

## Divide y Vencerás (DyV): Concepto

```
-- Técnica básica de diseño de algoritmos que consiste en:  
-- (1) División del problema a resolver en subproblemas de la misma  
-- naturalidad  
-- (2) Resolución (independiente) de los subproblemas  
-- (3) Combinación las "subsoluciones" para formar  
-- la solución del problema original  
  
-- Asimismo, los tres pasos anteriores se aplican recursivamente  
-- a los subproblemas generados, hasta llegar a un tamaño básico  
-- que se resuelve directamente.
```

```
-- Características de la técnica DyV:
```

```
-- *Simple estructuralmente  
-- *Eficiente computacionalmente (en general)  
-- *Aplicable en diversos problemas  
-- *Adecuada a soluciones paralelas
```

```
-- Cuestión de representar lo que es un problema
```

```
-- Máximo de una lista
```

```
-- -- (Nota: la siguiente versión utiliza una estrategia  
-- -- de Divide y Vencerás pero no utiliza el esquema genérico  
-- -- que emplearemos...)
```

```
-- primeros:: [a] -> Int -> [a]  
-- primeros l 0 = []  
-- primeros (x:xs) n = x:(primeros xs (n-1))
```

```
-- -- (Nota: Hugs proporciona las funciones take y drop.)  
-- quitar:: [a] -> Int -> [a]  
-- quitar l 0 = l  
-- quitar (x:xs) n = quitar xs (n-1)
```

```
-- partir:: [a] -> ([a], [a])  
-- partir [] = ([], [])  
-- partir ([x]) = ([x], [])  
-- partir l = (primeros l mitad, quitar l mitad) where mitad =  
-- (longitud `div` 2) where longitud = length (l)
```

```
-- -- PRE: lista de entrada no vacía  
-- maximo:: Ord a => [a] -> a  
-- maximo ([ele]) = ele  
-- maximo (x:xs) = let (lizq, lder) = partir (x:xs) in  
--                 max (maximo lizq) (maximo lder)
```

```
-- --Nota: sólo maximo necesita el contexto Ord, por la función  
-- --predefinida max
```

```
-- (Ver luego la ecuación de recurrencia...)
```

```
-- Nota: la fase de partición es ineficiente. Se pueden realizar
-- otros tipos de partición más eficientes, aunque en general
-- en el paradigma funcional en general no podremos evitar la
-- complejidad lineal para la fase de división de un problema.
-- (En el paradigma imperativo, sí podemos dividir con
-- complejidad constante un vector mediante subíndices...)
```

```
-- (Nota: funciones take y drop:
-- Prelude> take 2 [1,2,3,4,5,6]
-- [1,2] :: [Integer]

-- Prelude> drop 2 [1,2,3,4,5,6]
-- [3,4,5,6] :: [Integer]
-- )
```

```
-- -----
```

```
-- -- JC 3-3-2001 versión currificada en Haskell
```

```
-- module Divyvenc (dyv) where
```

```
-- import Holo
```

```
-- -- versión currificada
```

```
-- dyv::
--          (tproblema -> Bool)          -- Es Trivial ?
--          -> (tproblema -> tsolucion)   -- Resolver Caso
Trivial
--          -> (tproblema -> [tproblema]) -- Dividir
--          -> ([tsolucion] -> tsolucion)  -- Combinar
--          -> tproblema                  -- problema
--          -> tsolucion
```

```
-- dyv estrivial resolver dividir combinar problema =
--   if (estrivial problema) then
--     resolver (problema)
--   else
--     combinar (map
--               (dyv estrivial resolver dividir combinar)
--               (dividir problema))
```

```
-- ----
```

```
-- El máximo de una lista empleando el esquema con la
-- función dyv sería:
```

```
-- module ZMax (maximoDyV) where
```

```
-- import Divyvenc
```

```
-- type TProblema a = [a] -- lista de entrada
-- type TSolucion a = a   -- maximo
```

```

-- -- PRE: lista de entrada no vacía
-- estrivial:: (TProblema a) -> Bool
-- estrivial ([_]) = True
-- estrivial (x1:x2:xs) = False

-- resolver:: (TProblema a) -> (TSolucion a)
-- resolver ([x]) = x

-- dividir:: (TProblema a) -> [TProblema a]
-- dividir xs =
--   let mitad = length xs `div` 2 in [take mitad xs, drop mitad xs]

-- combinar:: Ord a => [TSolucion a] -> (TSolucion a)
-- combinar ([x, y]) = if (x > y) then x else y

-- maximoDyV:: Ord a => (TProblema a) -> (TSolucion a)
-- maximoDyV = dyv estrivial resolver dividir combinar

-- -- maximoDyV [2,4,7,1,2,3,4,6]
-- -- 7 :: Int

-- ---

-- Algunas ideas básicas sobre complejidad:

-- Complejidad: medida de los recursos necesarios para
-- ejecutar un programa

-- - Compl. en tiempo
-- - Compl. en memoria

-- Nos centraremos en la complejidad en tiempo
-- (la complejidad en espacio es quizá menos importantes
-- con los modernos sistemas operativos de memoria virtual,
-- y en programación funcional es más difícil de estimar).

-- Se utilizará la complejidad asintótica, en la que
-- se desprecian constantes multiplicativas, etc.,
-- importando sólo el término dominante de mayor orden.
-- Este tipo de complejidad nos permite independizarnos de
-- factores como el tipo de máquina, o de compilador, etc.

-- En general, se considera la complejidad del caso peor.

-- La referencia de la complejidad es el tamaño de la entrada.

-- longitud:: [a] -> Int
-- longitud [] = 0
-- longitud (x:xs) = 1 + longitud (xs)

-- Longitud:
--  $T(n) = k_1, n = 0$ 
--  $T(n-1) + k_2, n > 0$ 

