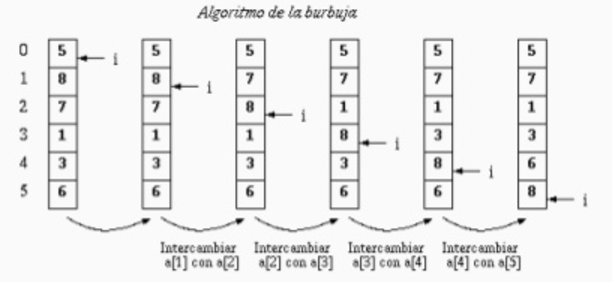
**ALGORITMOS DE ORDENAMIENTO**

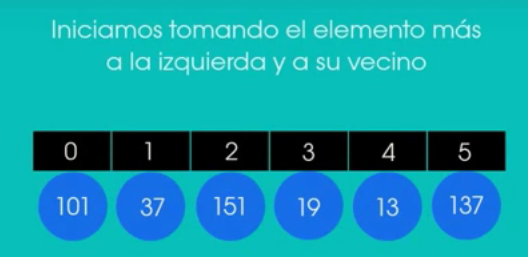
* **BURBUJA**

La **Ordenación de burbuja** es un sencillo [algoritmo de ordenamiento](https://es.wikipedia.org/wiki/Algoritmo_de_ordenamiento). Funciona revisando cada elemento de la lista que va a ser ordenada con el siguiente, intercambiándolos de posición si están en el orden equivocado. Es necesario revisar varias veces toda la lista hasta que no se necesiten más intercambios, lo cual significa que la lista está ordenada. Este [algoritmo](https://es.wikipedia.org/wiki/Algoritmo) obtiene su nombre de la forma con la que suben por la lista los elementos durante los intercambios, como si fueran pequeñas "burbujas". También es conocido como el **método del intercambio directo**. Dado que solo usa comparaciones para operar elementos, se lo considera un algoritmo de comparación, siendo uno de los más sencillo de implementar.



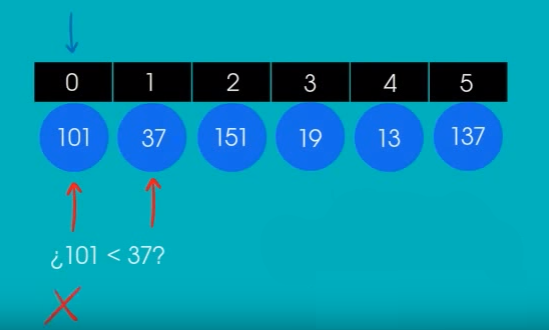
**Ejemplo**:

Supongamos qué queremos ordenar una lista de números almacenados en un arreglo, queremos ordenar de mayor a menor.

Iniciamos tomando el elemento más a la izquierda a su vecino

Nos preguntamos:

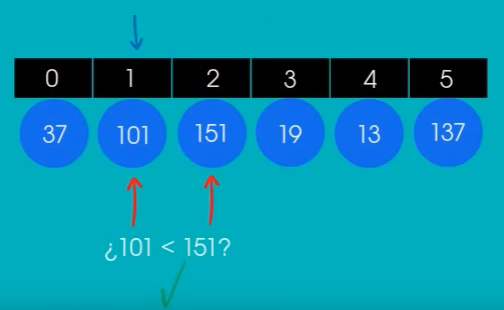
¿101 es menor que 37? Su respuesta obviamente será que no, entonces intercambiamos de lugar los elementos comparados



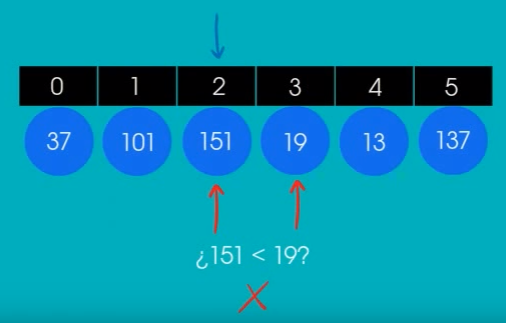
Avanzamos un lugar y comparamos con su vecino.

