Capítulo 5

Divide y Vencerás.

5.1. Competencias

Las competencias a desarrollar en el tema son:

- **CTEC3** Capacidad para evaluar la complejidad computacional de un problema, conocer estrategias algorítmicas que puedan conducir a su resolución y recomendar, desarrollar e implementar aquella que garantice el mejor rendimiento de acuerdo con los requisitos establecidos.

5.2. Introducción.

En este tema se va a estudiar una filosofía general de resolución de problemas más que un método concreto. Dicha filosofía está basada en la vieja consigna militar *si el enemigo es demasiado fuerte para tí, divídelo*. Es una técnica para diseñar algoritmos basada en la descomposición del problema a resolver en subproblemas que son a su vez subcasos de dicho problema. Resolviendo sucesiva e independientemente todos los subproblemas, y combinando las soluciones de los subproblemas se obtiene la solución del problema original. Si los subproblemas obtenidos no son lo suficientemente pequeños para resolverlos sin dividirlos, se vuelve a aplicar sobre dicho subproblema la idea de la división. Normalmente los subproblemas resultantes suelen ser del mismo tipo que el original, lo cual favorece un diseño recursivo para la solución del problema.

5.3. El método general.

Dado un problema para resolver para un conjunto de n datos de partida, la estrategia del divide y vencerás consiste en la división de los n datos de entrada en k subconjuntos distintos, donde $1 < k \leq n$, obteniéndose k subproblemas, del mismo carácter que el original. Estos subproblemas deben ser resueltos y combinando sus soluciones se obtiene la solución del problema original. Si los subproblemas resultantes son lo suficientemente amplios como para no poder resolverlos de forma directa, la estrategia se vuelve a aplicar de nuevo en esos subproblemas hasta que sean lo suficientemente pequeños como para resolverlos sin volver a subdividirlos.

El siguiente algoritmo recursivo resume esta idea. Supongamos que los n datos de entrada se almacenan en un vector A de n elementos. El algoritmo DyV será invocado inicialmente como $DyV(A, 1, n)$, donde los parámetros 1 y n definen el conjunto inicial de los datos de partida.

Algoritmo `DyV(A,p,q)`

```
inicio
  si Pequeño(A, p, q) = cierto entonces
    devolver Solucion(A, p, q)
  sino
    m ← dividir(A, p, q)
    devolver COMBINA(DYV(A,p,m),DYV(A,m+1,q))
  finsi
fin
```

$DyV(A, p, q)$ resuelve el problema para los elementos de A comprendidos entre p y q. *Pequeño(A, p, q)* evalúa si el problema aplicado al rango de datos comprendidios entre p y q es lo suficientemente pequeño como para ser resuelto de forma directa. Si esto es cierto, la solución a dicho subproblema es devuelta. En caso contrario, la llamada $dividir(A, p, q)$ vuelve a dividir el problema en 2 subproblemas, devolviendo el valor límite m que define los nuevos rangos de datos para los 2 subproblemas resultantes (A, p, m) y $(A, m + 1, q)$. Finalmente se invoca recursivamente al algoritmo DyV con los dos problemas resultantes y se combinan las soluciones de ambos usando la llamada $COMBINA(DYV(A, p, m), DYV(A, m + 1, q))$.

Es importante resaltar que el algoritmo *Solucion* es un algoritmo sencillo que resuelve los subproblemas correspondientes a los casos básicos. Este algoritmo es muy importante de cara a evaluar la eficiencia de un algoritmo que se diseñe mediante esta técnica. Para que esta técnica merezca la pena aplicarla es necesario que se cumplan tres condiciones:

- Se ha de seleccionar de una forma conveniente cuando usar el algoritmo básico en vez de seguir descomponiendo el problema.
- Ha de ser posible la descomposición del problema en subproblemas y recomponer las soluciones a dichos problemas de una manera eficiente.
- Los subproblemas han de ser aproximadamente del mismo tamaño.

5.4. Ejemplos.

A continuación se desarrollan una serie de ejemplos típicos basados en este método. Algunos de estos ejemplos ya han sido tratados en la asignatura de Metodología y Tecnología de la Programación, aunque no se han visto desde el punto de vista del método.

