**Objetos auxiliares - Salvador Jover**

mailto:s.jover@wanadoo.es

# Objetos Auxiliares II

Retomamos ahora, la pequeña aventura iniciada en nuestra cita anterior, renovando ilusiones, con la esperanza de seguir avanzando un poco mas día a día... Adelante pues.

A lo largo del artículo anterior, profundizamos en el concepto de Objeto Auxiliar, obteniendo una visión global de todos aquellos objetos, a los que relacionábamos directa o indirectamente con este concepto. Nombrábamos y describíamos cada uno de ellos, y detuvimos, finalmente, nuestros pasos ante la clase TList.

Como recordareis, y si no fuese así, os ruego sea releída la parte primera de esta serie, hacíamos una pequeña introducción a los tipos de datos abstractos, TDA o TAD según autores, y relacionábamos nuestra clase TList con un tipo de dato de esta naturaleza. Asimismo, y siguiendo esta filosofía, analizábamos uno por uno los distintos métodos u operaciones, que dotaban a la clase de una funcionalidad. Podíamos ver, como eran insertados elementos a nuestra lista, como eran borrados, extraídos, o movidos a una nueva posición. En ese afán nos quedamos. Nos despedimos y se dejo escrita una promesa de retomar en este punto la clase TList para abordar el método que nos falta, y que ciertamente es requerido: Un método de ordenación: el algoritmo de QuickSort de C.A.R. Hoare.

Estaba a punto de dar por finalizada nuestra clase TList. Y yo no podía evitar preguntarme: ¿Habrá variado algo la clase TList con la próxima aparición de Delphi 6? No tenía respuestas... Así que, ni corto ni perezoso, decidí izar las velas de mi barcaza y adentrarme mar adentro, en busca de respuestas, navegando sin dirección....

Metafóricamente se inició un viaje que hoy os resumo. Un viaje que ha durado varias semanas en busca de información sobre TList... Efectivamente, las preguntas han encontrado algunas respuestas que vamos a compartir. Sí. La clase TList se ha visto ampliada por un método, cuyo nombre ya nos es familiar: el método Assign( ). Con él, los programadores de Borland han ampliado la funcionalidad de la clase dotándola de la capacidad de poder operar con dos listas de punteros. Quedarán a nuestro alcance operaciones más típicas de conjuntos, como puede ser la Unión o la Intersección de los elementos de las lista.

Pero no nos adelantemos y veamos por partes esta nueva aventura que comienza.

## Sort o el método de ordenación rápida.

Hacemos un alto en el camino. Vamos a introducir algunas ideas que nos harán comprender el algoritmo implementado en el método Sort, para lo cual, tenemos que abordar, eso sí, muy superficialmente, conceptos más próximos a la programación clásica.

## Objetos Auxiliares II - Salvador Jover

Recursividad, Algoritmo, eficiencia... son algunas palabras claves con un nexo en común: la necesidad de buscar soluciones a los problemas planteados. Sin necesidad de introducir definiciones formales, lejos de convertir este texto en una copia de cualquier manual universitario, diremos que un algoritmo no es más que el conjunto de pasos que son necesarios para dar respuesta a un determinado problema. Es una definición informal, hecha desde el sentido común.

También desde el sentido común, y en esa necesidad de dar solución a los problemas, encontraremos dos enfoques distintos según la existencia o no de recursividad en su naturaleza. Así podemos hacer una distinción entre Algoritmos con un diseño recursivo y aquellos cuyo diseño se nos presenta en términos de iteración, o iterativos.

Partiremos de un ejemplo básico que nos permita llenar de contenido los conceptos planteados. Supongamos que deseamos crear una función que nos calcule la potencia enésima de un entero cualquiera. En ambos casos, nuestra función recibirá como parámetros los valores correspondientes a la base y al exponente de dicha potencia, y habrá de devolver como valor otro entero resultante de dicho calculo deseado.

Veamos la implementación:

**CALCULO DE LA POTENCIA ENESIMA:**

**ALGORITMO ITERATIVO:**

```
function potencia_enesima_iter(base, exponente: integer): integer;
var
contador: integer;
begin
  Result:= 1;
  for contador:= 0 to exponente - 1 do
    Result:= Result * base;
end;
```

**ALGORITMO RECURSIVO:**

```
function potencia_enesima_rec(base, exponente: integer): integer;
begin
  case exponente of
    0: Result:= 1; // solución o caso trivial
  else
    Result:= base * potencia_enesima_rec(base, exponente - 1);
  end;
end;
```

Si nos centramos en el primer algoritmo planteado, vemos como para resolver dicho problema se hace necesario encontrar otro, de distinta naturaleza, supuestamente más sencilla, que pueda ser resuelto en términos de unos datos distintos y más pequeños. En el ejemplo hemos convertido nuestro problema original, elevar a una potencia, en otro cuya naturaleza es distinta: multiplicar dos enteros. De la repetición de dicho problema, un número concreto de veces obtenemos la solución deseada.

$$A^n = \underbrace{A \times A \times A \times \ldots\ldots\ldots \times A}_{\text{n-veces}}$$

Por el contrario, en nuestro segundo algoritmo, nos preguntaremos si es posible resolver nuestro problema original, partiendo del supuesto de que dicho problema, ya se haya resuelto para otros datos mas sencillos, denominados solución trivial del algoritmo. Dicha premisa establece la búsqueda de una relación de recurrencia con respecto al problema original, mediante unos datos mas pequeños. Como podéis ver, sucesivamente será invocada la función ***potencia_enesima_rec***, cuyo parámetro ***exponente*** será reducido en una unidad a cada nueva llamada hasta alcanzar la solución trivial.