```

```

-- Desarrollando T(n):
--  $T(n) = T(n-1) + k_2 = (T(n-2) + k_2) + k_2 =$ 
--  $((T(n-3) + k_2) + k_2) + k_2 = \dots =$ 
--  $n \cdot k_2 + k_1$ 
--  $\Rightarrow O(n)$ , donde n es la longitud de la lista

-- factorial:: Int -> Int
-- factorial 0 = 1
-- factorial n = n * (factorial (n-1))

-- Factorial:
-- T(N) = k1, N = 0
--       T(N-1) + k2, N > 0

--  $\Rightarrow O(N)$ , donde N es el valor (absoluto) del número

-- Nota: el último caso no tiene una complejidad lineal "real",
-- ya que N no es el tamaño de la entrada.

-- Complejidad de concatenacion:

-- concatenacion:: [a] -> [a] -> [a]
-- concatenacion [] l2 = l2
-- concatenacion (x:xs) l2 = x:(concatenacion xs l2)

-- concatenacion [2,3] [4,8]
-- [2,3,4,8] :: [Integer]

-- Es lineal en el tamaño de la primera lista.

-- Entonces, la complejidad de la siguiente versión de darVuelta
-- será:

-- darVuelta1:: [a] -> [a]
-- darVuelta1 [] = []
-- darVuelta1 (x:xs) = (darVuelta1 (xs)) ++ [x]

-- T(n) = k1, n = 0
--       T(n-1) + k2*(n-1) + k3, n > 0

-- T(n) = k3 + k2*(n-1) + T(n-1) = k3 + k2*(n-1) + (k3 + k2*(n-2) + T
-- (n-2)) =
-- k3*(n-1) + k2*Sum_{i=1}^{n-1} i =
-- k3*(n-1) + k2*(n-1)*n/2 =
-- k4*n2 (complejidad cuadrática)

-- --

-- Nota: ++

-- Prelude> [1,2]++[3]
-- [1,2,3] :: [Integer]

-- Nota: concat
-- Prelude> concat

