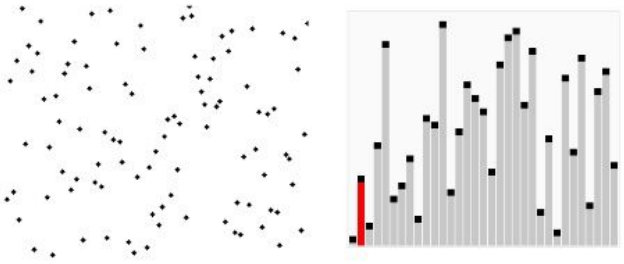
**Metodos de Ordenamiento**

En computación y matemáticas un algoritmo de ordenamiento es un algoritmo que pone elementos de una lista o un vector en una secuencia dada por una relación de orden, es decir, el resultado de salida ha de ser una permutación —o reordenamiento— de la entrada que satisfaga la relación de orden dada. Las relaciones de orden más usadas son el orden numérico y el orden lexicográfico. Ordenamientos eficientes son importantes para optimizar el uso de otros algoritmos (como los de búsqueda y fusión) que requieren listas ordenadas para una ejecución rápida. También es útil para poner datos en forma canónica y para generar resultados legibles por humanos.

Desde los comienzos de la computación, el problema del ordenamiento ha atraído gran cantidad de investigación, tal vez debido a la complejidad de resolverlo eficientemente a pesar de su planteamiento simple y familiar. Por ejemplo, BubbleSort fue analizado desde 1956. Aunque muchos puedan considerarlo un problema resuelto, nuevos y útiles algoritmos de ordenamiento se siguen inventado hasta el día de hoy (por ejemplo, el ordenamiento de biblioteca se publicó por primera vez en el 2004). Los algoritmos de ordenamiento son comunes en las clases introductorias a la computación, donde la abundancia de algoritmos para el problema proporciona una gentil introducción a la variedad de conceptos núcleo de los algoritmos, como notación de O mayúscula, algoritmos divide y vencerás, estructuras de datos, análisis de los casos peor, mejor, y promedio, y límites inferiores.



Debido a que las estructuras de datos son utilizadas para almacenar información, para poder recuperar esa información de manera eficiente es deseable que aquella esté ordenada.
En general los métodos de ordenamiento no son utilizados con frecuencia, en algunos casos sólo una vez. Hay métodos muy simples de implementar que son útiles en los casos en donde el número de elementos a ordenar no es muy grande (ej, menos de 500 elementos). Por otro lado hay métodos sofisticados, más difíciles de implementar pero que son más eficientes en cuestión de tiempo de ejecución.

Los métodos sencillos por lo general requieren de aproximadamente n x n pasos para ordenar n elementos.
Los métodos simples son: insertion sort (o por inserción directa) selection sort, bubble sort, y shellsort, en dónde el último es una extensión al insertion sort, siendo más rápido. Los métodos más complejos son el quick-sort, el heap sort, radix y address-calculation sort. El ordenar un grupo de datos significa mover los datos o sus referencias para que queden en una secuencia tal que represente un orden, el cual puede ser numérico, alfabético o incluso alfanumérico, ascendente o descendente.

**Bubble Sort**

La Ordenación de burbuja es un sencillo algoritmo de ordenamiento. Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiandolos de posición si están en el orden equivocado

Este algoritmo realiza el ordenamiento o reordenamiento de una lista a de n valores, en este caso de n términos numerados del 0 al n-1; consta de dos bucles anidados, uno con el índice i, que da un tamaño menor al recorrido de la burbuja en sentido inverso de 2 a n, y un segundo bucle con el índice j, con un recorrido desde 0 hasta n-i, para cada iteración del primer bucle, que indica el lugar de la burbuja.

La burbuja son dos términos de la lista seguidos, j y j+1, que se comparan: si el primero es mayor que el segundo sus valores se intercambian.

Esta comparación se repite en el centro de los dos bucles, dando lugar a la postre a una lista ordenada. Puede verse que el número de repeticiones solo depende de n y no del orden de los términos, esto es, si pasamos al algoritmo una lista ya ordenada, realizará todas las comparaciones exactamente igual que para una lista no ordenada. Esta es una característica de este algoritmo. Luego veremos una variante que evita este inconveniente.
Para comprender el funcionamiento, veamos un ejemplo sencillo:

```
procedimiento DeLaBurbuja (a_0, a_1, a_2, ..., a_(n-1))
    para i ← 2 hasta n hacer
        para j ← 0 hasta n − i hacer
            si a_(j) > a_(j+1) entonces
                aux ← a_(j)
                a_(j) ← a_(j+1)
                a_(j+1) ← aux
            fin si
        fin para
    fin para
fin procedimiento
```

El algoritmo consiste en comparaciones sucesivas de dos términos consecutivos ascendiendo de abajo a arriba en cada iteración, como la ascensión de las burbujas de aire en el agua, de ahí el nombre del procedimiento. En la primera iteración el recorrido ha sido completo, en el segundo se ha dejado él último término, al tener ya el mayor de los valores, en los sucesivos se irá dejando de realizar las últimas comparaciones, como se puede ver.