$A^n = A \times A^{n-1}$
$A^{n-1} = A \times A^{n-2}$
.....
$A^0 = 1$ [Solución trivial]

## ¿Eficiencia...?

En este punto, y una vez que podemos entender algo tan básico, como que pueden existir múltiples algoritmos para la resolución de un mismo problema, nos daremos cuenta de podemos establecer criterios que permitan afirmar que un algoritmo es mas eficiente que otro. Para estos casos, por convenio, se adoptan distintos criterios en aras de medir, de forma homogénea, todas las posibles soluciones, independientemente del compilador o del procesador en el que sean ejecutados, por enumerar dos razones que pueden modificar nuestra percepción de eficiencia de los mismos. Este criterio viene a denominarse Criterio Asintótico. También por convenio, se suele considerar la eficiencia del algoritmo en el caso peor, aunque en algunos caso pueda tomarse como referencia el promedio obtenido.

Una forma de medir la eficiencia de un algoritmo es básicamente contar el número de instrucciones que se ejecutan en el mismo. Es decir, si somos capaces de hallar este valor, si establecemos unos criterios que nos permitan contar ese número de instrucciones, hallaremos también una relación entre el tamaño del problema y la eficiencia del algoritmo. Lógicamente, cualquier comparación, necesita unas coordenadas sobre las que poder tomar una referencia y obtener una medida. Incorporamos aquí, y para estos ejemplos, la abscisa temporal, o dicho de otra forma, el tiempo de ejecución del algoritmo. En otros problemas, quizás nuestra abscisa de referencia no pueda ser temporal,, sino espacial, y lleguemos a tomar como base la ocupación de memoria. En definitiva, necesitamos una referencia sobre la que hacer una medida de comparación.

Entramos de lleno en la eficiencia. Decir que un algoritmo pertenece a un orden de complejidad de “n” o lineal, es lo mismo que decir que crece en la misma proporción, el tiempo de ejecución respecto al tamaño del problema. Existen, por tanto, distintos ordenes de complejidad según la naturaleza del problema planteado, siendo los mas usuales: '$2^n$' u orden exponencial, '$n^3$', '$n^2$' o cuadrática, 'nlogn', 'n' o lineal. Y aquí llegamos a una reflexión: ¿para todo problema planteado, nuestro algoritmo va a resolverlo? Y empezamos a comprender que no. Para ordenes de complejidad iguales o superiores a la exponencial, un pequeño incremento en el tamaño del problema generará un incremento inabordable en el tiempo de ejecución... Así pues, es importante siempre preguntarse si el algoritmo que hemos diseñado, es mejorable en términos de eficiencia.

¿Que no lo comprendes...? Ummmm.... A ver como os lo explico: Suponed que tenemos que hacer una operación con un vector ordenado de enteros. Por ejemplo, buscar un elemento del vector y devolver verdadero en caso de hallar dicho elemento. Supongamos que recorremos el vector mediante un bucle y al final del mismo, después de recorrer todos y cada uno de los elementos, devolvemos verdadero o falso según esté o no dicho elemento. Suponemos además que tardamos un tiempo X en resolver dicha función y que nuestro algoritmo tiene un orden de complejidad de 'n' o lineal. ¿No será también cierto, que si duplicamos el número de elementos del vector, es decir, duplicamos el tamaño del problema, se duplicará también el tiempo en resolverlo? Podemos de igual manera, plantear algoritmos que respondan a los distintos ordenes de complejidad planteados, pero desgraciadamente, es algo que se sale del alcance de este artículo. No obstante, sí que vamos a establecer al menos una jerarquía de órdenes de complejidad, de mayor a menor grado de eficiencia:

$$\log n - n - n\log n - n^2 - n^3 - 2^n - n!$$

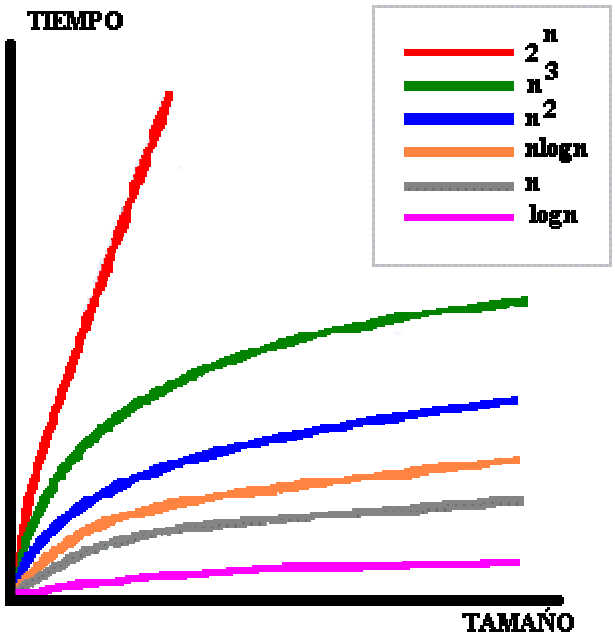


**Figura 1. Gráfica de Tasas de Crecimiento**

Para comprenderlo mejor, la figura 1 os puede dar una idea de las distintas tasas de crecimientos

Vemos pues, como la eficiencia es una medida relativa que relaciona magnitudes. .

