructura determina cuál es el siguiente elemento de la lista. Ejemplo: un array.
- *lista enlazada*: la posición del siguiente elemento de la estructura la determina el elemento actual. Es necesario almacenar al menos la posición de memoria del primer elemento. Además es dinámica, es decir, su tamaño cambia durante la ejecución del programa.

Una **lista enlazada** se puede definir recursivamente de la siguiente manera:

- una lista enlazada es una estructura vacía o
- un elemento de información y un enlace hacia una lista (un nodo).

Gráficamente se suele representar así:

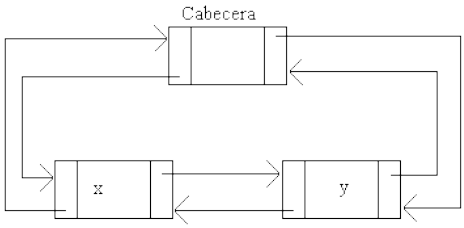


Como se ha dicho anteriormente, pueden cambiar de tamaño, pero su ventaja fundamental es que son flexibles a la hora de reorganizar sus elementos; a cambio se ha de pagar una mayor lentitud a la hora de acceder a cualquier elemento.

En la lista de la figura anterior se puede observar que hay dos elementos de información, x e y. Supongamos que queremos añadir un nuevo nodo, con la información p, al comienzo de la lista. Para hacerlo basta con crear ese nodo, introducir la información p, y hacer un enlace hacia el siguiente nodo, que en este caso contiene la información x.
¿Qué ocurre si quisiéramos hacer lo mismo sobre un array?. En ese caso sería necesario desplazar todos los elementos de información "hacia la derecha", para poder introducir el nuevo elemento, una operación muy engorrosa.

Listas doblemente enlazadas

Son listas que tienen un enlace con el elemento siguiente y con el anterior. Una ventaja que tienen es que pueden recorrerse en ambos sentidos, ya sea para efectuar una operación con cada elemento o para insertar/actualizar y borrar. La otra ventaja es que las búsquedas son algo más rápidas puesto que no hace falta hacer referencia al elemento anterior. Su inconveniente es que ocupan más memoria por nodo que una lista simple.



Pilas (Stack)

Una pila es una estructura de datos de acceso restrictivo a sus elementos. Se puede entender como una pila de libros que se amontonan de abajo hacia arriba. En principio no hay libros; después ponemos uno, y otro encima de éste, y así sucesivamente. Posteriormente los solemos retirar empezando desde la cima de la pila de libros, es decir, desde el último que pusimos, y terminaríamos por retirar el primero que pusimos, posiblemente ya cubierto de polvo.

En los programas estas estructuras suelen ser fundamentales. La recursividad se simula en un computador con la ayuda de una pila. Asimismo muchos algoritmos emplean las pilas como estructura de datos fundamental, por ejemplo para mantener una lista de tareas pendientes que se van acumulando.

Las pilas ofrecen dos operaciones fundamentales, que son apilar y desapilar sobre la cima. El uso que se les de a las pilas es independiente de su implementación interna. Es decir, se hace un encapsulamiento. Por eso se considera a la pila como un tipo abstracto de datos.

Es una estructra de tipo LIFO (Last In First Out), es decir, último en entrar, primero en salir.

Colas (Queue)

Una cola es una estructura de datos de acceso restrictivo a sus elementos. Un ejemplo sencillo es la cola del cine o del autobús, el primero que llegue será el primero en entrar, y afortunadamente en un sistema informático no se cuela nadie salvo que el programador lo diga.

Las colas serán de ayuda fundamental para ciertos recorridos de árboles y grafos.

Las colas ofrecen dos operaciones fundamentales, que son encolar (al final de la cola) y desencolar (del comienzo de la cola). Al igual que con las pilas, la implementación de las colas suele encapsularse, es decir, basta con conocer las operaciones de manipulación de la cola para poder usarla, olvidando su implementación interna.

Es una estructra de tipo FIFO (First In First Out), es decir: primero en entrar, primero en salir.

Arboles (Tree)

Un árbol es una estructura de datos, que puede definirse de forma recursiva como:

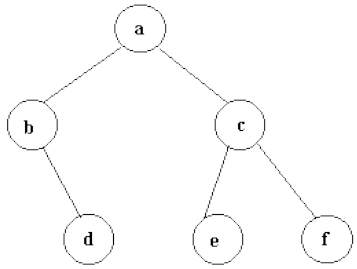
- Una estructura vacía o
- Un elemento o clave de información (nodo) más un número finito de estructuras tipo árbol, disjuntos, llamados subárboles. Si dicho número de estructuras es inferior o igual a 2, se tiene un árbol binario.