A pesar de que el ordenamiento de burbuja es uno de los algoritmos más sencillos de implementar, su orden O (n2) lo hace muy ineficiente para usar en listas que tengan más que un número reducido de elementos. Incluso entre los algoritmos de ordenamiento de orden O (n2), otros procedimientos como el ordenamiento por inserción son considerados más eficientes.

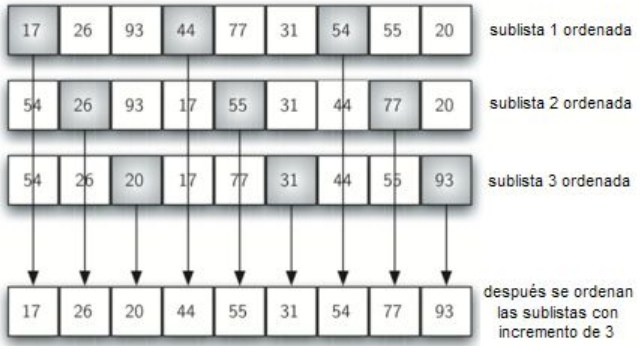
**Shell Sort**

El ordenamiento Shell (Shell sort en inglés) es un algoritmo de ordenamiento. El método se denomina Shell en honor de su inventor Donald Shell. Su implementación original, requiere O(n2) comparaciones e intercambios en el peor caso. Un cambio menor presentado en el libro de V. Pratt produce una implementación con un rendimiento de O(n log2 n) en el peor caso. Esto es mejor que las O(n2) comparaciones requeridas por algoritmos simples pero peor que el óptimo O(n log n). Aunque es fácil desarrollar un sentido intuitivo de cómo funciona este algoritmo, es muy difícil analizar su tiempo de ejecución.

El Shell sort es una generalización del ordenamiento por inserción, teniendo en cuenta dos observaciones:

- El ordenamiento por inserción es eficiente si la entrada está "casi ordenada".
- El ordenamiento por inserción es ineficiente, en general, porque mueve los valores sólo una posición cada vez.

El algoritmo Shell sort mejora el ordenamiento por inserción comparando elementos separados por un espacio de varias posiciones. Esto permite que un elemento haga "pasos más grandes" hacia su posición esperada. Los pasos múltiples sobre los datos se hacen con tamaños de espacio cada vez más pequeños. El último paso del Shell sort es un simple ordenamiento por inserción, pero para entonces, ya está garantizado que los datos del vector están casi ordenados.




El algoritmo Shell es una mejora de la ordenación por inserción, donde se van comparando elementos distantes, al tiempo que se los intercambian si corresponde. A medida que se aumentan los pasos, el tamaño de los saltos disminuye; por esto mismo, es útil tanto como si los datos desordenados se encuentran cercanos, o lejanos.

Es bastante adecuado para ordenar listas de tamaño moderado, debido a que su velocidad es aceptable y su codificación es bastante sencilla. Su velocidad depende de la secuencia de valores con los cuales trabaja, ordenándolos. El siguiente ejemplo muestra el proceso de forma gráfica:

Considerando un valor pequeño que está inicialmente almacenado en el final del vector. Usando un ordenamiento O(n2) como el ordenamiento de burbuja o el ordenamiento por inserción, tomará aproximadamente n comparaciones e intercambios para mover este valor hacia el otro extremo del vector.