## No seas pesado y vamos a lo que vamos...

Llegado este punto, podemos introducir la implementación hecha por el método Sort de la clase TList. Para empezar veremos como se implementa el método:

```
procedure TList.Sort(Compare:
TListSortCompare);
begin
if (FList <> nil) and (Count > 0) then
  QuickSort(FList, 0, Count - 1,
Compare);
end;
```

Posiblemente os haya pasado como a mí. Nos gusta ver las cosas evidentes y mientras vamos analizando el código, identificando cada uno de los parámetros, tipos y variables, nos detenemos en Compare. Pero, ¿qué es Compare?. Aquí ha saltado la primera voz de alarma. Nos hemos encontrado algo que a primera vista nos resulta especial. En una primera aproximación podemos percibir que nos encontramos frente a un tipo de dato definido y rápidamente, nos localizamos la linea en la que se declara TlistSortCompare.

Ahhh... Esto está un poco más claro:

```
TlistSortCompare = function (Item1, Item2: Pointer): Integer;
```

Abrimos el manual de Object Pascal que nos facilita Borland, y leemos literalmente:

**“Si se toma la cabecera de un procedimiento o una función y se elimina el identificador que va a continuación de las palabras procedure o function, lo que queda es el nombre de un tipo de procedimiento. Estos nombres de tipo pueden emplearse directamente en las declaraciones de variables...**
**...Todas estas variables son punteros a procedimientos, es decir, punteros a la dirección de un procedimiento o una función...”**

¡Ya está claro...! Nuestro parámetro va a ser un puntero a la dirección de una función. Pero ¿yo no veo por ningún lado la función?. ¿No deberían facilitarme una?

Pero claro, algunas cosas no llegan a ser tan evidentes. Para empezar, no estamos teniendo en cuenta que trabajamos con punteros, porque, claro está, ¿hacia donde apunta el puntero?. Averiguaremos posteriormente que la función Compare tiene una misión muy limitada: decirnos si dos valores son iguales, mayor el primero respecto al segundo o viceversa. Pero como parámetros han de recibir punteros por definición. Por lo que ¿como se puede saber entonces, a que estructura va apuntar el puntero si no somos nosotros mismos, que hemos creado la lista?

Si observáis atentamente el **listado 1**, donde aparece un ejemplo de utilización del método Sort, hemos implementado la función que espera como parámetro nuestro método:

```
function Compara(a, b: Pointer): Integer;
begin
  Result:= PInteger(a)^ - Pinteger(b)^;
end;
```

y posteriormente, en el cuerpo del procedimiento, la hemos invocado, entregando la dirección de dicha función. Aquí tenemos un ejemplo del uso del operador '@' que nos devuelve una dirección.

```
Lista.Sort(@Compara);
```

Adentrándonos propiamente en la implementación advertimos que previo a invocar el procedimiento QuickSort, se ha de comprobar que realmente está asignada nuestra lista y que contiene al menos un elemento. Cumplidas esas condiciones se invocará dicho procedimiento.

Y hemos llegado al fin a nuestro QuickSort.

```
procedure TForm1.Button1Click(Sender: TObject);
{Recibimos como parámetros dos punteros, por lo que debemos resolverlos para obtener los
valores a los que apuntan y así poder operar con ellos. Este proceso obliga a extremar la
precaución de que realmente sean punteros del tipo apropiado, que puede echar por tierra
nuestro código}
 function Compara(a, b: Pointer): Integer;
 begin
   Result:= PInteger(a)^ - Pinteger(b)^;
 end;

var
 Lista: TList;
 P: ^Integer;
 Indice: Integer;
begin

Lista:= TList.Create;//Creamos la lista para lo cual llamamos a Create de TList.

try
    New(P); //reservamos memoria para nuestro puntero
    P^:= 27; // procedemos a la asignación de un valor entero
    Lista.Add(P); // añadimos el puntero a nuestra lista

    New(P);  // repetimos el proceso anterior para cada valor de la lista
    P^:= 5;
    Lista.Add(P);

    New(P);
    P^:= 9;
    Lista.Add(P);

    New(P);
    P^:= 8;
    Lista.Add(P);

    New(P);
    P^:= 6;
    Lista.Add(P);

    New(P);
```

```
   P^:= 25;
   Lista.Add(P);

{Toda vez que tenemos nuestra lista preparada, podemos proceder a ordenarla. Tal y como
comentamos a lo largo del artículo, pasaremos como parámetro un puntero a la dirección de
una función, cuya misión será simplemente decir a A es mayor, menor o igual que B. De esa
forma hemos sido nosotros los que hemos establecido el criterio de ordenación}

    Lista.Sort(@Compara);

{Este ejemplo se vera más claramente desde el código fuente que se acompaña, ya que he
incorporado 6 componentes de la clase TEdit para reflejar el estado anterior de la lista
y estos seis más que son asignados a continuación para ver el estado final en que queda.}
    p:= Lista.Items[0];
    edit1.text:= IntToStr(p^);

    p:= Lista.Items[1];
    edit2.text:= IntToStr(p^);

    p:= Lista.Items[2];
    edit3.text:= IntToStr(p^);

    p:= Lista.Items[3];
    edit4.text:= IntToStr(p^);

    p:= Lista.Items[4];
    edit5.text:= IntToStr(p^);

    p:= Lista.Items[5];
    edit6.text:= IntToStr(p^);

//recorremos la lista para poder obtener la dirección del contenido de cada puntero
   for Indice:= 0 to Lista.Count - 1 do
      begin
      p:= Lista.Items[Indice]; // Apuntamos con el puntero p a cada elemento de la lista
      dispose(p); // liberamos la memoria a la que apunta p
      end;

finally
  Lista.Free; // finalmente procedemos a liberar nuestra lista
end;
end;
```