```

```

-- foldr (++) [] :: [[a]] -> [a]

-- Prelude> concat [[1,3],[5]]
-- [1,3,5] :: [Integer]

-- ---

-- Teorema fundamental sobre la complejidad algorítmica
-- de las soluciones pertenecientes a la técnica de
-- Divide y Vencerás:

-- Si  $T(n) = k_1$ ,  $n \leq n_0$ 
--     $= a \cdot T(n/b) + f(n)$ ,  $n > n_0$ 

-- y  $f(n)$  es  $k_2 \cdot n^k$ 

-- donde:
--     a: número de llamadas recursivas con subproblemas.
--     n/b: tamaño de los subproblemas.
--     f(n): coste en tiempo de fases de división y combinación.

-- Entonces

--  $T(n) = 0(n^k)$ , si  $a < b^k$ 
--  $T(n) = 0(n^k \cdot \log n)$ , si  $a = b^k$ 
--  $T(n) = 0(n^{\log_b(a)})$ , si  $a > b^k$ 

-- ---

-- En el caso de la función del cálculo del máximo anterior:

--  $T(n) = k_1$ ,  $n = 1$ 
--     $k_2 \cdot n + 2 \cdot T(n/2) + k_3$ ,  $n > 1$ 

-- En este caso:  $a = 2$ ;  $b^k = 2^1 = 2$ 
-- Estamos en el caso de  $a = b^k$ , por lo que, según el teorema,
--  $T(n) = 0(n^k \cdot \log n) = 0(n \cdot \log n)$ 

-- Se podría haber resuelto el problema con complejidad
-- lineal sin utilizar la técnica de divide y vencerás.
-- Vemos que en este caso que la complejidad con
-- divide y vencerás es mayor que lineal, por lo que
-- no es interesante.

-- Nota: en el paradigma imperativo se podría haber dividido
-- el problema con un coste constante delimitando el problema
-- con subíndices del array. Esto hubiera convertido la
-- ecuación de recurrencia en

-- (En imperativo...)
--  $T(n) = k_1$ ,  $n = 1$ 
--     $k_2 + 2 \cdot T(n/2) + k_3$ ,  $n > 1$ 

-- Entonces:  $a = 2$ ;  $b^k = 2^0 = 1$ 
-- Estamos en el caso de  $a > b^k$ , por lo que, según el teorema,
--  $T(n) = 0(n^{\log_b(a)}) = 0(n^{\log_2(2)}) = 0(n)$ 
-- La complejidad asintótica en este caso es también lineal

```

```

-- como en la solución "directa", aunque la complejidad
-- "práctica" de divide y vencerás es mayor por las
-- mayores constantes multiplicativas asociadas.

-- -----

-- Nota: diferencias entre el paradigma funcional y el imperativo
-- en cuanto a la eficiencia de la división de una lista
-- en dos sublistas... (la etapa de división de un problema
-- en subproblemas...)

-- ---

-- Ordenación por mezcla:

-- La ordenación por mezcla ("mergesort") es una importante
-- técnica de ordenación que emplea una estrategia de
-- Divide y Vencerás y que presenta una complejidad adecuada
-- (mejor que la complejidad cuadrática de los métodos básicos
-- de ordenación).
-- La etapa de división es simple (trivial), dividiendo
-- el problema en subproblemas (siempre que no tenga
-- un tamaño básico).
-- Es en la etapa de combinación donde recae el mayor "peso"
-- o esfuerzo del método. Esta etapa combina o une dos listas
-- ordenadas para formar una lista unión ordenada a su vez.
-- Una de las claves del método es que la combinación (mezcla)
-- de dos listas ordenadas se puede resolver de forma relativamente
-- sencilla

-- --module Mergesort (mergesort, Problema, Solucion) where

-- -- mergesort usando el esquema de divide y venceras
-- -- Para dividir una lista ponemos los elementos que esta
-- -- en la primera mitad de la lista en una lista
-- -- y los demas en otra. Para ello necesitamos calcular
-- -- la longitud de la lista antes y dividirla por dos
-- -- Ej dividir([3,45,9,8]) = [[3,45],[9,8]]

-- --JC 3-3-2001 versión en Haskell

-- import Divyvinc

-- type Problema = [Int]
-- type Solucion = [Int]

