Técnica de diseño de algoritmos Divide y Vencerás

El método consiste en descomponer el problema original en varios subproblemas más pequeños para luego resolver éstos. Por último, se combinan las soluciones de esos subproblemas para obtener la solución del problema original.

```
función divideYvencerás ( problema ){
     Descomponer el problema en n subproblemas más pequeños
     para i ← 1 hasta n hacer
          Resolver el subproblema i
     fpara
     Combinar las n soluciones
}
```

Si los subproblemas son todavía demasiado grandes, se puede utilizar la misma técnica con ellos, es decir, dividirlos. Para ello se diseña un algoritmo recursivo que vaya dividiendo el problema hasta alcanzar un caso trivial. Un algoritmo del siguiente tipo:

```
función divideYvencerás ( problema ){
     Si  problema es trivial  entonces
         Resolver el problema
     si_no
         Descomponer el problema en n subproblemas más pequeños
         para i ← 1 hasta n hacer
               divideYvencerás(subproblema i)
         fpara
         Combinar las n soluciones
     fsi
}
```

Ejemplos (para ordenar un vector):

- MergeSort: buscar punto medio, ordenar cada mitad por separado y combinar esos subvectores ordenados (mezclar).
- QuickSort: acondicionar el vector, ordenar cada parte por separado.

Algoritmo de ordenación rápida (Quicksort)

El método consiste en dividir el vector en dos partes de forma que la ordenación por separado de cada una de ellas dé como resultado la ordenación total del vector.

Para ello se toma un elemento que se denomina **pivote** de forma que queden a su izquierda todos los elementos menores que él y a su derecha todos los mayores.

A continuación, se aplica el mismo método a las dos partes.

Hay distintas estrategias para seleccionar el pivote y acondicionar el vector. Una de ellas consiste en tomar como pivote el primer elemento del vector y realizar dos búsquedas: una de izquierda a derecha buscando el primer elemento mayor que el pivote y otra de derecha a izquierda buscando el primer elemento menor que el pivote; cuando se hayan encontrado, se intercambian los elementos. Se continúa así hasta que los índices se cruzan, momento en el que se intercambia el pivote por el elemento más pequeño.

```
Tipodef entero=tVector[100]
```

```
acción quickSortCompleto(e/s tVector v, ent entero n)
{Pre: v de tamaño n; 0<=n<=100}
{Post: las n primeras componentes de v ordenadas
crecientemente}
principio
     quickSort(v,0,n-1);
fin

acción quickSort(e/s tVector v,
                                ent entero inf, ent entero sup)
{Pre:     0<=inf<=sup+1<=tam}
{Post:    ∀i,j tq inf<=i<=j<=sup se tiene que v[i]<=v[j]}

variables
     k : entero;
principio
     Si (inf<=sup) entonces
          particion(v,inf+1,sup,v[inf],k);
          permutar(v[inf],v[k]);
          quickSort(v,inf,k-1);
          quickSort(v,k+1,sup);
     fsi
fin

acción particion(e/s tVector v, ent entero iz, ent entero
de,ent entero pivote, sal entero pos)
{Pre: 0<=iz<=de+1<=tam}
{Post: iz<=pos<=de tq
       ∀j con iz<=j< =pos se tiene que v[j]<=pivote
       ∀i con  pos+1<=i<=de se tiene que pivote<=v[i]}
principio
     mientras que( iz<=de) hacer
          mientras que (iz<=de AND v[iz]<=pivote) hacer
               iz ← iz+1;
          fmq
          mientras que (iz<=de AND pivote<=v[de]) hacer
               de ← de-1;
          fmq
          Si (iz<=de) entonces
               permutar(v[iz],v[de]);
               iz ← iz+1;
               de ← de-1;
          fsi
     fmq
     pos ← iz-1;
fin
```

La complejidad del quicksort es, en media, de O(n log n).
En el peor caso llega a $O(n^2)$.

Algoritmo de ordenación rápida (MergeSort)

```
acción mergeSortCompleto(e/s tVector v, ent entero n)
{Pre: v de tamaño n; 0<=n<=100}
{Post: las n primeras componentes de v ordenadas
crecientemente}
principio
     mergeSort(v,0,n-1);
fin

acción mergeSort(e/s tVector v,ent entero inf, ent entero sup)
{Pre:     0<=inf<=sup+1<=tam}
{Post:    ∀i,j tq inf<=i<=j<=sup se tiene que v[i]<=v[j]}

variables
     m : entero;
principio
     Si (inf<=sup) entonces
          m ← (inf + sup) DIV 2
          mergeSort(v,inf,m)
          mergeSort(v,m+1,sup);
          combinar(v,izq,m+1,der)
     fsi
fin

acción combinar(e/s tVector v, ent entero iz,
                ent entero iz2:,ent entero de2)
{Pre: v de tamaño n, 1<=iz1<=iz2<=de2<=n, las componentes de v
de iz1 a iz2 están ordenadas de manera creciente, y las de iz2
a de2 también}
{Post: componentes de iz1 a de2 ordenadas de manera creciente,
mezclando componentes de iz1 a iz2-1 y de iz2 a de2}
variables
    aux : vector [1..100] de entero;
    i, de1, i1,i2,j : entero;
principio
     i1 ← iz1;
     i2 ← iz2;
     de1 ← iz2-1;
     i ← 0;
     mientras que( i1<=de1 AND i2<=de2) hacer
          si (v[i1]<v[i2]) hacer
               aux[i] ←v[i1];
               i1 ← i1+1;
          si_no
               aux[i]<-v[i2];
               i2 ← i2+1;
          fsi
          i ← i+1;
     fmq
     mientras que( i1<=de1) hacer
          aux[i] ←v[i1];
          i1 ← i1+1;
```

```
            i ← i+1;
        fmq
        mientras que( i2<=de2) hacer
            aux[i]← v[i2];
            i2 ← i2+1;
            i ← i+1;
        fmq
        i ← 0;
        para j ← iz1 hasta de2 hacer
            v[j] ← aux[i];
            i ← i+1;
        fpara
fin
```

La complejidad del mergeSort es, en media, de O(n log n).