Es, por tanto, una estructura no secuencial.

Otra definición nos da el árbol como un tipo de grafo un árbol es un grafo acíclico, conexo y no dirigido. Es decir, es un grafo no dirigido en el que existe exactamente un camino entre todo par de nodos. Esta definición permite implementar un árbol y sus operaciones empleando las representaciones que se utilizan para los grafos. Sin embargo, en esta sección no se tratará esta implementación.

Formas de representación

- Mediante un grafo:



En la computación se utiliza mucho una estructura de datos, que son los árboles binarios. Estos árboles tienen 0, 1 ó 2 descendientes como máximo. El árbol de la figura anterior es un ejemplo válido de árbol binario.

Nomenclatura sobre árboles

- **Raíz**: es aquel elemento que no tiene antecesor; ejemplo: a.
- **Rama**: arista entre dos nodos.
- **Antecesor**: un nodo X es es antecesor de un nodo Y si por alguna de las ramas de X se puede llegar a Y.
- **Sucesor**: un nodo X es sucesor de un nodo Y si por alguna de las ramas de Y se puede llegar a X.
- **Grado de un nodo**: el número de descendientes directos que tiene. Ejemplo: c tiene grado 2, d tiene grado 0, a tiene grado 2.
- **Hoja**: nodo que no tiene descendientes: grado 0. Ejemplo: d
- **Nodo interno**: aquel que tiene al menos un descendiente.
- **Nivel**: número de ramas que hay que recorrer para llegar de la raíz a un nodo. Ejemplo: el nivel del nodo a es 1 (es un convenio), el nivel del nodo e es 3.
- **Altura**: el nivel más alto del árbol. En el ejemplo de la figura 1 la altura es 3.
- **Anchura**: es el mayor valor del número de nodos que hay en un nivel. En la figura, la anchura es 3.

Aclaraciones: se ha denominado a a la raíz, pero se puede observar según la figura que cualquier nodo podría ser considerado raíz, basta con girar el árbol. Podría determinarse por ejemplo que b fuera la raíz, y a y d los sucesores inmediatos de la raíz b. Sin embargo, en las implementaciones sobre un computador que se realizan a continuación es necesaria una jerarquía, es decir, que haya una única raíz.

Arboles binarios

Un **árbol binario** es una estructura de datos en la cual cada nodo siempre tiene un hijo izquierdo y un hijo derecho. No pueden tener más de dos hijos (de ahí el nombre "binario"). Si algún hijo tiene como referencia a **null**, es decir que no almacena ningún dato, entonces este es llamado un nodo externo. En el caso contrario el hijo es llamado un nodo interno. Usos comunes de los árboles binarios son los árboles binarios de búsqueda, los montículos binarios y Codificación de Huffman

Tipos de arboles binarios

- Un **árbol binario** es un árbol **con raíz** en el que cada nodo tiene como máximo dos hijos.
- Un **árbol binario lleno** es un árbol en el que cada nodo tiene cero o dos hijos.
- Un **árbol binario perfecto** es un árbol binario lleno en el que todas las **hojas** (vértices con cero hijos) están a la misma profundidad (distancia desde la **raíz**, también llamada **altura**).
- A veces un árbol binario perfecto es denominado **árbol binario completo**. Otros definen un árbol **binario completo** como un árbol binario lleno en el que todas las hojas están a profundidad *n* o *n-1*, para alguna *n*.

Un **árbol binario** es un árbol en el que ningún nodo puede tener más de dos subárboles. En un árbol binario cada nodo puede tener cero, uno o dos hijos (subárboles). Se conoce el nodo de la izquierda como hijo izquierdo y el nodo de la derecha como hijo derecho