Y nos preguntamos de nuevo

¿101 es menor que 151?, en este caso la respuesta es sí, cómo el de la izquierda sí es menor entonces no intercambiamos nada

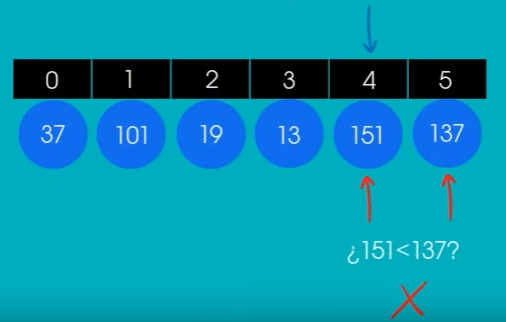


Pasamos al elemento en la siguiente posición y su vecino:

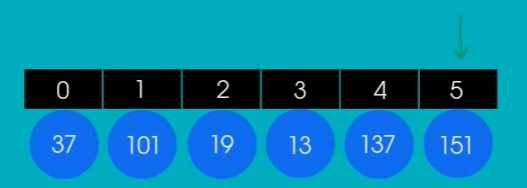
¿151 es menor que 19? La respuesta es no, cómo el elemento no es menor entonces intercambiamos de lugar con el otro.

Avanzamos un nuevo lugar y comparamos con su vecino

¿151 es menor que 137? No, entonces intercambiamos.

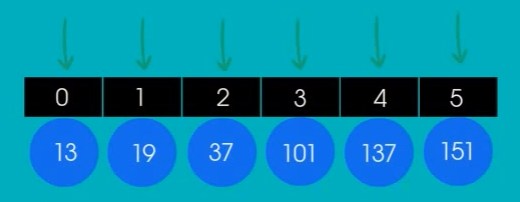


**Observemos:**



Si avanzamos uno más ya no podemos comprar contra nadie. Pero esto no es necesario, ya que el último en la posición ya es el mayor que todos a su izquierda, es decir, está en su posición correcta.

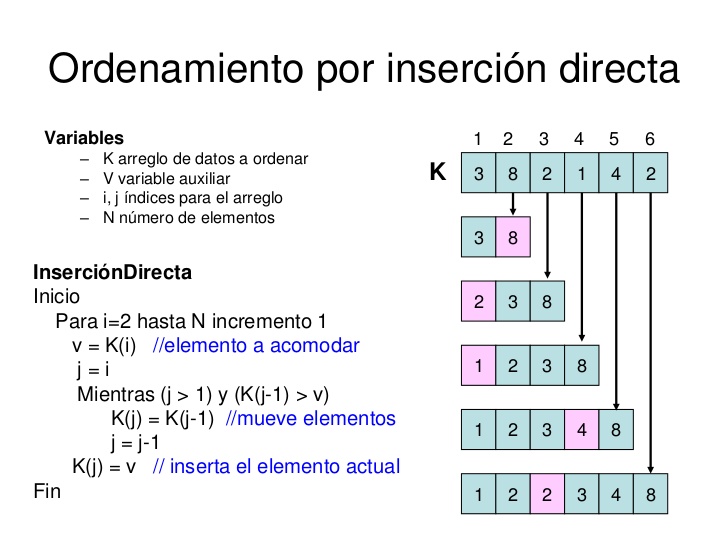
Aún vemos que nuestra lista no está totalmente ordenada de menor a mayor, entonces de nuevo continuamos comparando el primer elementó con su vecino hasta que terminemos la lista, este proceso debe de ser repetitivo tantas veces sea necesario para que nuestra lista quede ordenada.



* **INSERCION**

El **ordenamiento por inserción** es una manera muy natural de ordenar para un ser humano, y puede usarse fácilmente para ordenar un mazo de cartas numeradas en forma arbitraria. Requiere **O(n²)** operaciones para ordenar una lista de **n** elementos.

Inicialmente se tiene un solo elemento, que obviamente es un conjunto ordenado. Después, cuando hay ***k*** elementos ordenados de menor a mayor, se toma el elemento ***k+1*** y se compara con todos los elementos ya ordenados, deteniéndose cuando se encuentra un elemento menor (todos los elementos mayores han sido desplazados una posición a la derecha) o cuando ya no se encuentran elementos (todos los elementos fueron desplazados y este es el más pequeño). En este punto se *inserta* el elemento ***k+1*** debiendo desplazarse los demás elementos.

