[**Algoritmos de Ordenación en Javascript (revisión ES6)**](http://www.etnassoft.com/2017/03/24/algoritmos-de-ordenacion-en-javascript-revision-es6/)

**Introducción**

Hace algún tiempo, revisamos en este blog [algunos de los patrones de ordenación más utilizados en programación](http://www.etnassoft.com/2011/10/17/ordenacion-basica-de-datos-en-javascript/): el bubble sort, selection sort, insertion sort… Javascript ha cambiado mucho desde aquel 2011, por lo que quizá, ha llegado la hora de actualizarlos a los nuevos tiempos y su nueva sintaxis.

**El porqué de re visitar ahora estos algoritmos:**

* Como métodos de ordenación, siempre resultan útiles. Sin embargo, sus implementaciones tradicionales pueden haber quedado obsoletas para los estándares modernos y los actuales motores.
* Reescribir estas funciones es un magnífico ejercicio para practicar las nuevas posibilidades del lenguaje que ya hemos estudiado aquí como la **desestructuración** ([parte 1](http://www.etnassoft.com/2016/07/04/desestructuracion-en-javascript-parte-1/) y [parte 2](http://www.etnassoft.com/2016/07/07/desestructuracion-en-javascript-parte-2-recetas-y-ejemplos/)), [parámetros de arrastre](http://www.etnassoft.com/2014/06/03/el-operador-de-propagacion-en-javascript-ecmascript-6-y-polyfill/), intercambio de valores…
* El nuevo editor de código interactivo del blog, permite de un modo cómodo jugar con los distintos ejemplos e ir viendo inmediatamente sus resultados.

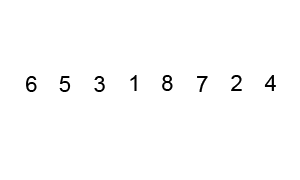
**Algoritmos básicos**

Considerados como los más sencillos, los algoritmos básicos **son aquellos que se estructuran en base a bucles anidados**.

En términos generales, un bucle exterior se encarga de recorrer cada uno de los elementos mientras que otro interno los compara entre sí. El modo en que se realiza esta comparación, es el que determina, y da nombre, a cada algoritmo.

**Bubble Sort**

La **ordenación de burbuja** presenta un esquema muy básico en el que se trabajan los elementos de un conjunto por pares. Cada elemento se compará así con su inmediato sucesor para ordenarse según sea mayor o menor que este. Una vez la pareja está ordenada correctamente, se toma el último número de cada tupla para volver a compararlo con el siguiente.



const bubbleSort = arr => {

const l = arr.length;

for (let i = 0; i < l; i++ ) {

for (let j = 0; j < l - 1 - i; j++ ) {

if ( arr[ j ] > arr[ j + 1 ] ) {

[ arr[ j ], arr[ j + 1 ] ] = [ arr[ j + 1 ], arr[ j ] ];

}

}

}

​

return arr;

};

​

const arr = [10, 4, 40, 32, 67, 12, 43, 31, 65, 1];

const result = bubbleSort(arr);

​

result; *// [1, 4, 19, 12, 31, 32, 40, 43, 65, 67]*

Más información aquí: <https://en.wikipedia.org/wiki/Bubble_sort>

**Insertion Sort**

Este algoritmo, tan sencillo como poco eficiente, trabaja analizando de forma secuencial cada elemento para trasladarlo a su posición correcta dentro del conjunto desplazando al resto.



const insertionSort = arr => {

const l = arr.length;

let j, temp;

​

for ( let i = 1; i < l; i++ ) {

j = i;

temp = arr[ i ];

while ( j > 0 && arr[ j - 1 ] > temp ) {

arr[ j ] = arr[ j - 1 ];

j--;

}

arr[ j ] = temp;

}

​

return arr;

};

​

const arr = [10, 4, 40, 32, 67, 12, 43, 31, 65, 1];

const result = insertionSort(arr);

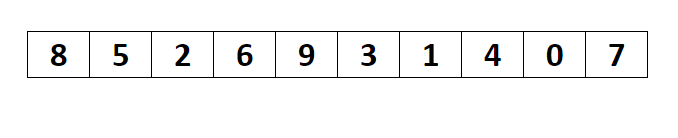
​

result; *// [1, 4, 19, 12, 31, 32, 40, 43, 65, 67]*

Más información aquí: <https://en.wikipedia.org/wiki/Insertion_sort>

**Selection Sort**

Algoritmo en la línea de los anteriores en cuanto a complejidad. El procedimiento recorre el conjunto buscando el elemento de menor valor para intercambiarlo inmediatamente con el de la primera posición. De forma recursiva, se continúan buscando aquellos los siguientes elementos de menor valor para intercambiarlo con sus posiciones finales.



const selectionSort = arr => {

for ( let j = 0; j < arr.length; ++j ) {

let i = iMin = j;

for ( ++i; i < arr.length; ++i ) {

( arr[ i ] < arr[ iMin ] ) && ( iMin = i );

}

[ arr[ j ], arr[ iMin ] ] = [ arr[ iMin ], arr[ j ] ];

}

​

return arr;

}

​

const arr = [10, 4, 40, 32, 67, 12, 43, 31, 65, 1];

const result = selectionSort(arr);

​

result; *// [1, 4, 19, 12, 31, 32, 40, 43, 65, 67]*

Más información aquí: <https://en.wikipedia.org/wiki/Selection_sort>

**Algoritmos complejos**

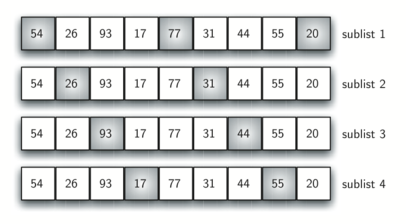
Los algoritmos complejos son aquellos que resultan más efectivos cuanto mayor es el conjunto de datos inicial.