Arboles AVL

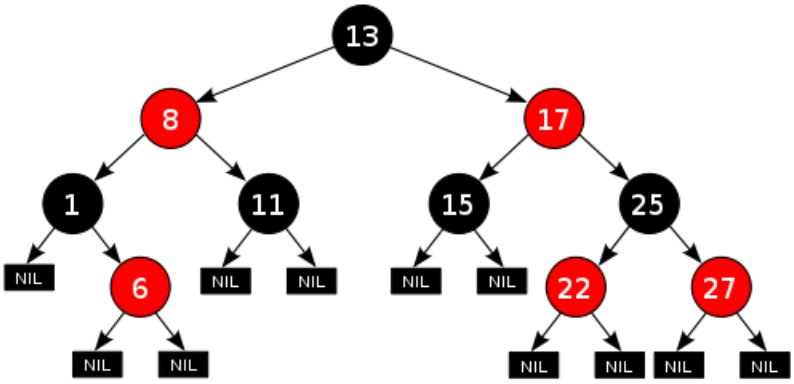
El **árbol AVL** toma su nombre de las iniciales de los apellidos de sus inventores, **A**delson-**V**elskii y **L**andis. Lo dieron a conocer en la publicación de un artículo en 1962: "An algorithm for the organization of information" ("Un algoritmo para la organización de la información").

Los **árboles AVL** están siempre equilibrados de tal modo que para todos los nodos, la altura de la rama izquierda no difiere en más de una unidad de la altura de la rama derecha o viceversa. Gracias a esta forma de equilibrio (o balanceo), la complejidad de una búsqueda en uno de estos árboles se mantiene siempre en orden de complejidad O(log n). El **factor de equilibrio** puede ser almacenado directamente en cada nodo o ser computado a partir de las alturas de los subárboles.

Para conseguir esta propiedad de equilibrio, la inserción y el borrado de los nodos se ha de realizar de una forma especial. Si al realizar una operación de inserción o borrado se rompe la condición de equilibrio, hay que realizar una serie de rotaciones de los nodos.

Arboles rojo-negro (Red-Black)

Un **árbol rojo negro** es un tipo abstracto de datos, concretamente es un árbol binario de búsqueda equilibrado, una estructura de datos utilizada en informática y ciencias de la computación. La estructura original fue creada por Rudolf Bayer en 1972, que le dio el nombre de "árboles-B binarios simétricos", pero tomó su nombre moderno en un trabajo deLeo J. Guibas y Robert Sedgewick realizado en 1978. Es complejo, pero tiene un buen peor caso de tiempo de ejecución para sus operaciones y es eficiente en la práctica. Puede buscar, insertar y borrar en un tiempo O(log n), donde n es el número de elementos del árbol.



Terminología

Un árbol rojo-negro es un tipo especial de árbol binario usado en informática para organizar información compuesta por datos comparables (como por ejemplo números).

En los árboles rojo-negro las hojas no son relevantes y no contienen datos. A la hora de implementarlo en un lenguaje de programación, para ahorrar memoria, un único nodo (nodo-centinela) hace de nodo hoja para todas las ramas. Así,todas las referencias de los nodos internos a las hojas van a parar al nodo centinela.

En los árboles rojo-negro, como en todos los árboles binarios de búsqueda, es posible moverse ordenadamente a través de los elementos de forma eficiente si hay forma de localizar el padre de cualquier nodo. El tiempo de desplazarse desde la raíz hasta una hoja a través de un árbol equilibrado que tiene la mínima altura posible es de O(log n).

Propiedades

Un árbol rojo-negro es un árbol binario de búsqueda en el que cada nodo tiene un atributo de color cuyo valor es o bien **rojo** o bien **negro**. Además de los requisitos impuestos a los árboles binarios de búsqueda convencionales, se deben satisfacer los siguientes para tener un árbol rojo-negro válido:

- Todo nodo es o bien rojo o bien negro.
- La raíz es negra.
- Todas las hojas son negras (las hojas son los hijos nulos).
- Los hijos de todo nodo rojo son negros (también llamada "Propiedad del rojo").

Cada camino simple desde un nodo a una hoja descendiente contiene el mismo número de nodos negros, ya sea contando siempre los nodos negros nulos, o bien no contándolos nunca (el resultado es equivalente). También es llamada "Propiedad del camino", y al número de nodos negros de cada camino, que es constante para todos los caminos, se le denomina "Altura negra del árbol", y por tanto el camino no puede tener dos rojos seguidos.

El camino más largo desde la raíz hasta una hoja no es más largo que 2 veces el camino más corto desde la raíz del árbol a una hoja en dicho árbol. El resultado es que dicho árbol está aproximadamente equilibrado.