```

```

-- estrivial:: Problema -> Bool
-- estrivial ([]) = True
-- estrivial (primero:resto) = resto == []

-- resolver :: Problema -> Solucion
-- resolver (lista) = lista

-- --PRE: n <= Longitud(lista)
-- partir :: [a] -> Int -> [[a]]
-- partir [] _ = [[]], [[]]
-- partir (primero:resto) n =
--     if n > 0 then
--         let [lista1, lista2] = partir resto (n-1) in
--             [primero:lista1, lista2]
--     else [[], primero:resto]

-- dividir :: Problema -> [Problema]
-- dividir (lista) = partir lista ((length (lista)) `div` 2)

-- --PRE: Longitud(lista) = 2
-- combinar :: [Solucion] -> Solucion
-- combinar [], lista2 = lista2
-- combinar [lista1, []] = lista1
-- combinar [primero1:resto1, primero2:resto2] =
--     if primero1 <= primero2 then
--         primero1:(combinar [resto1, primero2:resto2])
--     else
--         primero2:(combinar [primero1:resto1, resto2])

-- mergesort :: Problema -> Solucion
-- mergesort = dyv estrivial resolver dividir combinar

-- --Ejemplo utilización:
-- --mergesort [3,6,1,3]

-- --

-- T(n) = k1, n = 1
--       2*T(n/2) + k2*n, n > 1

-- a=2=b^k=2^1=2
-- => O(n^1 * log n) = O(n log n)

-- --

-- --module Mergesort2 (mergesort, Problema, Solucion) where
-- -- mergesort usando el esquema de divide y venceras
-- -- la lista se divide separando los elementos de posición
-- -- par de los de posición impar en la lista original.

-- --JC 3-3-2001 versión en Haskell

-- import Divyvenc

```

```

-- type Problema = [Int]
-- type Solucion = [Int]

-- estrivial:: Problema -> Bool
-- estrivial ([]) = True
-- estrivial (primero:resto) = resto == []

-- resolver :: Problema -> Solucion
-- resolver (lista) = lista

-- dividir :: Problema -> [Problema]
-- dividir ([]) = [[]]
-- dividir ([elto]) = [[elto]]
-- dividir (primero:segundo:resto) =
--     [primero:restolista1, segundo:restolista2]
--     where [restolista1, restolista2] = dividir
--         (resto)

-- --PRE: Longitud(lista) = 2
-- combinar :: [Solucion] -> Solucion
-- combinar [], lista2 = lista2
-- combinar [lista1, []] = lista1
-- combinar [primero1:resto1, primero2:resto2] =
--     if primero1 <= primero2 then
--         primero1:(combinar [resto1, primero2:resto2])
--     else
--         primero2:(combinar [primero1:resto1, resto2])

-- mergesort :: Problema -> Solucion
-- mergesort = dyv estrivial resolver dividir combinar

-- --Ejemplo utilización:
-- --mergesort [3,6,1,3]
-- ---

```

-- El esquema de Simplificación es una variante del  
 -- esquema de Divide y Vencerás (es, de hecho, un  
 -- caso particular).

-- En este esquema el problema a resolver no se divide  
 -- en una lista de subproblemas que se deban resolver  
 -- luego "mapeando" el esquema recursivo de DyV y cuyas  
 -- soluciones se deban combinar, sino que el problema  
 -- a resolver se "divide" a un solo subproblema.  
 -- Se dice que el problema se simplifica, se reduce.

-- Todo lo que podemos resolver con el esquema de  
 -- Simplificación lo podemos resolver con el de  
 -- Divide y Vencerás.