5.4.1. Búsqueda Binaria o Dicotómica.

Este algoritmo ya ha sido estudiado en las asignaturas previas de Programación. Se puede considerar la aplicación más simple del método Divide y Vencerás. Sea $v(n)$ un vector ordenado en forma creciente de n elementos de un tipo elemental cualquiera. El problema consiste en ver si un valor x del mismo tipo está presente o no en el vector. En el caso de que x esté presente, habrá que determinar un valor j tal que $v(j) = x$. Si x no está en el vector entonces $j = 0$. Si el valor x está repetido en el vector, j reflejará la posición que ocupa una de sus ocurrencias. La estrategia del método implica la división del entorno del problema inicial $E = (n, v_1, v_2, ..., v_n, x)$ en 3 subentornos, que originan 3 problemas del mismo caracter que el de partida. Para ello se selecciona un índice k, que suele ser la posición del elemento central del vector, de forma tal que ahora se tienen 3 subentornos: $E_1(k - 1, v_1, v_2, ..., v_{k-1}, x)$, $E_2(1, v_k, x)$ y $E_3(n - k, v_{k+1}, v_{k+2}, ..., v_n)$. La solución de 2 de los 3 subproblemas correspondientes a estos entornos es inmediata, y se obtiene comparando x con v_k. Tendríamos 3 posibilidades:

- Si $x = v_k$, entonces $j = k$ y E_1 y E_3 no necesitan ser resueltos.
- Si $x < v_k$ entonces $j = 0$ para los subproblemas correspondientes a los entornos E_2 y E_3, y seguiremos resolviendo el problema correspondiente a E_1.
- Si $x > v_k$ entonces $j = 0$ para los subproblemas correspondientes a los entornos E_1 y E_2, y seguiremos resolviendo el problema correspondiente a E_3.

```
Algoritmo BusquedaBinaria(v, n, x; ; )
  inicio
    izq ← 1
    der ← n
    mientras izq ≤ der hacer
      k ← (izq+der)/2
      si x < v(k) entonces
        der ← k − 1
      sino
        si x > v(k) entonces
          iz ← k + 1
        sino
          j ← k
          devolver j
        finsi
      finsi
    finmientras
    j ← 0
    devolver j
  fin
```

5.4.2. Máximo y mínimo elemento de un vector.

Otro problema simple que se puede resolver usando *Divide y vencerás* es el de encontrar el máximo y el mínimo elemento en un vector de n elementos. Un primer algoritmo de resolución de este problema podría ser el siguiente:

```
Algoritmo MaximoMinimo1(v, n; ; maximo, minimo)
  inicio
    maximo ← v(1)
    minimo ← v(1)
    para i de 2 a n hacer
      si v(i) > maximo entonces
        maximo ← v(i)
      finsi
      si v(i) < minimo entonces
        minimo ← v(i)
      finsi
```

finpara

fin

Analizando el algoritmo se puede ver que se necesitan siempre $2(n-1)$ comparaciones. Este número se puede reducir teniendo en cuenta que si $v(i) > maximo$, entonces no tiene sentido realizar la comparación del mínimo. Teniendo este hecho en cuenta, podríamos implementar una versión más eficiente, que redujese el número de comparaciones.

Algoritmo $MaximoMinimo2(v, n; ; maximo, minimo)$
inicio
$maximo \leftarrow v(1)$
$minimo \leftarrow v(1)$
para i **de** 2 **a** n **hacer**
si $v(i) > maximo$ **entonces**
$maximo \leftarrow v(i)$
sino
si $v(i) < minimo$ **entonces**
$minimo \leftarrow v(i)$
finsi
finsi
finpara
fin

Para esta versión, el mejor caso se da cuando el vector está en orden creciente, y el número de comparaciones sería $n-1$. El peor caso se da cuando el vector está ordenado decrecientemente y el número de comparaciones sería $2(n-1)$. El caso medio tendría $\frac{3n-3}{2}$ comparaciones.