Dado que las operaciones básicas como insertar, borrar y encontrar valores tienen un peor tiempo de búsqueda proporcional a la altura del árbol, esta cota superior de la altura permite a los árboles rojo-negro ser eficientes en el peor caso, de forma contraria a lo que sucede en los árboles binarios de búsqueda. Para ver que estas propiedades garantizan lo dicho, basta ver que ningún camino puede tener 2 nodos rojos seguidos debido a la propiedad 4. El camino más corto posible tiene todos sus nodos negros, y el más largo alterna entre nodos rojos y negros. Como todos los caminos máximos tienen el mismo número de nodos negros, por la propiedad 5, esto muestra que no hay ningún camino que pueda tener el doble de longitud que otro camino.

En muchas presentaciones de estructuras arbóreas de datos, es posible para un nodo tener solo un hijo y las hojas contienen información. Es posible presentar los árboles rojo-negro en este paradigma, pero cambian algunas de las propiedades y se complican los algoritmos. Por esta razón, este artículo utilizan "hojas nulas", que no contienen información y simplemente sirven para indicar dónde el árbol acaba, como se mostró antes. Habitualmente estos nodos son omitidos en las representaciones, lo cual da como resultado un árbol que parece contradecir los principios expuestos antes, pero que realmente no los contradice. Como consecuencia de esto todos los nodos internos tienen dos hijos, aunque uno o ambos nodos podrían ser una hoja nula.

Otra explicación que se da del árbol rojo-negro es la de tratarlo como un árbol binario de búsqueda cuyas aristas, en lugar de nodos, son coloreadas de color rojo o negro, pero esto no produce ninguna diferencia. El color de cada nodo en la terminología de este artículo corresponde al color de la arista que une el nodo a su padre, excepto la raíz, que es siempre negra (por la propiedad 2) donde la correspondiente arista no existe.

Arboles B

La idea tras los árboles-B es que los nodos internos deben tener un número variable de nodos hijo dentro de un rango predefinido. Cuando se inserta o se elimina un dato de la estructura, la cantidad de nodos hijo varía dentro de un nodo. Para que siga manteniéndose el número de nodos dentro del rango predefinido, los nodos internos se juntan o se parten. Dado que se permite un rango variable de nodos hijo, los árboles-B no necesitan rebalancearse tan frecuentemente como los árboles binarios de búsqueda auto-balanceables, pero por otro lado pueden desperdiciar memoria, porque los nodos no permanecen totalmente ocupados. Los límites superior e inferior en el número de nodos hijo son definidos para cada implementación en particular. Por ejemplo, en un **árbol-B 2-3** (A menudo simplemente llamado árbol 2-3), cada nodo sólo puede tener 2 ó 3 nodos hijo.

Un árbol-B se mantiene balanceado porque requiere que todos los nodos hoja se encuentren a la misma altura.

Los árboles B tienen ventajas sustanciales sobre otras implementaciones cuando el tiempo de acceso a los nodos excede al tiempo de acceso entre nodos. Este caso se da usualmente cuando los nodos se encuentran en dispositivos de almacenamiento secundario como los discos rígidos. Al maximizar el número de nodos hijo de cada nodo interno, la altura del árbol decrece, las operaciones para balancearlo se reducen, y aumenta la eficiencia. Usualmente este valor se coloca

de forma tal que cada nodo ocupe un bloque de disco, o un tamaño análogo en el dispositivo. Mientras que los árboles B 2-3 pueden ser útiles en la memoria principal, y además más fáciles de explicar, si el tamaño de los nodos se ajustan para caber en un bloque de disco, el resultado puede ser un árbol B 129-513.

Los creadores del árbol B, Rudolf Bayer y Ed McCreight, no han explicado el significado de la letra B de su nombre. Se cree que la B es de balanceado, dado que todos los nodos hoja se mantienen al mismo nivel en el árbol. La B también puede referirse a Bayer, o a Boeing, porque sus creadores trabajaban en el Boeing Scientific Research Labs en ese entonces.

B-árbol es un árbol de búsqueda que puede estar vacío o aquel cuyos nodos pueden tener varios hijos, existiendo una relación de orden entre ellos, tal como muestra el dibujo.

