TEMA 1. INTRODUCCIÓN A LA ALGORITMIA

1. Definición de algoritmo

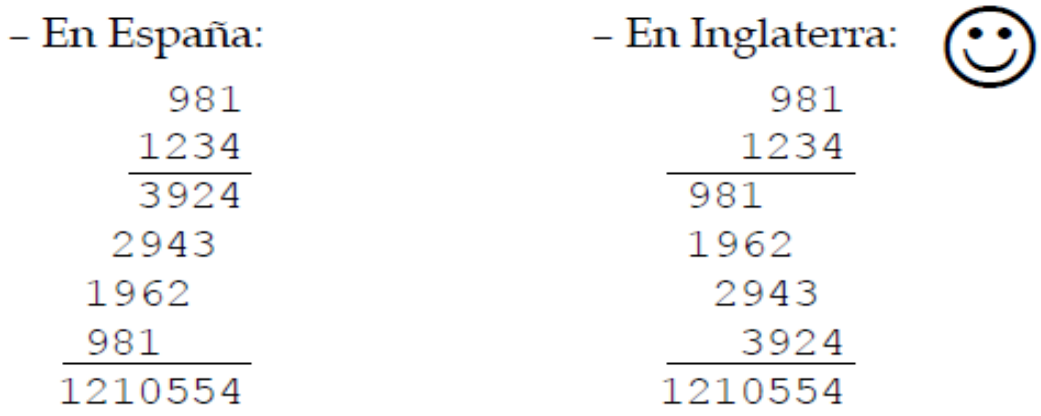
Un algoritmo es una descripción abstracta y ordenada de todas las acciones que se deben realizar, así como la descripción de los datos que deben ser manipulados por dichas acciones, para llegar a la solución de un problema. Debe:

- Ser independiente tanto del lenguaje de programación en que se exprese, como del ordenador en que se vaya a ejecutar.
- Ser claro y sencillo.
- Indicar el orden de realización de cada paso.
- Tener un número finito de pasos (así como un principio y un fin).
- Ser flexible (para facilitar su mantenimiento).
- Estar definido (de manera que si se sigue un algoritmo N veces con los mismos datos de entrada, se debe obtener el mismo resultado N veces).

Carácterísticas:

- Los algoritmos resuelven problemas.
- Un problema suele tener muchos ejemplares (a veces infinitos).
- Los algoritmos deben funcionar correctamente en todos los casos del problema que afirman resolver.
- Un solo caso erróneo, hace que el algoritmo sea incorrecto.
- Normalmente encontramos más de una forma de resolver un problema, es decir, hay varios algoritmos que resuelven el mismo problema.

Ejemplo: multiplicación de dos números



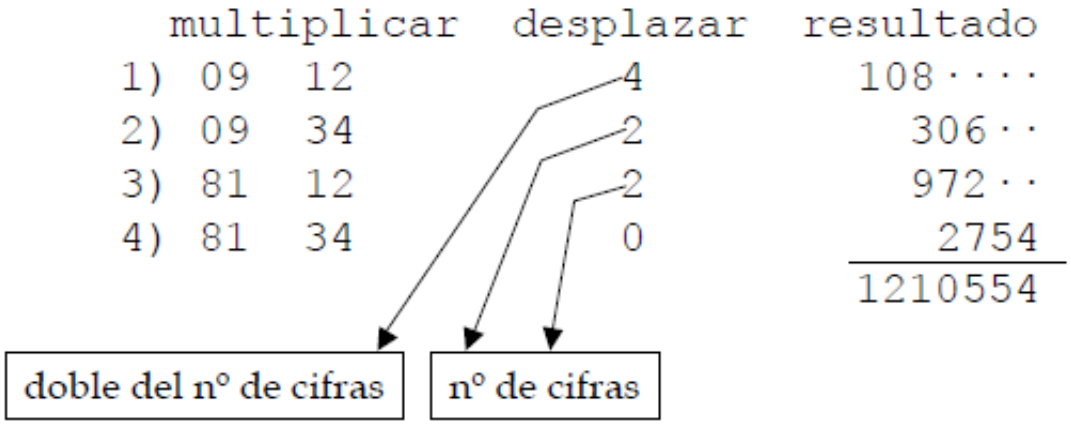
Ambos métodos son muy similares y los podemos denominar como algoritmo “clásico” de la multiplicación.