Los procedimientos difieren de un algoritmo a otro pero por lo general, se suele fragmentar el conjunto inicial en subconjuntos más pequeños para trabajarlos de forma independiente.

**Shell Sort**

Para este algoritmo, podemos presentar dos posibles implementaciones; una con gaps fijos y otra en la que estos se calculan de forma dinámica a partir del propio conjunto de entrada.

El Shell sort es un ejemplo de algoritmo complejo donde los cálculos se reparten en vectores. Dichos vectores pueden contener a su vez otros subvectores con la característica de que todos permanecen k-ordenados mientras el espacio disminuye.



**Implementación con GAPS fijos**

Para el siguiente ejemplo, se ha utilizado una secuencia GAP corta de solo tres elementos (5, 3, 1). La secuencia original más óptima para conjuntos de gran tamaño sería:

701, 301, 132, 57, 23, 10, 4, 1

const shellSort = arr => {

const gaps = [5, 3, 1];

for ( let g = 0; g < gaps.length; ++g ) {

for ( let i = gaps[ g ]; i < arr.length; ++i ) {

const temp = arr[ i ];

let j = i;

for ( ; j >= gaps[ g ] &&

arr[j-gaps[ g ] ] > temp;

j -= gaps[ g ] ) {

arr[ j ] = arr[ j - gaps[ g ] ];

}

arr[ j ] = temp;

}

}

​

return arr;

}

​

​

const arr = [10, 4, 40, 32, 67, 12, 43, 31, 65, 1];

const result = shellSort(arr);

​

result; *// [1, 4, 19, 12, 31, 32, 40, 43, 65, 67]*

**Implementación con GAPS generados de forma dinámica**

const shellSort = arr => {

const gap = arr.length;

let h = 1;

while ( h < gap / 3 ) {

h = 3 \* h + 1;

}

while ( h >= 1 ) {

for ( let i = h; i < gap; i++ ) {

for ( let j = i; j >= h && arr[ j ] < arr[ j - h ]; j -= h ) {

[ arr[ j ], arr[ j - h ] ] = [ arr[ j - h ], arr[ j ] ];

}

}

h = ( h - 1 ) / 3;

}

​

return arr;

}

​

​

const arr = [10, 4, 40, 32, 67, 12, 43, 31, 65, 1];

const result = shellSort(arr);

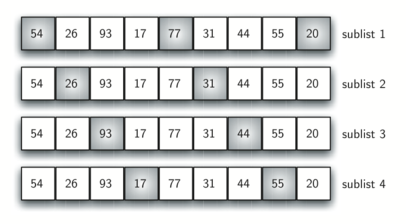
​

result; *// [1, 4, 19, 12, 31, 32, 40, 43, 65, 67]*

Más información aquí: <https://en.wikipedia.org/wiki/Shellsort>

**Merge Sort**

El algoritmo de ordenamiento por mezcla trabaja también mediante el concepto de sub listas. De este modo, el conjunto inicial se divide en dos sub conjuntos de aproximadamente igual tamaño para proceder a su ordenamiento de forma independiente. Finalmente, se mezclan ambos conjuntos para obtener el ordenamiento final.



const mergeSort = arr => {

if (arr.length < 2) {

return arr;

}

​

const middle = parseInt(arr.length / 2) | 0;

const left = arr.slice(0, middle);

const right = arr.slice(middle);

const merge = (left, right) => {

const result = [];

let il = ir = 0;

​

while (il < left.length && ir < right.length) {

result.push( (left[il] < right[ir]) ? left[il++] : right[ir++] );

}

​

return [...result, ...left.slice(il), ...right.slice(ir)];

}

​

return merge(mergeSort(left), mergeSort(right));

}

​

const arr = [10, 4, 40, 32, 67, 12, 43, 31, 65, 1];

const result = mergeSort(arr);

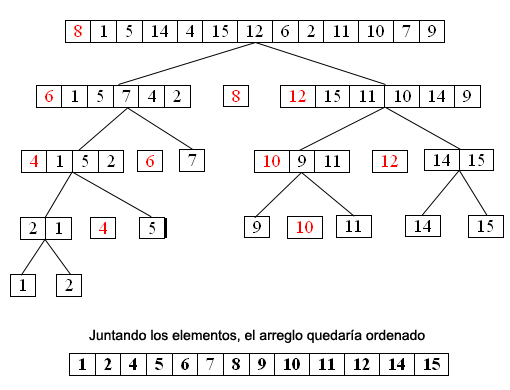
​

result; *// [1, 4, 19, 12, 31, 32, 40, 43, 65, 67]*

Más información aquí: <https://en.wikipedia.org/wiki/Merge_sort>

**Quicksort**

El algoritmo de ordenamiento rápido retoma de nuevo la aproximación del divide y vencerás: a partir de un elemento cualquiera de la lista, denominado pivote, resitua el conjunto restante a ambos lados según sus elementos sean mayores o menores que la muestra escogida. Este proceso se repite de forma recursiva hasta agotar una sublista, quedando con ello el conjunto ordenado.



const quickSort = ( [ x = [], ...xs ] ) => {

return ( x.length === 0 ) ? [] : [

...quickSort( xs.filter( y => y <= x ) ),

x,

...quickSort( xs.filter( y => y > x ) )

];

}

​

const arr = [10, 4, 40, 32, 67, 12, 43, 31, 65, 1];

const result = quickSort(arr);

​

result; *// [1, 4, 19, 12, 31, 32, 40, 43, 65, 67]*

Más información aquí: <https://en.wikipedia.org/wiki/Quicksort>