Un algoritmo basado en la técnica divide y vencerás para resolver este problema, podría plantearse dividiento el entorno inicial del problema en dos subentornos que originan dos problemas similares al de partida, pero para vectores con la mitad de tamaño. El entorno del problema inicial sería $E = (n, v_1, v_2, ..., v_n)$ y se divide en dos subentornos $E_1(n/2, v_1, v_2, ..., v_{n/2})$ y $E_2(n - n/2, v_{n/2+1}, v_{n/2+2}, ..., v_n)$. Si $maximo(E)$ y $minimo(E)$ son el máximo y el mínimo para el entorno inicial E, entonces $maximo(E) = maximo(maximo(E_1), maximo(E_2))$ y $minimo(E) = minimo(minimo(E_1), minimo(E_2))$. Cuando el entorno inicial contenga un solo elemento, la solución se puede obtener sin dividir.

A continuación se detalla el algoritmo recursivo basado en esta idea. Este algoritmo buscará el máximo y el mínimo en el intervalo de indices que va de i a j en el vector v. Los casos en los que $i = j$ y $i = j - 1$ (subvectores con 1 o 2 elementos), se tratan de forma separada. Para subvectores con más de dos elementos, se hace un tratamiento similar al de la búsqueda binaria, determinando el elemento central para dividir en dos subproblemas de tamaños similares. Cuando se obtiene el máximo y el mínimo para esos dos subproblemas, los dos máximos y los dos mínimos son comparados para obtener la solución del problema de partida.

En la primera llamada $i = 1 \;\; j = n$
Algoritmo $MaximoMinimo(v, n, i, j; ; maximo, minimo)$
inicio
si $i = j$ **entonces**
$maximo \leftarrow v(i)$
$minimo \leftarrow v(i)$
sino

si $i = j - 1$ **entonces**
si $v(i) < v(j)$ **entonces**
$maximo \leftarrow v(j)$
$minimo \leftarrow v(i)$
sino
$maximo \leftarrow v(i)$
$minimo \leftarrow v(j)$
finsi
sino
$mitad \leftarrow \frac{i+j}{2}$
$MaximoMinimo(v, n, i, mitad; ; maximo1, minimo1)$
$MaximoMinimo(v, n, mitad + 1, j; ; maximo2, minimo2)$
$maximo = Maximo(maximo1, maximo2)$ $calcula\ \ el\ \ mayor\ \ de\ \ los\ \ maximos$
$minimo = Minimo(minimo1, minimo2)$ $calcula\ \ el\ \ menor\ \ de\ \ los\ \ minimos$
finsi
finsi
finsi
fin

Al ser un algoritmo recursivo, se pueden plantear las siguientes ecuaciones de recurrencia (Aquí solo se han tenido en cuenta las comparaciones y no las asignaciones, igual que se ha hecho en las dos versiones anteriores):

$$t(n) = \begin{cases} 1 & si\ n = 1 \\ 2 & si\ n = 2 \\ 2t(n/2) + 2 & si\ n > 2 \end{cases}$$

Suponiendo que n es una potencia de dos, es decir $n = 2^k$ y expandiendo las recurrencias, se tiene:

$$t(n) = 2t(\tfrac{n}{2}) + 2$$

$$= 2(2t(\tfrac{n}{4}) + 2) + 2$$

$$= 4t(\tfrac{n}{4}) + 4 + 2$$

... Expandiendo $k - 1$ veces

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

$$= 2^{k-1}t(2) + \sum_{i=2}^{k-1} 2^i$$

$$= 2^k + 2^k - 2 = 2n - 2$$

Como se puede apreciar este valor es el peor, promedio y mejor cuando n es una potencia de 2.