Multiplicación “a la rusa”:

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

Tiene las ventajas de que no hay que almacenar los productos parciales y que solo hay que sumar y dividir por dos.

Otro algoritmo más: en este caso ambos números deben tener el mismo número de cifras y este debe ser potencia de 2. El primer paso es dividir ambos números por la mitad y realizar cuatro productos:



Es decir, se reduce un producto de números de cuatro cifras en cuatro productos de números de dos cifras, varios desplazamientos y una suma. Los productos de números de dos cifras pueden hacerse de la misma manera:

	multiplicar		desplazar	resultado
1)	0	1	2	0··
2)	0	2	1	0·
3)	9	1	1	9·
4)	9	2	0	18
				108

Es un ejemplo de la técnica “divide y vencerás”.

Tal y como se han presentado los diferentes algoritmos de multiplicación no se mejora en eficiencia al algoritmo clásico. Pero, puede mejorarse: es posible reducir un producto de dos números de muchas cifras a 3 (en vez de 4) productos de números de la mitad de cifras, y éste sí que mejora al algoritmo clásico. Y aún se conocen métodos más rápidos para multiplicar números muy grandes. Necesitamos alguna forma de medir la eficiencia de un algoritmo para poder encontrar “el mejor” algoritmo. La tarea de encontrar “el mejor” algoritmo es la base de lo que se conoce como algoritmia cuyos objetivos principales son:

- Realizar un tratamiento sistemático de técnicas fundamentales para el diseño y análisis de algoritmos eficientes.
- Estudiar las propiedades de los algoritmos y elegir la solución más adecuada en cada situación. Esto ahorra tiempo y dinero. En muchos casos, una buena elección marca la diferencia entre poder resolver un problema y no poder hacerlo.

2. Eficiencia de los algoritmos

La unidad para medir la eficiencia de los algoritmos la vamos a encontrar a partir del Principio de Invarianza, que dice que “dos implementaciones diferentes de un mismo algoritmo no diferirán en eficiencia más que, a lo sumo, en una constante multiplicativa”. Esto quiere decir que si dos implementaciones consumen t1(n) y t2(n) unidades de tiempo, respectivamente, en resolver un caso de tamaño n, entonces siempre existe una constante positiva c tal que t1(n)<=ct2(n), siempre que n sea

suficientemente grande. El Principio de Invarianza es válido, independientemente del ordenador usado, del lenguaje de programación empleado y de la habilidad del programador (supuesto que no modifica el algoritmo).

El hecho de que un tiempo de ejecución dependa del input nos indica que ese tiempo debe definirse en función de dicho input (o del tamaño del input). Por tanto, T(n) es el tiempo de ejecución de un programa para un input de tamaño n, y también para el del algoritmo en el que se basa.

Diremos que un algoritmo consume un tiempo de orden t(n), para una función dada t, si existe una constante positiva c y una implementación del algoritmo capaz de resolver cualquier caso del problema en un tiempo acotado superiormente por ct(n) unidades de tiempo, donde n es el tamaño (o el valor, para problemas numéricos) del caso considerado.

Notación asintótica

Un algoritmo con un tiempo de ejecución T(n) se dice que es de orden O(f(n)) si existe una constante positiva c y un número entero n_0 tales que para todo $n>=n_0$ entonces $T(n)<=cf(n)$.

Por ejemplo, si T(0)=1, T(1)=4 y $T(n)=(n+1)^2$. Entonces T(n) es $O(n^2)$, puesto que si tomamos $n_0=1$ y c=4, se verifica que para todo n>=1 entonces $(n+1)^2<=4n^2$

Si un algoritmo tiene un tiempo de ejecución O(f(n)), a f(n) le llamaremos Tasa de Crecimiento. O(f(n)) (que se lee "el orden de f(n)") es el conjunto de todas las funciones t(n) acotadas superiormente por un múltiplo real positivo de f(n), dado que n es suficientemente grande (mayor que algún umbral n_0). Cuando T(n) es O(f(n)), estamos dando es una cota superior para el tiempo de ejecución, que siempre referiremos al peor caso.

Regla de la suma.