**Listado 1. Ejemplo de invocación al método Sort.**

Cualquier estudiante de informática, en sus primeros años universitarios, se entregará al aprendizaje de las estructuras de datos más habituales, al estudio de los algoritmos y demás temas básicos en la programación. El estudio del Algoritmo de QuickSort de C.A.R. Hoare es un paso obligado en dicho aprendizaje, entre otros motivos por ser un algoritmo con un excelente orden de complejidad (nlogn), siempre que se den las condiciones requeridas. Frente a algoritmos ineficaces como el de la Burbuja, éste algoritmo recursivo obtiene un rendimiento como ya se ha comentado excelente.

Dentro de la clasificación habitual, pertenece al grupo de algoritmos que ordenan por partición. Es decir, está basado en la estrategia de diseño conocida por Divide y Vencerás, donde se elige un elemento del vector o lista que hará las veces de pivote. Sobre este elemento, y establecidos un tope por la derecha y otro por la izquierda, nos iremos acercando al pivote, intercambiando aquellos elementos mayores que el pivote y que están a la izquierda del mismo, con aquellos que son menores y se sitúan a la derecha del pivote. Toda vez que se cruzan los indices que controlan ese acercamiento, será invocado recursivamente el procedimiento dos

veces, una para el bloque a la izquierda incluyendo el pivote, y otra para el bloque ordenado a la derecha del pivote, iniciándose de nuevo el proceso de intercambio para estos dos nuevos procesos.

De existir, dicho procedimiento sería algo así:

```
procedure QuickSort(SortList: PPointerList; L, R: Integer; SCompare: TListSortCompare);
var
I, J: Integer;
P, T: Pointer;
begin
I:= L;
J:= R;
P:= SortList^[(L+R) shr 1];
    repeat
      while SCompare(SortList^[I], P) < 0 do
        Inc(I);
      while SCompare(SortList^[J], P) > 0 do
        Dec(J);
      if I <= J then
      begin
        T := SortList^[I];
        SortList^[I] := SortList^[J];
        SortList^[J] := T;
        Inc(I);
        Dec(J);
      end;
    until I > J;

    if L < J then
      QuickSort(SortList, L, J, Scompare); // primera invocación recursiva
    if I < R then
      QuickSort(SortList, I, R, Scompare); // segunda invocación recursiva
 end;
```

Digo de existir, porque realmente NO existe. El algoritmo implementado en QuickSort es tal y como aparece a continuación:

```
procedure QuickSort(SortList: PPointerList; L, R: Integer;
  SCompare: TListSortCompare);
var
  I, J: Integer;
  P, T: Pointer;
begin
  repeat
    I := L;
    J := R;
    P := SortList^[(L + R) shr 1];
    repeat   // iteración que sustituye a la segunda invocación recursiva
      while SCompare(SortList^[I], P) < 0 do
        Inc(I);
      while SCompare(SortList^[J], P) > 0 do
        Dec(J);
      if I <= J then
      begin
        T := SortList^[I];
        SortList^[I] := SortList^[J];
        SortList^[J] := T;
        Inc(I);
        Dec(J);
      end;
    until I > J;
```

```
    if L < J then
      QuickSort(SortList, L, J, Scompare); // primera invocación recursiva
    L := I;
  until I >= R;
end;
```

Y aquí llegamos al primer interrogante: ¿en qué se diferencia la implementación hecha por los programadores de Borland y la propuesta originalmente por Hoare en su algoritmo original? ¿por qué razón ha sido modificado?.

## Analizando ambos algoritmos.

Vamos a intentar resolver estos interrogantes. Podéis seguir con mayor claridad el desarrollo, ojeando el ejemplo que he introducido en la **figura 2** y que compara una hipotética ordenación de una lista de 6 elementos. Si bien no es un ejemplo demasiado representativo, al menos, puede dar una idea visual de las diferencias entre los dos algoritmos.