-- Un caso paradigmático de este tipo de técnica es  
 -- el de la búsqueda de una palabra en un diccionario.  
 -- Actuando de una forma simplista (sin considerar  
 -- la distribución real de las palabras, etc.), abriríamos el  
 -- diccionario por la mitad y, según la palabra que buscamos

```

-- esté antes o después, el problema se nos "simplifica" a
-- la búsqueda de la palabra en la primera o segunda mitad
-- del diccionario. Así, hasta que encontremos la palabra.
-- Nota: hemos supuesto que abrimos el diccionario por la
-- mitad para que se ajuste mejor al caso de búsqueda binaria.

-- -- JC 3-3-2001 versión currificada en Haskell

-- module Simplif (simplificacion) where

-- simplificacion::
--      (tproblema -> Bool)          -- Es Trivial ?
--      -> (tproblema -> tsolucion)  -- Resolver Caso
Trivial
--      -> (tproblema -> tproblema)  -- Simplificar
--      -> tproblema                 -- problema
--      -> tsolucion
-- simplificacion estrivial resolver simplificar problema =
--      if estrivial (problema) then
--          resolver (problema)
--      else
--          simplificacion estrivial resolver simplificar
--              (simplificar (problema))

-- ---

-- Búsqueda binaria:

-- Este tipo de búsqueda surge cuando la entrada
-- está ordenada respecto a cierta relación de orden.

-- Un caso parecido de este tipo de búsqueda lo tenemos al
-- buscar una palabra en un diccionario. Abrimos el
-- diccionario por la mitad y, según la palabra que buscamos
-- esté antes o después, el problema se nos "simplifica" a
-- la búsqueda de la palabra en la primera o segunda mitad
-- del diccionario. Así, hasta que encontremos la palabra.
-- Nota: hemos supuesto que abrimos el diccionario por la
-- mitad para que se ajuste mejor al caso de búsqueda binaria.

-- Comentaremos el caso de la búsqueda de un número
-- en una entrada ordenada, con el objetivo de devolver
-- el índice donde está el elemento buscado.

-- Si el vector es [1,3,4,5,7,8,9,10] y el elemento a buscar es
-- el 3:

-- Definimos dos índices iizq e ider, con valores iniciales 1 y 8.
-- Consideramos en entrada[ider DIV 2]; si es ele es menor
-- que entrada[ider DIV 2], nos quedamos con la primera
-- mitad, definida ahora por iizq=1 e ider=4; si es mayor,
-- nos quedamos con la otra mitad. Esto se realiza sucesivamente
-- mediante llamadas recursivas hasta encontrar el número buscado.

-- En nuestro caso, por ejemplo, en la primera vez nos quedaríamos
-- con la primera mitad; en la segunda también, etc.

-- En Modula-2:

```

```

-- TYPE TipoElemento = INTEGER;
--     TipoDatos = ARRAY [0..MAX-1] OF TipoElemento;

-- PROCEDURE BusquedaBinaria (VAR datos: TipoDatos; ele:
TipoElemento;
--                               iizq, iber: INTEGER):
--                               INTEGER;
--
-- CONST FLAG = -1;
-- VAR mitad: INTEGER;
-- BEGIN
--   IF (iizq = iber) THEN
--     (* si se encuentra, se devuelve el índice; si no, -1 *)
--     IF (datos [iizq] = ele) THEN RETURN iizq ELSE RETURN FLAG END;
--   ELSE
--     mitad := (iizq + iber) DIV 2;
--     IF (ele <= datos [mitad]) THEN
--       RETURN BusquedaBinaria (datos, ele, iizq, mitad)
--     ELSE
--       RETURN BusquedaBinaria (datos, ele, mitad+1, iber)
--     END
--   END;
-- END BusquedaBinaria;

```

```

-- (En imperativo:)
--  $T(n) = k_1, n = 1$ 
--  $= T(n/2) + k_2, n > 1$ 

```

```

-- En este caso:  $a = 1; b^k = 2^0 = 1$ 
-- Estamos en el caso de  $a = b^k$ , por lo que, según el teorema,
--  $T(n) = O(n^k \log n) = O(\log n)$ 

```

```

-- Vemos que es menor que lineal. Pero no es ilógico, debemos
-- tener en cuenta que la entrada estaba ordenada.