Supongamos, en primer lugar, que T1(n) y T2(n) son los tiempos de ejecución de dos segmentos de programa, P1 y P2, que T1(n) es O(f(n)) y T2(n) es O(g(n)). Entonces el tiempo de ejecución de P1 seguido de P2, es decir T1(n)+T2(n), es O(max(f(n), g(n))).

Por ejemplo, tenemos un algoritmo constituido por 3 etapas, en el que cada una de ellas puede ser un fragmento arbitrario de algoritmo con bucles y

ramas. Supongamos sus tiempos respectivos $O(n^2)$, $O(n^3)$ y O(nlogn). Entonces

- El tiempo de ejecución de las dos primeras etapas ejecutadas secuencialmente es $O(\max(n^2, n^3))$, es decir $O(n^3)$.

- El tiempo de ejecución de las tres juntas es $O(\max(n^2, n^3, \text{nlogn}))$, es decir $O(n^3)$.

Regla del producto.

Si T1(n) y T2(n) son los tiempos de ejecución de dos segmentos de programa, P1 y P2, T1(n) es O(f(n)) y T2(n) es O(g(n)), entonces T1(n)T2(n) es O(f(n)g(n)). De esta regla se deduce que O(cf(n)) es lo mismo que O(f(n)) si c es una constante positiva, así que por ejemplo $O(n^2/2)$ es lo mismo que $O(n^2)$.

3. Teoría del cálculo de la eficiencia

Operaciones elementales

La medición del tiempo de un algoritmo se realizará en función del número de operaciones elementales (OE) que este realiza. Una operación elemental es aquella cuyo consumo de tiempo de ejecución puede acotarse por una constante, es decir, que no depende del tamaño del ejemplar que se esté ejecutando. Para simplificar se considera el coste de una operación elemental, o de un conjunto de operaciones elementales como coste 1, es decir, OE es O(1).

Las siguientes operaciones si se aplican sobre datos de tipo básico son operaciones elementales:

- Operaciones aritméticas básicas: +,-,*,/
- Operaciones lógicas: AND, OR, NOT.
- Operaciones de orden: <,>,=
- Lectura o escritura
- Asignación de valores
- La instrucción Devolver

Instrucción condicional

Son instrucciones del tipo:

Si Cond entonces InstrV si no InstrF fsi

El coste de la instrucción completa es:

Coste(Cond)+Max{Coste(InstrV)+Coste(InstrF)}

Instrucción por bucle

Para los bucles no determinados como en la instrucción

Repetit InstrR hasta Cond fRepetir

El coste se obtiene mediante:

Coste(InstrR)+Coste(cond)+$\sum_{cond=Falso}(\text{Coste(InstrR)} + \text{Coste(Cond)})$

Para los bucles eterminados como en la instrucción

Desde Var=Vini hasta Vfin Hacer InstrD fDesde

El coste se obtiene mediante:

Max{Coste(Vini)+Coste(Vfin)}+$\sum_{Var=Vini}^{Vfin} \text{Coste(InstrD)}$

Eficiencia recursiva. Ecuación característica

Para encontrar la eficiencia de un método recursivo se usará el método de la ecuación característica, que consiste en encontrar las raíces de la ecuación matemática que representa el uso de recursos de un método recursivo.

Ejemplos

Determinar la Función de eficiencia y el Orden de complejidad de los siguientes códigos:

a)
```
if A[1,1] = 0 Then
        for i := 1 to n do
                for j := 1 to n do
                        A[i,j] := 0
else
        for i := 1 to n do
                A[i,i] := 1
```

b)
```
for i := 1 to n-1 do begin
        min := i
        for j := i+1 to n do
                if A[j] < A[min] Then
                        min := j
        temp := A[min]
        A[min] := A[i]
        A[i] := temp
end
```

c)
```
Procedure Insert (T[1..n])
for i := 2 to n do
        x:= T[i];
        j := i-1
        while j > 0 and x < T[j] do
                T[j+1] := T[j]
                j := j-1
        T[j+1] := x
End
```

d)

```
const dim = ...
tipos tipovector = array[1..dim] de entero
proc Ejercicio3(E n: entero; E/S v: tipovector)
var i: entero
    si (n>0) y (n<=dim) entonces
        Desde i ← 1 hasta (n-1) Hacer v[i] ← v[i+1] fdesde
        Ejercicio(n-1, v)
    fsi
fproc
```