5.4.3. Algoritmo de ordenación de Hoare (Quicksort).

Este algoritmo, ya estudiado en cursos anteriores, también se basa en esta técnica. En primer lugar se selecciona un elemento pivote del vector, de forma aleatoria, permutándose los elementos del vector de forma que los que son mayores que el pivote quedan a la derecha del mismo y los menores quedan a la izquierda. De esta forma, el pivote queda en su posición final, y la ordenación de los subvectores que quedan a la derecha y a la izquierda del mismo se pueden realizar independientemente y son dos subproblemas del mismo carácter que el original pero para un ámbito más reducido. A este proceso se le denomina *partición*. Aplicando la idea de la partición sucesivas veces, con los subvectores resultantes de las particiones anteriores, iremos obteniendo subproblemas similares al original, pero con vectores cada vez más pequeños. Los subproblemas resultantes tendrán solución inmediata cuando los subvectores obtenidos tengan un solo elemento. A continuación se detalla dicho algoritmo:

$en\ \ la\ \ primera\ \ llamada\ \ iz = 1; de = n$
Algoritmo $QuickSort(iz, de, n; v;)$
inicio
$i \leftarrow iz$
$j \leftarrow de$
$x \leftarrow v(E(\frac{iz+de}{2}))$
repetir
mientras $v(i) < x$ **hacer**
$i \leftarrow i + 1$
finmientras
mientras $v(j) > x$ **hacer**
$j \leftarrow j - 1$
finmientras
si $i \leq j$ **entonces**
$aux \leftarrow v(i)$
$v(i) \leftarrow v(j)$
$v(j) \leftarrow aux$
$i \leftarrow i + 1$
$j \leftarrow j - 1$
finsi
hasta que $i > j$
si $iz < j$ **entonces** $El\ \ izquierdo\ \ tiene\ \ mas\ \ de\ \ un\ \ elemento$
$QuickSort(iz, j, n, v)$
finsi
si $i < de$ **entonces** $El\ \ derecho\ \ tiene\ \ mas\ \ de\ \ un\ \ elemento$
$QuickSort(i, de, n, v)$
finsi
fin

5.4.4. Selección de los k menores elementos de un vector.

La idea de la *partición* expuesta en el ejemplo anterior se puede utilizar para obtener los k-menores elementos de un vector, sin necesidad de que se den ordenados. Un algoritmo inmediato para tal fin, se podría implementar obteniendo el mínimo elemento del vector k veces y tendría complejidad kn. Sin embargo, si usamos el algoritmo de la partición, y el elemento pivote alcanza la posición k-ésima después de aplicarlo, teniendo en cuenta que los que hay a su izquierda son menores o iguales que el pivote, dichos elementos junto con el pivote serán los k menores elementos del vector. Este procedimiento, por término medio, es mucho más eficiente que el anterior, aunque tiene el inconveniente de que los elementos no están ordenados. Evidentemente, es muy improbable que al aplicar el algoritmo de la partición, el elemento pivote ocupe la posición k. En ese caso, podríamos aplicar un procedimiento similar al de la búsqueda binaria, intentando buscar un elemento pivote que ocupe la posición k en una partición posterior. Para ello supongamos que en la primera partición el pivote ocupa una posición p distinta de k. Tendríamos dos posibilidades:

- Si $p > k$ entonces dentro del subvector que componen los p menores están los k menores, con lo cual volveríamos a aplicar el algoritmo de partición en el subvector de p elementos.
- Si $p < k$ entonces tendríamos una parte, compuesta por p elementos, de los k menores que estamos buscando, y volveríamos a aplicar el algoritmo de *partición* a los $n-p$ elementos que quedan a la derecha del pivote obtenido.

El algoritmo quedaria como sigue:

Algoritmo $KMenores(iz, de, n, k; v;)$
inicio
$i \leftarrow iz$
$j \leftarrow de$
$x \leftarrow v(E(\frac{iz+de}{2}))$
repetir
mientras $v(i) < x$ **hacer**
$i \leftarrow i + 1$
finmientras
mientras $v(j) > x$ **hacer**
$j \leftarrow j - 1$
finmientras
si $i \leq j$ **entonces**
$aux \leftarrow v(i)$
$v(i) \leftarrow v(j)$
$v(j) \leftarrow aux$
$i \leftarrow i + 1$
$j \leftarrow j - 1$
finsi
hasta que $i > j$
si $k = i - 1$ **entonces** $problema\ resuelto$
escribir $los\ k\ primeros$
sino
si $k < i - 1$ **entonces**