```

```

-- Nota: en el caso del paradigma funcional, tendríamos
-- otra ecuación de recurrencia, debido a que no
-- podemos en general dividir con complejidad constante:

```

```

-- (En funcional:)
--  $T(n) = k_1, n = 1$ 
--  $= T(n/2) + k_2 \cdot n, n > 1$ 

```

```

-- En este caso:  $a = 1; b^k = 2^1 = 2$ 
-- Estamos en el caso de  $a < b^k$ , por lo que, según el teorema,
--  $T(n) = O(n^k) = O(n)$ 

```

```

-- Tenemos en este caso una complejidad lineal.

```

```

-- ---

```

```

-- Problema de la potencia de un número:

```

```

-- Podemos abordar el problema del cálculo de
-- la potencia de un número con una estrategia de
-- divide y vencerás:

```

```

-- 2^6 = 2^3*2^3
-- 2^7 = 2*2^6

-- Si n es par:
-- 2^n = 2^(n/2)*2^(n/2)      (ó 2^2n = 2^n*2^n)
-- (dos subproblemas o más bien uno, como se comenta más adelante)

-- Si n es impar:
-- 2^n = 2*2^n      (ó 2^2n+1 = 2*2^2n)
-- (un subproblema)

-- De hecho, es más bien un problema de simplificación,
-- porque nos damos cuenta que los subproblemas del caso n par
-- son idénticos.

-- --module Potencia (potencia, Problema, Solucion) where

-- --!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
-- --!!!!!!! Potencia con complejidad logarítmica !!!!!!!!!!!!!!!
-- --!!!!!!! Marta Patiño !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
-- --!!!!!!! 22-4-99 !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
-- --!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!

-- -- JC versión currifica en Haskell 3-3-2001

-- import Divyvinc

-- type Problema = (Int, Int)
-- type Solucion = Int

-- estrivial:: Problema -> Bool
-- estrivial (base, exp) = exp <= 1

-- resolver :: Problema -> Solucion
-- resolver (base, exp) = if exp == 1 then
--     base
--     else
--     1

-- dividir :: Problema -> [Problema]
-- dividir (base, exp) = if (exp `mod` 2) == 0 then
--     [(base, exp `div` 2)]
--     else
--     [(base, 1), (base, exp-1)]

-- combinar :: [Solucion] -> Solucion
-- combinar ([resultado]) = resultado * resultado
-- combinar ([base, resultado]) = base * resultado

-- potencia :: (Int, Int) -> Int
-- potencia = dyv estrivial resolver dividir combinar

-- --Ejemplo utilización:
-- --potencia (3, 4)

-- T(N) = k1, N = 1

```

```

--      = T(N/2) + k2, N > 1

-- En este caso: a = 1; b^k = 2^0 = 1
-- Estamos en el caso de a = b^k, por lo que, según el teorema,
-- T(N) = O(N^k * logN) = O(logN)

-- ---
-- ---

-- -- JC 3-3-2001 versión en Haskell

-- import Simplif

-- data Arbol a = C_vacio | C_arbol (Arbol a, a, Arbol a)

-- type Problema a = (a, Arbol a)

-- type Solucion = Bool

-- estrivial:: Eq a => Problema a -> Bool
-- estrivial (valor, C_vacio) = True
-- estrivial (valor, C_arbol (_,nodo,_)) = valor == nodo

-- resolver:: Problema a -> Solucion
-- resolver (valor, C_vacio) = False
-- resolver (valor, C_arbol(_,_,_)) = True

-- simplificar:: Ord a => Problema a -> Problema a
-- simplificar (valor, C_arbol (h_izdo, nodo, h_dcho)) =
--   if valor < nodo then (valor, h_izdo)
--   else (valor,h_dcho)

-- estaenarbol:: Ord a => Arbol a -> a -> Bool
-- estaenarbol arbol valor =
--   simplificacion estrivial resolver simplificar (valor, arbol)

-- ---

-- --Ejemplo utilización:

-- --arbol_ejemplo :: Arbol Int
-- --arbol_ejemplo =
-- --   C_arbol(
-- --     C_arbol(
-- --       C_arbol(C_vacio,1,C_vacio),
-- --       3,
-- --       C_arbol(C_vacio,4,C_vacio)),
-- --     5,
-- --     C_arbol(C_vacio,7,C_vacio))

-- --estaenarbol arbol_ejemplo 4

-- --estaenarbol arbol_ejemplo 6

-- T(n) = k1, n = 1
--       = T(n/2) + k2, n > 1

