Introducción al estudio de la complejidad de 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 una computadora.
- *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 la computadora.
- *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.

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

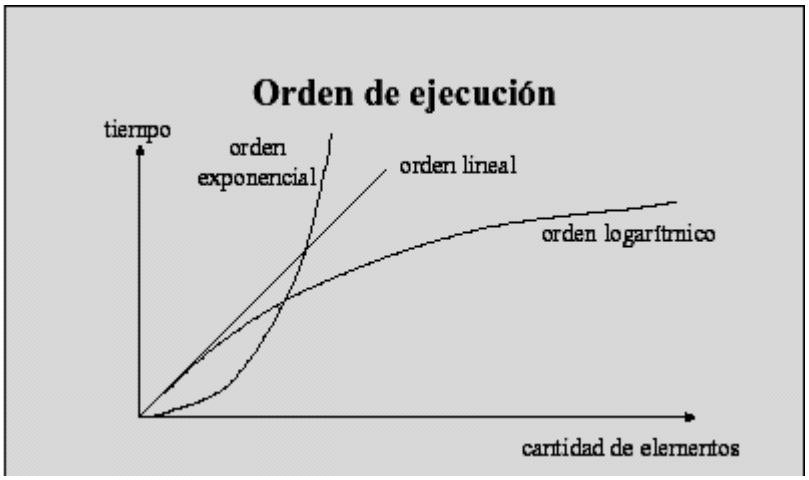
* Algoritmo determinista: en cada paso del algoritmo se determina de forma única el siguiente paso.
* Algoritmo no determinista: deben decidir en cada paso de la ejecución entre varias alternativas y agotarlas todas antes de encontrar la solución.

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.
- **log N** : 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 · log N** : El tiempo de ejecución es N·logN. Si N se duplica, el tiempo de ejecución es ligeramente mayor del doble.
- **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.
- **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.
- **2^N** : Tiempo de ejecución exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. 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 $O(N^2)$.

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.

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* $N >= N_0$.

BUSQUEDA

Cuestiones generales

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:

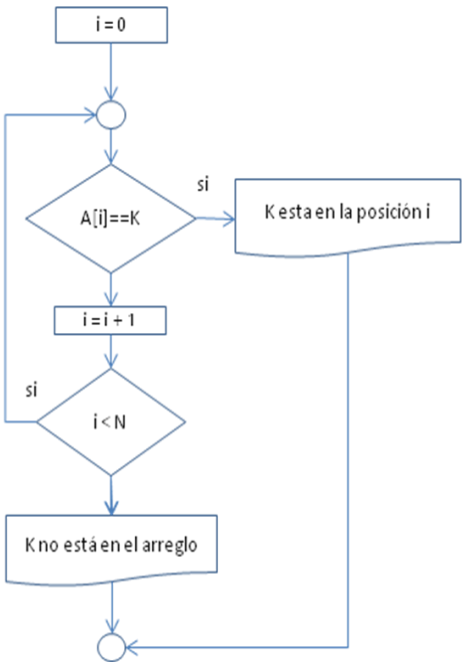
- Determinar si el elemento buscado se encuentra en el conjunto en el que se busca.
- Si el elemento está en el conjunto, hallar la posición en la que se encuentra.

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, está última será estudiada en materias posteriores.

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. Si no hay elementos repetidos o sólo interesa conocer la primera ocurrencia del elemento en el array:

A: array, i: índice, k: dato a buscar

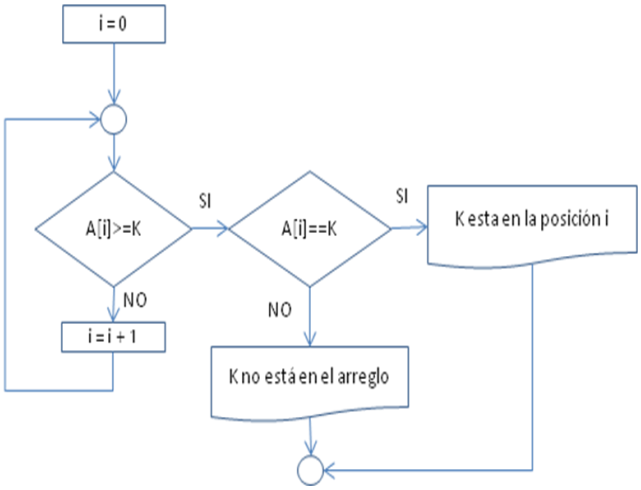



```
for(i=0;i<N;i++)
{
   if(array[i]==elemento)
   {
        printf("esta en %d", i);
        break;
    };
```

```
};
if (i==N)
        printf(" no esta en ");
```

Si al acabar el bucle, i vale N es que no se encontraba el elemento.