Un árbol-B de orden M (el máximo número de hijos que puede tener cada nodo) es un árbol que satisface las siguientes propiedades:

- Cada nodo tiene como máximo M hijos.
- Cada nodo (excepto raíz y hojas) tiene como mínimo (M+1)/2 hijos.
- La raíz tiene al menos 2 hijos si no es un nodo hoja.
- Todos los nodos hoja aparecen al mismo nivel.
- Un nodo no hoja con k hijos contiene k-1 elementos almacenados.
- Los hijos que cuelgan de la raíz (r1, ⋯, rm) tienen que cumplir ciertas condiciones:
- El primero tiene valor menor que r1.
- El segundo tiene valor mayor que r1 y menor que r2, etc.
- El último hijo tiene valor mayor que rm.

Arboles B+

En informática, un **árbol-B** es un tipo de estructura de datos de árboles. Representa una colección de datos ordenados de manera que se permite una inserción y borrado eficientes de elementos. Es un índice, multinivel, dinámico, con un límite máximo y mínimo en el número de claves por nodo.

Un árbol-B+ es una variación de un árbol-B. En un árbol-B+, en contraste respecto un árbol-B, toda la información se guarda en las hojas. Los nodos internos sólo contienen claves y punteros. Todas las hojas se encuentran en el mismo, más bajo nivel. Los nodos hoja se encuentran unidos entre sí como una lista enlazada para permitir búsqueda secuencial.

- El número máximo de claves en un registro es llamado el orden del árbol-B+.
- El mínimo número de claves por registro es la mitad del máximo número de claves. Por ejemplo, si el orden de un árbol-B+ es n, cada nodo (exceptuando la raíz) debe tener entre n/2 y n claves.
- El número de claves que pueden ser indexadas usando un árbol-B+ está en función del orden del árbol y su altura.

*Arboles B**

Un árbol-B* es una estructura de datos de árbol, una variante de Árbol-B utilizado en los sistemas de ficheros HFS y Reiser4, que requiere que los nodos no raíz estén por lo menos a 2/3 de ocupación en lugar de 1/2. Para mantener esto los nodos, en lugar de generar inmediatamente un nodo cuando se llenan, comparten sus claves con el nodo adyacente. Cuando ambos están llenos, entonces los dos nodos se transforman en tres. También requiere que la clave más a la izquierda no sea usada nunca.

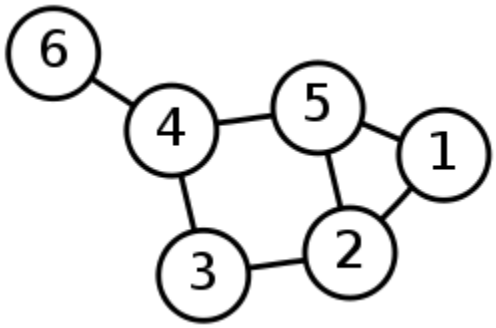
No se debe confundir un árbol-B* con un árbol-B+, en el que los nodos hoja del árbol están conectados entre sí a través de una lista enlazada, aumentando el coste de inserción para mejorar la eficiencia en la búsqueda.

Grafos

Un grafo es un objeto matemático que se utiliza para representar circuitos, redes, etc. Los grafos son muy utilizados en computación, ya que permiten resolver problemas muy complejos.

Imaginemos que disponemos de una serie de ciudades y de carreteras que las unen. De cada ciudad saldrán varias carreteras, por lo que para ir de una ciudad a otra se podrán tomar diversos caminos. Cada carretera tendrá un coste asociado (por ejemplo, la longitud de la misma). Gracias a la representación por grafos podremos elegir el camino más corto que conecta dos ciudades, determinar si es posible llegar de una ciudad a otra, si desde cualquier ciudad existe un camino que llegue a cualquier otra, etc.

El estudio de grafos es una rama de la algoritmia muy importante. Estudiaremos primero sus rasgos generales y sus recorridos fundamentales, para tener una buena base que permita comprender los algoritmos que se pueden aplicar.



Glosario

Un grafo consta de vértices (o nodos) y aristas. Los vértices son objetos que contienen información y las aristas son conexiones entre vértices. Para representarlos, se suelen utilizar puntos para los vértices y líneas para las conexiones, aunque hay que recordar siempre que la definición de un grafo no depende de su representación.

Un camino entre dos vértices es una lista de vértices en la que dos elementos sucesivos están conectados por una arista del grafo. Así, el camino AJLOE es un camino que comienza en el vértice

A y pasa por los vértices J,L y O (en ese orden) y al final va del O al E. El grafo será conexo si existe un camino desde cualquier nodo del grafo hasta cualquier otro. Si no es conexo constará de varias componentes conexas.