En los dos casos, prescindiendo de la declaración de las variables y del bucle repeat until del segundo, lo primero a hacer es obtener los valores de los indices de recorrido tanto por la derecha como por la izquierda. Estos valores, pasados como parámetros y respectivamente 0 y Count - 1, es decir, el primer y ultimo elemento de la lista, nos servirán para poder calcular el pivote sobre el que podremos hacer los intercambios. A primera vista, podríamos pensar que dichos valores carecen de importancia y sin embargo, no es así. Precisamente, la obtención de un orden de complejidad de 'nlogn' viene dado de la obtención de un valor medio en nuestro vector o lista. De no ser así y si supuestamente tomásemos como pivote un extremo de la lista, nuestra eficiencia se vería reducida y se obtendría así un orden de complejidad cuadrática, a todas luces ineficaz. Para remediar esto, se opta por obtener el elemento central de nuestra lista.

Queda así justificada la elección de la siguiente asignación: P:= SortList^[(L+R) shr 1], en donde lo que realmente obtenemos es el termino central. Como ya sabéis, un desplazamiento bit a bit hacia la derecha (shr) lo que realmente producirá es un división por 2 del dividendo (L + R), ya que estamos operando en base 2 o binario. Es decir, la variable de tipo puntero P tomará como valor el contenido el termino central de nuestra lista.

Toda vez que se han fijado los valores de los índices y el pivote, podemos iniciar el proceso de intercambio. Este bloque que delimita dicho proceso viene marcado por el segundo REPEAT hasta que se cumpla la condición (I > J), que obligará a que se crucen dichos índices en su recorrido a través de la lista. En dicho recorrido se seguirán los siguientes pasos:

```
while SCompare(SortList^[I], P) < 0 do Inc(I);
```

Comparamos el valor de la lista en la posición [I] con nuestro pivote. Recordemos que el indice I nos recorría el vector partiendo de la posición más a la izquierda (L). Así pues, mientras el valor evaluado sea menor que el pivote, incrementamos I, en busca de un elemento que sea mayor, para poder proceder a ordenarlo posteriormente. En el momento en que localicemos dicho valor nos saldremos del bucle While, para iniciar el proceso de búsqueda a la derecha de la lista.

```
while SCompare(SortList^[J], P) > 0 do Dec(J);
```

Comparamos el valor de la lista en la posición [J] con nuestro pivote. Comentamos en páginas anteriores que al invocar el método Sort, entregábamos como parámetro la dirección de una función. Aquí vemos que se

invoca a dicha función pasando como parámetros de comparación los valores que nos interesan. Esto que ahora se comenta, es válido para el While anterior. Solamente en el caso de que el valor en el elemento J de nuestra lista, sea mayor que el valor del pivote, procederemos a decrementar J.

```
if I <= J then
      begin
        T := SortList^[I];
        SortList^[I] := SortList^[J];
        SortList^[J] := T;
        Inc(I);
        Dec(J);
      end;
```