**Quicksort**

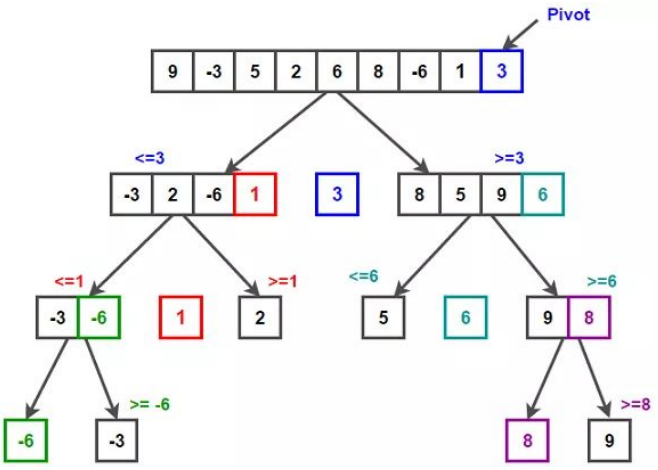
El algoritmo trabaja de la siguiente forma:

- Elegir un elemento del arreglo de elementos a ordenar, al que llamaremos pivote.
- Resituar los demás elementos de la lista a cada lado del pivote, de manera que a un lado queden todos los menores que él, y al otro los mayores. Los elementos iguales al pivote pueden ser colocados tanto a su derecha como a su izquierda, dependiendo de la implementación deseada. En este momento, el pivote ocupa exactamente el lugar que le corresponderá en la lista ordenada.
- La lista queda separada en dos sublistas, una formada por los elementos a la izquierda del pivote, y otra por los elementos a su derecha.
- Repetir este proceso de forma recursiva para cada sublista mientras éstas contengan más de un elemento. Una vez terminado este proceso todos los elementos estarán ordenados.

Como se puede suponer, la eficiencia del algoritmo depende de la posición en la que termine el pivote elegido.

- En el mejor caso, el pivote termina en el centro de la lista, dividiéndola en dos sublistas de igual tamaño. En este caso, el orden de complejidad del algoritmo es O(n·log n).
- En el peor caso, el pivote termina en un extremo de la lista. El orden de complejidad del algoritmo es entonces de $O(n^2)$. El peor caso dependerá de la implementación del algoritmo, aunque habitualmente ocurre en listas que se encuentran ordenadas, o casi ordenadas. Pero principalmente depende del pivote, si por ejemplo el algoritmo implementado toma como pivote siempre el primer elemento del array, y el array que le pasamos está ordenado, siempre va a generar a su izquierda un array vacío, lo que es ineficiente.
- En el caso promedio, el orden es O(n·log n).

No es extraño, pues, que la mayoría de optimizaciones que se aplican al algoritmo se centren en la elección del pivote.



**Ordenamiento Radix**

En informática, el ordenamiento Radix (radix sort en inglés) es un algoritmo de ordenamiento que ordena enteros procesando sus dígitos de forma individual. Como los enteros pueden representar cadenas de caracteres (por ejemplo, nombres o fechas) y, especialmente, números en punto flotante especialmente formateados, radix sort no está limitado sólo a los enteros.

La mayor parte de los ordenadores digitales representan internamente todos sus datos como representaciones electrónicas de números binarios, por lo que procesar los dígitos de las representaciones de enteros por representaciones de grupos de dígitos binarios es lo más conveniente. Existen dos clasificaciones de radix sort: el de dígito menos significativo (LSD) y el de dígito más significativo (MSD). Radix sort LSD procesa las representaciones de enteros empezando por el dígito menos significativo y moviéndose hacia el dígito más significativo. Radix sort MSD trabaja en sentido contrario.

Las representaciones de enteros que son procesadas por los algoritmos de ordenamiento se les llama a menudo "claves", que pueden existir por sí mismas o asociadas a otros datos. Radix sort LSD usa típicamente el siguiente orden: claves cortas aparecen antes que las claves largas, y claves de la misma longitud son ordenadas de forma léxica. Esto coincide con el orden normal de las representaciones de enteros, como la secuencia "1, 2, 3, 4, 5, 6, 7, 8, 9, 10". Radix sorts MSD usa orden léxico, que es ideal para la ordenación de cadenas de caracteres, como las palabras o representaciones de enteros de longitud fija. Una secuencia como "b, c, d, e, f, g, h, i, j, ba" será ordenada léxicamente como "b, ba, c, d, e, f, g, h, i, j". Si se usa orden léxico para ordenar representaciones de enteros de longitud variable, entonces la ordenación de las representaciones de los números del 1 al 10 será "1, 10, 2, 3, 4, 5, 6, 7, 8, 9", como si las claves más cortas estuvieran justificadas a la izquierda y rellenadas a la derecha con espacios en blanco, para hacerlas tan largas como la clave más larga, para el propósito de este ordenamiento, cabe destacar que este método no funciona para la estructura de datos debido a que los ciclos for que se implementaran marcan error debido a las matrices bidimensionales.

Considera esta matriz de entrada


Observador de que 170 ha llegado antes de 90, esto se debe a que apareció anteriormente en la lista original.