Un camino simple es un camino desde un nodo a otro en el que ningún nodo se repite (no se pasa dos veces). Si el camino simple tiene como primer y último elemento al mismo nodo se denomina ciclo. Cuando el grafo no tiene ciclos tenemos un árbol. Varios árboles independientes forman un bosque. Un árbol de expansión de un grafo es una reducción del grafo en el que solo entran a formar parte el número mínimo de aristas que forman un árbol y conectan a todos los nodos.

Según el número de aristas que contiene, un grafo es completo si cuenta con todas las aristas posibles (es decir, todos los nodos están conectados con todos),disperso si tiene relativamente pocas aristas y denso si le faltan pocas para ser completo.

Las aristas son la mayor parte de las veces bidireccionales, es decir, si una arista conecta dos nodos A y B se puede recorrer tanto en sentido hacia B como en sentido hacia A: estos son llamados grafos no dirigidos. Sin embargo, en ocasiones tenemos que las uniones son unidireccionales. Estas uniones se suelen dibujar con una flecha y definen un grafo dirigido. Cuando las aristas llevan un coste asociado (un entero al que se denomina peso) el grafo es ponderado. Una red es un grafo dirigido y ponderado.

Representación de grafos

Una característica especial en los grafos es que podemos representarlos utilizando dos estructuras de datos distintas. En los algoritmos que se aplican sobre ellos veremos que adoptarán tiempos distintos dependiendo de la forma de representación elegida. En particular, los tiempos de ejecución variarán en función del número de vértices y el de aristas, por lo que la utilización de una representación u otra dependerá en gran medida de si el grafo es denso o disperso.

Para nombrar los nodos utilizaremos letras mayúsculas, aunque en el código deberemos hacer corresponder cada nodo con un entero entre 1 y V (número de vértices) de cara a la manipulación de los mismos.

Algoritmos en grafos

Los algoritmos que se tratan en este texto son fundamentales y son muy útiles en muchas aplicaciones, pero solamente son una introducción al tema de algoritmos de grafos.

Existe un gran variedad de problemas relacionados con grafos y una gran variedad de algoritmos para procesamiento de grafos, pero claramente no todo problema de grafos es sencillo de resolver y en muchas ocasiones tampoco es sencillo determinar que tan difícil puede ser resolverlo.

Podemos encontrar una categorización de problemas de grafos de acuerdo al grado de dificultad para resolverlos, a saber:

- **Fáciles**: Un problema fácil de procesamiento de grafos es aquel que se puede resolver utilizando un programa eficiente y elegante. Frecuentemente su tiempo de ejecución es lineal en el peor caso, o limitado por un polinomio de bajo grado en el número de nodos o el número de aristas. Generalmente, también podemos decir que el problema es fácil si podemos desarrollar un algoritmo de fuerza bruta que aunque sea lento para grandes grafos, es útil para grafos pequeños e inclusive de tamaño medio. Entonces, una vez que sabemos que el problema es fácil, buscamos soluciones eficientes y escogemos la mejor de ellas.
- **Tratable**: Un problema tratable de procesamiento de grafos es aquel para el que se conoce un algoritmo que garantiza que sus requerimientos en tiempo y espacio están limitados por una función polinomial en el tamaño del grafo (número de nodos + número de aristas). Todo problema fácil es tratable, pero se hace la distinción debido a que el desarrollo de una solución eficiente para resolverlo es extremadamente difícil o imposible. Las soluciones a algunos problemas intratables nunca han sido escritas en programas, o tiempo tiempos de ejecución tan altos que no puede contemplarse su utilización en la práctica.
- **Intratable**: Un problema intratable de procesamiento de grafos es aquel para el que no se conoce algoritmo que garantice obtener un solución del problema en una cantidad razonable de tiempo. Muchos de estos problemas tienen la característica de que podemos utilizar un método de fuerza bruta para probar todas las posibilidades de calcular la solución, y se consideran intratables porque existen demasiadas posibilidades a considerar.

  Esta clase de problemas es extensa y muchos expertos piensan que no existen algoritmos eficientes para solucionar estos problemas. El término NP-hard describe los problemas de esta clase, el cual representa un altísimo nivel de dificultad.
- **Desconocida**: Existen problemas de procesamiento de grafos cuya dificultad es desconocida. No hay un algoritmo eficiente conocido para resolverlos, ni son conocidos como NP-hard. El problema de isomorfismo de grafos pertenece a esta clase.