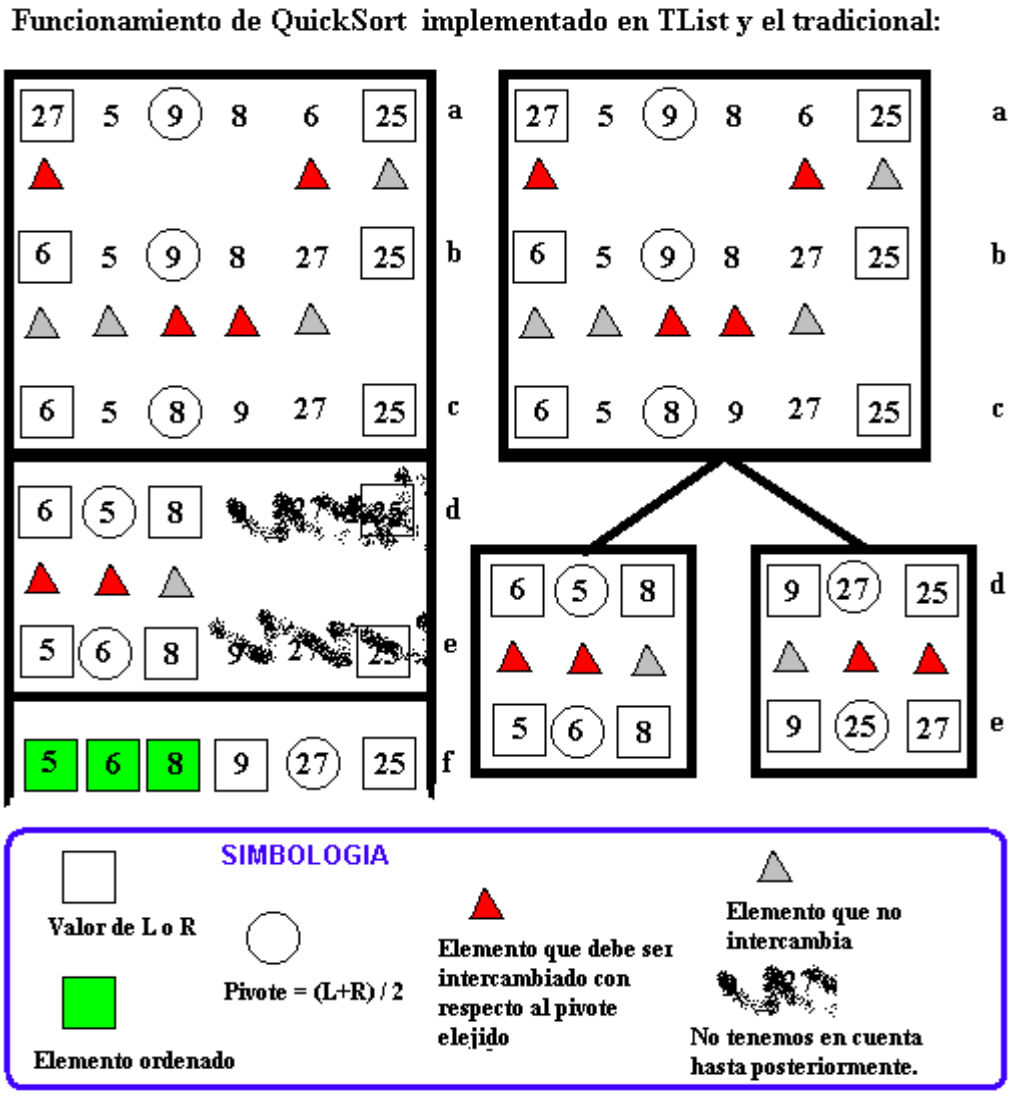


Figura 2 - Desarrollo visual de ambos algoritmos con un ejemplo

Como condición: que I no sea mayor que J. En caso de ser esto cierto, se proceden a intercambiar el contenido en dichas posiciones, para la cual, se ha referenciado un puntero más, (T), que almacenará el valor de uno de los elementos, permitiendo hacer el intercambio. T vale I, I vale J, J vale T y T ya no nos interesa, pues a cumplido su misión. Como ya se ha efectuado dicho cambio, se procede a incrementar I y decrementar J para seguir buscando nuevos elementos que se puedan intercambiar.

```
Until I > J
```

Finalmente la condición 'until' nos mantiene en el bucle hasta que se cruzan los indices I y J.

Y ahora viene la parte menos intuitiva. Hasta el momento, nuestro algoritmo tan solo sabe ordenar los elementos de una lista sobre un pivote elegido. Intercambia todos los elementos que hace falta hasta dejar a la izquierda del pivote, aquellos elemento menores que el, y a su derecha, todos los mayores. Si os fijáis en el paso 'c' del primer y segundo algoritmo de la figura 1, los elementos 6, 5 y 8 cumplen ser menores que el pivote, pero sin embargo, todavía no están ordenados.

Aquí, y por esa misma razón, intervienen las invocaciones recursivas que precisamente, su misión será terminar de ordenar los elementos no ordenados. En el algoritmo original de Hoare, se producen dos llamadas a la función. La primera llamada asume como parámetros L y J, mientras que la segunda, lo hace con I y R, dividiendo la lista en dos nuevos bloques, que a su vez, cada uno de ellos se subdividirá en dos nuevos bloques, y así hasta el final, en un proceso en el que las invocaciones crecen en una progresión 1, 2, 4, 8, 16... es decir $2^0$, $2^1$, $2^2$, $2^3$,..., $2^n$.

Pero nos interesa el algoritmo implementado por Borland. En este caso tenemos una sola llamada recursiva, que abarcará un bloque desde L hasta J, como podéis ver en la figura 1, letra d del primer algoritmo. El resto de elementos los ignora, y tan solo será, cuando salga de todas las llamadas recursivas generadas, cuando retoma dicho resto de elementos al ejecutar la sentencia (L **:=** I), (I en ese momento valía una unidad mas que J pues se habían cruzado) volviendo a repetir todo el bloque de código desde el principio hasta que I sea mayor o igual que R.

Por dicha razón, si miráis el algoritmo 1, letra f de la figura 1, los elementos 5, 6, y 8 ya están ordenados (color verde), e inicia entonces el proceso de ordenación de 9, 27 y 25, repitiendo los mismos pasos seguidos desde la letra 'a' pero con un subvector de menor tamaño.

Como conclusión a todo lo dicho, nos preguntábamos cual era el motivo por el que Borland había elegido esta modificación del algoritmo QuickSort de Hoare, y la única conclusión a la que he llegado, es considerar motivos de precaución. En mi opinión, me parece que se intenta evitar desbordar la pila, de tal forma que sustituye la segunda invocación recursiva, por una estructura iterativa que permitirá que todas las llamadas se hagan a través de una única invocación (la primera), manteniendo prácticamente la misma eficacia. Pero en fin, esto es algo subjetivo y es tan solo mi opinión..

## Un nuevo método: Assign.

Para cada pregunta siempre existe una respuesta... ¿o quizás no?. Me preguntaba, tal y como os he comentado en la introducción, que cosas nuevas, con respecto a esta clase, íbamos a poder disfrutar. No es fácil obtener respuestas, pero es ¿imposible?

Imposible o no, vamos a compartirlas, con las reservas propias de la información que se obtiene a través de Internet.