$KMenores(iz, j, n, k; v;)$
sino
$KMenores(i, de, n, k; v;)$
finsi
finsi
fin

5.4.5. La aritmética de los enteros grandes.

En las operaciones de suma, resta, multiplicación y división, se puede considerar que el tiempo que se invierte en ellas está acotado superiormente en función del ordenador utilizado. Evidentemente, esta suposición es razonable siempre que no se sobrepase el tamaño máximo que el hardware utilice para almacenar un valor numérico. En caso de que se precise trabajar con enteros muy grandes y con precisión, de forma tal que no puedan ser almacenados en coma flotante, es necesario implementar su almacenamiento y manipulación mediante software. Como dato relevante indicar que en 1985, para calcular las cifras decimales de Π y sus propiedades estadísticas, se manipularon en Japón operandos con 10 millones de cifras. En 1986, en EEUU, el cálculo se hizo con operandos de 30 millones de cifras, necesitando unas treinta horas de cálculo un ordenador $CRAY - 2$. En 2004, mediante un ordenador $Hitachi$ se obtuvieron 1 billón, $400{,}000$ millones de decimales.

El algoritmo que se va a desarrollar como ejemplo no es eficiente para estos tamaños, pero si es un ejemplo ilustrativo de la técnica *Divide y Vencerás*. Partimos de la base de que ya está resuelto el problema del almacenamiento de un entero grande, y el de calcular la suma de los mismos. A continuación se va a detallar un algoritmo para obtener el producto de dos enteros grandes, en un tiempo razonable.

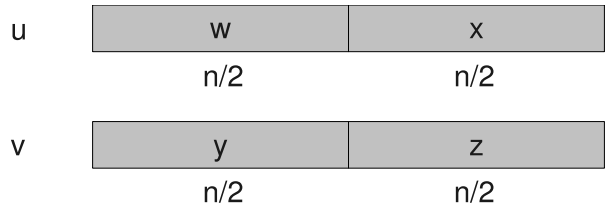

Figura 5.1: Descomposición de u y v.

Sean u y v dos enteros de n cifras que se desean multiplicar, y son lo suficientemente grandes como para no poder hacerlo directamente. Estos dos números se podrían descomponer de la siguiente forma:

$$u = 10^s w + x$$

$$v = 10^s y + z$$

donde $0 \leq x < 10^s$, $0 \leq z < 10^s$ y $s = E(n/2)$.

De esta forma los enteros w e y tienen $n - E(n/2)$ cifras. Diremos que un entero tiene j cifras si es menor que 10^j, aunque no sea mayor que 10^{j-1}.

El producto en cuestión sería:

$$uv = 10^{2s} wy + 10^s (wz + xy) + xz$$

El algoritmo resultante sería:

Algoritmo $MultiplicarEG1(u, v)$
inicio
$n \leftarrow MayorMagnitud(u, v)$
si $pequeño(n) = cierto$ **entonces**
devolver $u * v$ *producto de u y v por el algoritmo clasico*
sino
$s \leftarrow \frac{n}{2}$
$w \leftarrow \frac{u}{10^s}$
$x \leftarrow u \ \ mod \ \ 10^s$
$y \leftarrow \frac{v}{10^s}$
$z \leftarrow v \ \ mod \ \ 10^s$
devolver $MultiplicarEG1(w, y) * 10^{2s} + (MultiplicarEG1(w, z) + MultiplicarEG1(x, y)) * 10^s + MultiplicarEG1(x, z)$
finsi
fin

Se puede demostrar que el tiempo de ejecución de este algoritmo es cuadrático. No obstante, dicho algoritmo puede ser mejorado, reduciendo los cuatro productos a tres, a base de realizar más sumas ya que éstas son más rápidas que las multiplicaciones cuando los operandos son grandes. Para ello se usa la siguiente idea:

$$r = (w + x)(y + z) = wy + (wz + xy) + xz$$

Sean p y q:

$$p = wy$$
$$q = xz$$

De esta forma el algoritmo quedaría:

Algoritmo $MultiplicarEG2(u, v)$
inicio
$n \leftarrow MayorMagnitud(u, v)$
si $pequeño(n) = cierto$ **entonces**
devolver $u * v$ *producto de u y v por el algoritmo clasico*
sino
$s \leftarrow \frac{n}{2}$
$w \leftarrow \frac{u}{10^s}$
$x \leftarrow u \ \ mod \ \ 10^s$
$y \leftarrow \frac{v}{10^s}$
$z \leftarrow v \ \ mod \ \ 10^s$
$r \leftarrow MultiplicarEG2(w + x, y + z)$
$p \leftarrow MultiplicarEG2(w, y)$
$q \leftarrow MultiplicarEG2(x, z)$
devolver $10^{2s} * p + 10^s * (r - p - q) + q$
finsi
fin

De esta forma se mejora la eficiencia del algoritmo anterior, y su tiempo de ejecución es del orden de n^{log3}.

5.4.6. Ordenación por fusión.

En este caso lo que se hace es descomponer el vector en dos partes cuyos tamaños sean similares, ordenar las dos partes mediante llamadas recursivas y finalmente fusionar las soluciones de cada parte de forma tal que se mantenga el orden. Para poder realizar estos pasos se necesita un algoritmo eficiente que fusione dos vectores ordenados en un único vector, cuya longitud sea la suma de los vectores fusionados. La forma más eficiente de hacerlo es ubicando un espacio adicional en los vectores a fusionar, para usarlo como centinela.

Algoritmo $fusionar(u(m+1), v(n+1), t(m+n))$
  inicio
    $i \leftarrow 1$
    $j \leftarrow 1$
    $u(m+1) \leftarrow \infty$
    $v(n+1) \leftarrow \infty$
    para k **de** 1 **a** $m+n$ **hacer**
      si $u(i) < v(j)$ **entonces**
        $t(k) \leftarrow u(i)$
        $i \leftarrow i+1$
      sino
        $t(k) \leftarrow v(j)$
        $j \leftarrow j+1$
      finsi
    finpara
  fin

El algoritmo *ordenacion* que se refleja en el algoritmo *OrdenarFusion*, ordenará el vector por un método simple cuando el vector sea lo suficientemente pequeño como para ordenarlo por fusión. En caso contrario se ejecutará la ordenación por fusión. Los vectores intermedios se crearán con la mitad de los elementos del vector original, y se les añadirá un elemento más para el centinela.

Algoritmo $OrdenarFusion(t(n))$
  inicio
    si $pequeño(n) = cierto$ **entonces** `/*Ordena usando un metodo no sofisticado */`
      $ordenar(t)$
    sino
      $Crear\ vector\ u\ con\ \frac{n}{2}+1\ elementos$
      $Crear\ vector\ v\ con\ n-\frac{n}{2}+1\ elementos$
      para i **de** 1 **a** $\frac{n}{2}$ **hacer**
        $u(i) \leftarrow t(i)$
      finpara
      para i **de** $1+\frac{n}{2}$ **a** n **hacer**
        $v(i-\frac{n}{2}) \leftarrow t(i)$
      finpara
      $OrdenarFusion(u)$
      $OrdenarFusion(v)$
      $fusionar(u, v, t)$

finsi

fin

Este algoritmo es un ejemplo claro del divide y vencerás. Cuando el vector es lo suficientemente pequeño lo ordena por un método sencillo y en caso contrario lo divide en dos partes, similares en tamaño, y los resuelve recursivamente, combinando ambas soluciones para obtener la solución del caso original.

El tiempo de la fusión es de $O(n)$ y al ser un algoritmo recursivo, se pueden plantear las siguientes ecuaciones de recurrencia:

$$t(n) = \begin{cases} a & si\ n = 1\ a\ es\ una\ constante \\ 2t(n/2) + cn & si\ n > 1\ c\ es\ una\ constante \end{cases}$$

Suponiendo que n es una potencia de dos, es decir $n = 2^k$ y expandiendo las recurrencias, se tiene:

$$\begin{aligned} t(n) &= 2(2t(\tfrac{n}{4}) + \tfrac{cn}{2}) + cn \\ &= 4t(\tfrac{n}{4}) + 2cn \\ &= 4(2t(\tfrac{n}{8}) + \tfrac{cn}{4}) + 2cn \\ &... \\ &= 2^k t(1) + kcn = an + cnlogn \end{aligned}$$

como $2^k < n \leq 2^{k+1}$ entonces $t(n) \leq t(2^{k+1})$, por tanto el orden de complejidad es $O(nlogn)$.