Algunos de los problemas más conocidos de procesamiento de grafos son:

- **Conectividad Simple**: Consiste en estudiar si el grafo es conexo, es decir, si existe al menos un camino entre cada par de vértices.
- **Detección de Ciclos**: Consiste en estudiar la existencia de al menos un ciclo en el grafo
- **Camino Simple**: Consiste en estudiar la existencia de un camino entre dos vértices cualquiera.
- **Camino de Euler**: Consiste en estudiar la existencia de un camino que conecte dos vértices dados usando cada arista del grafo exactamente una sola vez. Si el camino tiene como inicio y final el mismo vértice, entonces se desea encontrar un tour de Euler.
- **Camino de Hamilton**: Consiste en estudiar la existencia de un camino que conecte dos vértices dados que visite cada nodo del grafo exactamente una vez. Si el camino tiene como inicio y final el mismo vértice, entonces se desea encontrar un tour de Hamilton.

- **Conectividad Fuerte en Dígrafos**: Consiste en estudiar si hay un camino dirigido conectando cada par de vértices del dígrafo. Inclusive se puede estudiar si existe un camino dirigido entre cada par de vértices, en ambas direcciones.8
- **Clausura Transitiva**: Consiste en tratar de encontrar un conjunto de vértices que pueda ser alcanzado siguiendo aristas dirigidas desde cada vértice del dígrafo.
- **Árbol de Expansión Mínima**: Consiste en encontrar, en un grafo pesado, el conjunto de aristas de peso mínimo que conecta a todos los vértices.
- **Caminos cortos a partir de un mismo origen**: Consiste en encontrar cuales son los caminos más cortos conectando a un vértice v cualquier con cada uno de los otros vértices de un dígrafo pesado. Este es un problema que por lo general se presenta en redes de computadores, representadas como grafos.
- **Planaridad**: Consiste en estudiar si un grafo puede ser dibujado sin que ninguna de las líneas que representan las aristas se intercepten.
- **Pareamiento (Matching)**: Dado un grafo, consiste en encontrar cual es el subconjunto más largo de sus aristas con las propiedad de que no haya dos conectados al mismo vértice. Se sabe que este problema clásico es resoluble en tiempo proporcional a una función polinomial en el número de vértices y de aristas, pero aun no existe un algoritmo rápido que se ajuste a grandes grafos.
- **Ciclos Pares en Dígrafos**: Consiste en encontrar en un dígrafo un camino de longitud par. Este problema puede lucir simple ya que la solución para grafos no dirigidos es sencilla. Sin embargo, aun no se conoce si existe un algoritmoeficiente para resolverlo.
- **Asignación**: Este problema se conoce también como pareamiento bipartitopesado (bipartite weigthed matching). Consiste en encontrar un pareamientoperfecto de peso mínimo en un grafo bipartito. Un grafo bipartito es aquel cuyos vértices se pueden separar en dos conjuntos, de tal manera que todas las aristas conecten a un vértice en un conjunto con otro vértice en el otro conjunto.
- **Conectividad General**: Consiste en encontrar el número mínimo de aristas que al ser removidas separarán el grafo en dos partes disjuntas (conectividad de aristas). También se puede encontrar el número mínimo de nodos que al ser removidos separarán el grafo en dos partes disjuntas (conectividad de nodos).
- **El camino más largo**: Consiste en encontrar cual es el camino más largo que conecte a dos nodos dados en el grafo. Aunque parece sencillo, este problema es una versión del problema del tour de Hamilton y es NP-hard.
- **Colorabilidad**: Consiste en estudiar si existe alguna manera de asignar k colores a cada uno de los vértices de un grafo, de tal forma de que ninguna arista conecte dos vértices del mismo color. Este problema clásico es fácil para k=2 pero es NP-hard para k=3. 9
- **Conjunto Independiente**: Consiste en encontrar el tamaño del mayor subconjunto de nodos de un grafo con la propiedad de que no haya ningún par conectado por una arista. Este problema es NP-hard.
- **Clique**: Consiste en encontrar el tamaño del clique (subgrafo completo) másgrande en un grafo dado.

- **Isomorfismo de grafos**: Consiste en estudiar la posibilidad de hacer dos grafos idénticos con solo renombrar sus nodos. Se conocen algoritmos eficientes para solucionar este problema, para varios clases particulares de grafos, pero no setiene solución para el problema general. Este problema es NP-hard.