Este es el prototipo de la nueva función:

```
procedure Assign(ListA: TList; Aoperator: TListAssignOp = laCopy; ListB: TList = Nil);
```

## Objetos Auxiliares II - Salvador Jover

Aquí se introducen unos conceptos nuevos propios de Object Pascal y que casualmente, se han visto abordados por recientes preguntas en los foros del Grupo Albor. Concretamente estoy hablando de pasar parámetros a procedimientos y funciones con valores por defecto. La pregunta de un compañero era clara: tengo una función con muchos parámetros y quisiera obviar parte de ellos en la invocación de la misma. La respuesta también era clara: Object Pascal te permitirá pasar pasar dichos parámetros con valores por defecto, escribiendo un prototipo similar al escrito lineas más arriba. Hecho esto, podemos obviar todos aquellos a los que tienen valores por defecto con la única condición de que estén al final de la lista de parámetros. Es decir, que no se pueden intercalar.

¿Que nos supone a nosotros esto en el actual procedimiento? Veamos la implementación hecha:

```
if ListB <> nil then
begin
  LSource := ListB;
  Assign(ListA);
end
else
  LSource := ListA;
```

Es un ejemplo claro de lo dicho anteriormente. En el caso de que el segundo objeto TList sea distinto de Nil, hecho este que implica la existencia del mismo, se invocará el procedimiento Assign con un único parámetro (ListA). Esto significa, ni más ni menos, que será ejecutado el procedimiento y será sobrescrita la lista destino, desde la que se llama al método, por la de origen (ListA), para que así pueda seguir el resto de ejecución de código y operar con la lista ListB, asignada como origen en la linea anterior *(LSource := ListB;)*

Pero, nos es posible invocar el procedimiento con tan solo los dos primeros parámetros y obviar el segundo TList. En ese caso, también nos lo deja claro: *(LSource := ListA;),* Si en la anterior ocasión, operábamos con la lista destino, sobrescrita por listA, y ListB, ahora es listA quien operará con la lista destino, que vuelvo a comentar, -ya que me parece importante- será la que invoque al método.

Las operaciones que se pueden efectuar entre dos lista son variadas; y son las propias de la teoría de los conjuntos, en cuanto dicho procedimiento combina origen y destino mediante intersección de elementos, unión, etc...

Vamos a ver como se implementa el código de cada una de las opciones, explicando la finalidad de las mismas.

```
Case Aoperator of    // según el valor del operador pasado por parámetro. En el caso de no
                     //recibir ninguno asumirá que recibe laCopy
```

He respetado el ejemplo original que acompaña a la implementación ya que me parece representativo de la operación que se efectúa. El ejemplo lo acompaño a la derecha y en formato de comentario.

- **laCopy:**

```
                                                    // 12345, 346 = 346
  begin
    Clear;
    Capacity := LSource.Capacity;
    for I := 0 to LSource.Count - 1 do
      Add(LSource[I]);
  end;
```

Es el valor por defecto que toma la variable Aoperator.

El efecto que produce es el de sobrescribir la lista de destino con la lista de origen. Si pasamos a analizar las distintas rutinas observamos que se inicia el bloque de código con una llamada al método Clear de la lista destino. Podemos recordar y para eso retomamos conceptos del artículo anterior, como la llamada al método Clear, invocaba respectivamente SetCount(0) y SetCapacity)(0) y liberaba la memoria mediante ReallocMem. Tras esto se fija la nueva capacidad de la lista, asignando a la lista destino la capacidad de la lista origen. Y nos queda para finalizar, recorrer cada uno de los elementos de la lista destino, y proceder a añadir cada uno de los punteros almacenados en la lista origen.

Hecho esto, sobrescrita la lista destino por ListA, estamos preparados para operar entre nuestra lista destino y ListB. Y esa operación puede ser cualquiera de los valores que puede tomar la enumeración del tipo TlistAssignOp.

- **laAnd:**

```
                                                        // 12345, 346 = 34
  for I := Count - 1 downto 0 do
    if LSource.IndexOf(Items[I]) = -1 then
      Delete(I);
```

El efecto que produce es el de la intersección de elementos de dos conjuntos. Nos quedaremos, pues, con todos aquellos elementos comunes a los dos, y serán eliminados el resto en la lista destino.

Un detalle importante de resaltar es el hecho de utilizar la estructura **f*or downto do***, que nos permite, como ya sabéis recorrer en sentido descendente el bucle. De esa forma recorremos hacia atrás en la lista destino, desde el último elemento de la lista, todos y cada uno de los elementos hasta llegar al primero. Mientras hacemos este recorrido, comprobamos que el elemento Items[I], de tipo puntero, es encontrado en la lista origen. Ese “encontrar”, era producido por la función IndexOf( ), devolviendo -1 en caso de finalizar sin éxito esa búsqueda. Y lógicamente, de ser así, será borrado de la lista destino.

¿Por qué para atrás y no hacia delante?. Muy sencillo. Borrar un elemento en sentido ascendente, hubiera provocado el cambio de posiciones, pues todos los elementos se verían adelantados en una posición. Y eso echaría al traste nuestro objetivo.

- **laOr:**

```
                                                  // 12345, 346 = 123456
  for I := 0 to LSource.Count - 1 do
    if IndexOf(LSource[I]) = -1 then
      Add(LSource[I]);
```

El efecto que produce es de obtener una lista que contiene todos y cada uno de los elementos de las dos listas. El bucle recorre todos los elementos de la lista origen, al tiempo que se busca en la lista destino si existe el elemento Lsource[I], de la lista origen, y en el caso de que este no exista, lo añade, obteniendo así en destino todos y cada uno de los elementos (punteros).