Como se ha demostrado, la ordenación por fusión es de $O(nlogn)$, aunque hay que resaltar que este orden se consigue cuando el vector se divide en dos partes de tamaño similar. Si por ejemplo, en la división del vector se obtuviesen partes de tamaño muy distinto, por ejemplo una de tamaño 1 y la otra de tamaño $n-1$, el orden obtenido sería $O(n^2)$, ya que se necesitarían $n-1$ llamadas recursivas en vez de $log_2 n$ llamadas recursivas.

5.4.7. Exponenciación.

Sean a y n dos enteros y se desea calcular a^n. En caso de que n sea pequeño el algoritmo más simple sería el siguiente:

Algoritmo $Exponenciacion(a, n; ; resultado)$

inicio

$resultado \leftarrow a$

para i **de** 1 **a** $n-1$ **hacer**

$resultado \leftarrow a * resultado$

finpara

fin

Este algoritmo es $O(n)$ siempre y cuando las multiplicaciones se consideren como operaciones elementales. Sin embargo valores relativamente pequeños de a y n pueden hacer que los enteros se desborden. Por ejemplo 16^{18} no cabe en un entero de 64 bits. Para trabajar con operandos más grandes es preciso tener en cuenta el tiempo necesario para cada multiplicación.

Un aspecto clave para mejorar el algoritmo anterior se basa en que $a^n = (a^{n/2})^2$ cuando n es par. Por otra parte, si n es impar, podemos reducir al caso anterior teniendo en cuenta que $a^n =$

$a * a^{n-1} = a * (a^{(n-1)/2})^2$. De esta forma se podría plantear el siguiente algoritmo recursivo basado en Divide y Vencerás:

Algoritmo $ExponenciacionDyV(a, n; ;)$
inicio
si $n = 1$ **entonces**
devolver a
sino
si $n \ mod \ 2 = 0$ **entonces** `n es par`
$aux \leftarrow ExponenciacionDyV(a, n/2)$
devolver $aux * aux$
sino
devolver $a * ExponenciacionDyV(a, n-1)$
finsi
finsi
fin

El tiempo de ejecución del algoritmo sería $O(\log n)$

Se puede plantear una versión iterativa de este algoritmo, descomponiendo a^n en productos de potencias de a cuyos exponentes sean potencias de 2. Por ejemplo $a^{61} = a^{32} * a^{16} * a^8 * a^4 * a^1$. De esta forma tendríamos el siguiente algoritmo iterativo:

Algoritmo $ExponenciacionDyVIterativo(a, n; ; r)$
inicio
$i \leftarrow n$
$x \leftarrow a$
$r \leftarrow 1$
mientras $i > 0$ **hacer**
si $i \ mod \ 2 = 1$ **entonces** `i es impar`
$r \leftarrow rx$
finsi
$x \leftarrow x^2$
$i \leftarrow \frac{i}{2}$
finmientras
fin

En este algoritmo se calcularía innecesariamente x^2 en la última iteración.

Si por el tamaño de los operandos el producto no es de tiempo constante y depende del tamaño m de los operandos, se puede demostrar que los tiempos de ejecución de los algoritmos de $Exponenciacion$ y $ExponenciacionDyV$ si se usa la multiplicación clásica, son ambos de $O(m^2n^2)$, y en el caso de que se use la multiplicación basada en Divide y Vencerás son de $O(m^{\log 3}n^2)$ y de $O(m^{\log_3}n^{\log_3})$ respectivamente. Esto implica que la reducción drástica que se produce en el número de multiplicaciones al calcular a^n no se traduzca en una reducción igual de drástica en el tiempo de ejecución del algoritmo.