```

```

-- En este caso:  $a = 1$ ;  $b^k = 2^0 = 1$ 
-- Estamos en el caso de  $a = b^k$ , por lo que, según el teorema,
--  $T(n) = O(n^k \log n) = O(\log n)$ 

-- ---

-- Método de ordenación rápida (Quicksort)

-- En este método la parte clave es la fase de división
-- del problema, mientras que la fase de combinación
-- es trivial (a diferencia del método de ordenación por mezcla).

-- Es un método rápido y eficiente de ordenación, aunque
-- la complejidad del caso peor no es buena.
-- (La división de la entrada en dos partes de tamaño
-- similar (es decir, en dos "mitades") no está garantizada.)

-- Fase de división:
-- Se elige arbitrariamente un elemento denominado "pivote"
-- (en general, el primero), y se divide la entrada en
-- elementos menores y mayores que el pivote.

-- La fase de combinación es luego trivial, habrá luego
-- simplemente que concatenar la subsolución de elementos
-- menores que el pivote (que estarán ya ordenados),
-- el elemento pivote y la subsolución de elementos mayores
-- que el pivote (que estarán ya ordenados)

-- Ejemplo:
--   |6|2|5|7|9|8|4|3|      (Pivote: 6)
--   |2|5|4|3|      |6|      |7|9|8|
--
--   |2|3|4|5|      |6|      |7|8|9|
--   |2|3|4|5|6|7|8|9|

--
--
--Ejemplo de división de los sucesivos subproblemas, siguiendo
aspectos de
--la implementación en Haskell del algoritmo:
--           [26, 5, 37, 1, 61, 11, 59, 15, 48, 19]      (Pivote: 26)
--           [5, 1, 11, 15, 19]      26      [37, 61, 59, 48]
--   [1] 5 [11, 15, 19]      [] 37 [61, 59, 48]
--           [] 11 [15, 19]      [59, 48] 61 []
--           [] 15 [19]      [48] 59 []
--
--[1, 5, 11, 15, 19, 37, 48, 59, 61]
--
--
-- --module Quicksort2 (quicksort, Problema, Solucion) where
--
-- --ordena una lista utilizando el algoritmo de quicksort
-- --basado en el esquema generico de divide y venceras.
-- --Es trivial resolver listas de 2 ó 1 ó 0 elementos
--
-- -- JC 3-3-2001 versión en Haskell
--
-- import Divyvinc

```

```

-- type Problema = [Int]
-- type Solucion = [Int]

-- estrivial :: Problema -> Bool
-- estrivial ([]) = True
-- estrivial ([primero]) = True
-- estrivial (primero:segundo:resto) = resto == []

-- resolver :: Problema -> Solucion
-- resolver ([]) = []
-- resolver ([primero]) = [primero]
-- resolver (primero:segundo:[]) =
--     if primero <= segundo then
--         primero:segundo:[]
--     else
--         segundo:primero:[]

-- --Nota: "Ord a" en declaración por utilización del "<" estándar
-- separarmayoresymenores :: Ord a => ([a], a) -> ([a], [a])
-- separarmayoresymenores ([], n) = ([], [])
-- separarmayoresymenores (primero:resto, n) =
--     let (menores, mayores) = separarmayoresymenores (resto, n) in
--     if primero < n then
--         (primero:menores, mayores)
--     else
--         (menores, primero:mayores)

-- dividir :: Problema -> [Problema]
-- dividir ([]) = []
-- dividir (primero:resto) =
--     [menores, [primero], mayores] where
--         (menores, mayores) = separarmayoresymenores (resto,
primero)

-- combinar :: [Solucion] -> Solucion
-- combinar ([primero, [segundo], tercero]) = concat [primero,
segundo:tercero]

-- quicksort :: Problema -> Solucion
-- quicksort = divv estrivial resolver dividir combinar

```