- **laXor:**

```
                                                    // 12345, 346 = 1256
  begin
    LTemp := TList.Create;
    try
      LTemp.Capacity := LSource.Count;
      for I := 0 to LSource.Count - 1 do
        if IndexOf(LSource[I]) = -1 then
          LTemp.Add(LSource[I]);
```

```
      for I := Count - 1 downto 0 do
        if LSource.IndexOf(Items[I]) <> -1 then
          Delete(I);
      I := Count + LTemp.Count;
      if Capacity < I then
        Capacity := I;
      for I := 0 to LTemp.Count - 1 do
        Add(LTemp[I]);
    finally
      LTemp.Free;
    end;
  end;
```

El efecto que produce es conservar en la lista destino todos aquellos elementos no comunes con la lista origen mas aquellos que existen en las lista origen y que no existen en la lista destino.

Necesitamos una lista temporal para ambas operaciones. La primera de ellas nos permitirá obtener todos aquellos elementos de la lista origen que no están presentes en la lista destino: Asignamos la capacidad de la lista temporal al numero de elementos en la lista de origen. Seguidamente, recorremos todos los elementos de la lista de origen y cada uno de ellos, es buscado en la lista de destino. En el caso de no obtener resultado, que es lo que pretendemos, son añadidos a la lista temporal.

El segundo bucle ***for downto do,***

```
for I := Count - 1 downto 0 do
      if LSource.IndexOf(Items[I]) <> -1 then
        Delete(I);
```

recorrerá en sentido descendente la lista destino, cuyos elementos serán buscados en la lista de origen y en el caso de que sean encontrados, procederemos a borrarlos en la lista destino.

Así, por un lado tenemos en la lista temporal los elementos de la lista origen no presentes en la lista destino y por otro lado, todos los elementos en la lista destino que no están presentes en la lista origen. Nos queda reunirlos y para ello nos valemos del indice I que suma el total de elementos de la lista temporal y la de destino para poder obtener la nueva capacidad en la lista destino. Nos aseguramos de que existe reservada memoria para los nuevos elementos.

Hecho esto, añadiremos a la lista destino cada uno de los elementos de la lista temporal y finalmente será destruida dicha lista temporal, liberando su memoria.

- **laSrcUnique:**

```
                                                    // 12345, 346 = 125
  for I := Count - 1 downto 0 do
    if LSource.IndexOf(Items[I]) <> -1 then
      Delete(I);
```

Es el efecto contrario a *laAnd*, ya que la condición de búsqueda es que lo encuentre para ser borrado. Solamente serán conservados los elementos de la lista destino que no existan en la lista origen.

- **laDestUnique:**

```
                                                      // 12345, 346 = 6
  begin
    LTemp := TList.Create;
    try
      LTemp.Capacity := LSource.Count;
```

```
        for I := LSource.Count - 1 downto 0 do
          if IndexOf(LSource[I]) = -1 then
            LTemp.Add(LSource[I]);
        Assign(LTemp);
      finally
        LTemp.Free;
      end;
    end;
```

Es la última de las alternativas. Con ella, tendremos en destino aquellos elementos de la lista de origen no comunes.

Necesitamos de una lista temporal para almacenar los resultados de las operaciones parciales. La asignación capacidad de la lista temporal toma el valor de capacidad de la lista de origen nos asegura el espacio suficiente para añadir los elementos. Se recorre en sentido descendente los elementos de la lista origen y se buscan cada uno de ellos en destino, y en el caso resultar sin éxito dicha búsqueda, procedemos a añadirlos a la lista temporal, cumpliendo así el objetivo. Para volcar el contenido de la lista temporal sobre la lista destino tan solo hemos de invocar el método Assign, tomando como parámetro dicha lista temporal. Los parámetros pasados con valores por defecto hacen el resto, sobrescribiendo la lista destino con los elementos de la lista temporal.

Finalmente, destruimos la lista temporal, liberando la memoria.

## Y nos despedimos, por fin, de nuestra clase TList...

Finalizamos aquí nuestro primer objeto auxiliar, la clase TList. Hemos conocido prácticamente todos los detalles de esta importante clase, desgranando procedimientos, funciones, métodos, variables... todo ha sido diseccionado para vuestra curiosidad ¿malsana? No creo... ;-).

En el artículo anterior podíamos observar que, dentro de los objetos auxiliares con capacidad de gestionar listas de punteros, nos encontrábamos también varios descendientes de dicha clase, como lo eran TClassList, TObjectList o TComponentList. En el presente número de la revista, vais a poder acompañar a nuestro compañero Tavo Ibaceta en su recorrido acerca de las ventanas, adentrándonos de forma natural al uso de estos objetos descendientes de la clase TList sobre los que hemos hablado intensamente. Un magnifico artículo, por cierto, que no debéis perderos: ”Ventanas - Reflexiones desordenadas”.

## ¿Y ahora, qué...?

Nuestra siguiente parada: la clase TStringList.

Nos espera un apasionante viaje sobre una lista especializada en algo muy necesario al programador: las listas de cadenas. ¿Estáis dispuestos a viajar un par de números conmigo? Vosotros elegís los puntos de parada. El viaje continua...