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




```
for(i=0;array[i]<=elemento;i++);
if (array[i]==elemento)
        printf("esta en %d", i)
else
        printf(" no esta en ");
```

Complejidad:

El elemento	Array ordenado	Array desordenado
Está	N/2 (promedio)	N/2 (promedio)
No está	N/2 (promedio)	N

El número medio de comparaciones que hay que hacer antes de encontrar el elemento buscado es de N/2. Por lo tanto la complejidad es lineal O(N).

Búsqueda binaria o dicotómica

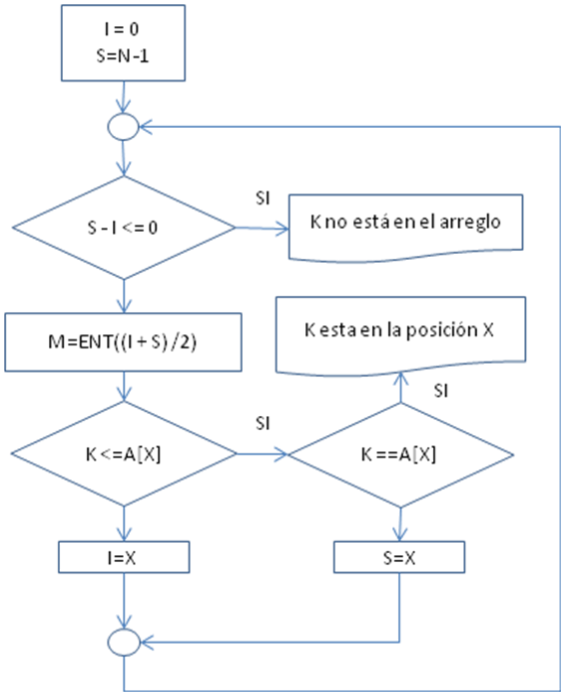
Para utilizar este algoritmo, el array debe estar ordenado. La búsqueda binaria consiste en dividir el array por su elemento medio en dos subarrays más pequeños, y comparar el elemento con el del centro. Si coinciden, la búsqueda se termina. Si el elemento es menor, debe estar (si está) en el primer 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 {}, entonces el elemento no se encuentra en el array.

A: lista (array), i: desde, s: hasta, M:medio



La implementación sería:

```
int desde, hasta, medio, elemento, posición, encontrado;
int lista[N];
conts false = 0;
const true = 1;

encontrado=false;
desde=1;
hasta=N;

do
{
  if (desde==hasta)              //si el array sólo tiene un elemento
   {
    if (lista[desde]==elemento)  //si es la solución
        posicion=desde           //se le da el valor
    else
        posicion:=-1;            //si no, es que no está en el array
   encontrado=true;
   };
```

```
  medio = (desde+hasta) / 2;      //divide el array en dos

  if (lista[medio]==elemento)     //si coincide con el central
   {
    posicion=medio;               //es la solución
    encontrado=true;
   }
  else
   if (lista[medio]<elemento)     //si es mayor
    desde=medio+1                 //se coge la mitad de la derecha
   else                           //si es menor
    hasta=medio-1;                //la mitad de la izquierda
} while (!encontrado);
```

Analicemos la complejidad:

- Para N=2 se harán 1 comparaciones,
- Para N=4 se harán 2 comparaciones,
- Para N=8 se harán 3 comparaciones,
- . . .
- Para N= 2^r se harán r comparaciones, de donde: r = $\log_2 N$

La complejidad es entonces logarítmica: O ($\log_2 N$). Comparándola con la busque secuencial ordenada, la búsqueda binaria es mucho más eficiente.