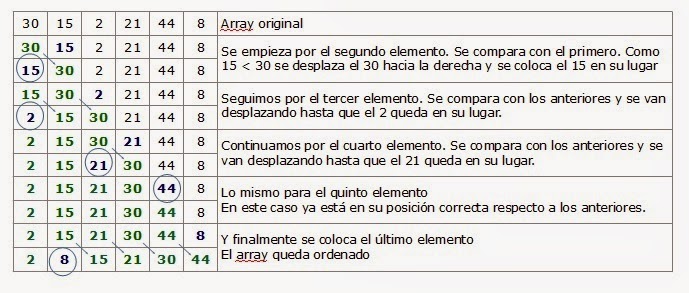


Ejemplo:

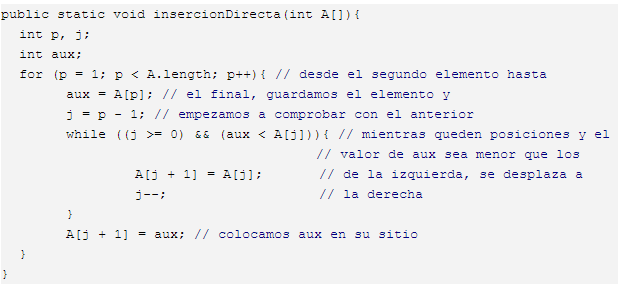
El método de ordenación por inserción directa consiste en recorrer todo el array comenzando desde el segundo elemento hasta el final. Para cada elemento, se trata de colocarlo en el lugar correcto entre todos los elementos anteriores a él o sea entre los elementos a su izquierda en el array.

Dada una posición actual p, el algoritmo se basa en que los elementos A[0], A[1], ..., A[p-1] ya están ordenados.

De forma gráfica el proceso que sigue el método de inserción directa es el siguiente:



El método de ordenamiento por inserción directa en Java es el siguiente:



**En el peor de los casos, el tiempo de ejecución en O(n2).**

**En el mejor caso** (cuando el array ya estaba ordenado), el tiempo de ejecución de este método de ordenamiento es **O(n).**

El caso medio dependerá de cómo están inicialmente distribuidos los elementos. Cuanto más ordenada esté inicialmente más se acerca a O(n) y cuanto más desordenada, más se acerca a O(n2).

El peor caso el método de inserción directa es igual que en los métodos de burbuja y selección, pero el mejor caso podemos tener ahorros en tiempo de ejecución.

* **QUICKSORT**

El **ordenamiento rápido** es un algoritmo creado por el científico británico en computación C.A.R Hoare basado en la técnica de divide y vencerás que permite, en promedio, ordenar *n* elementos en un tiempo proporcional a *n* log *n*.

El método de ordenación Quicksort fue desarrollado por Hoare en el año 1960. Es el algoritmo de ordenación más rápido.

Se basa en la técnica **divide y vencerás**, que consiste en ir subdividiendo el array en arrays más pequeños, y ordenar éstos. Para hacer esta división, se toma un valor del array como **pivote**, y se mueven todos los elementos menores que este pivote a su izquierda, y los mayores a su derecha. A continuación se aplica el mismo método a cada una de las dos partes en las que queda dividido el array.

Después de elegir el pivote se realizan dos búsquedas:

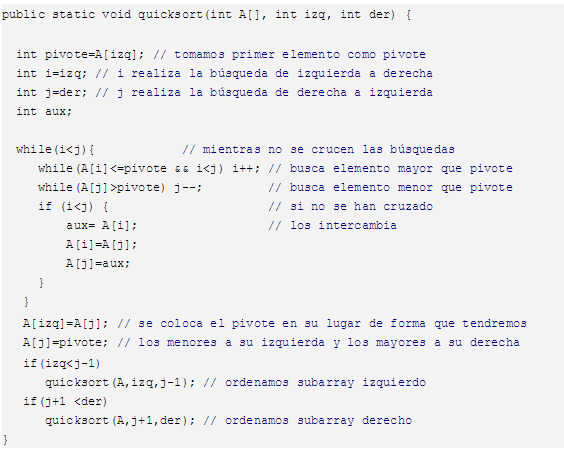
* Una de izquierda a derecha, buscando un elemento mayor que el pivote
* Otra de derecha a izquierda, buscando un elemento menor que el pivote.

Cuando se han encontrado los dos elementos anteriores, se intercambian, y se sigue realizando la búsqueda hasta que las dos búsquedas se encuentran.

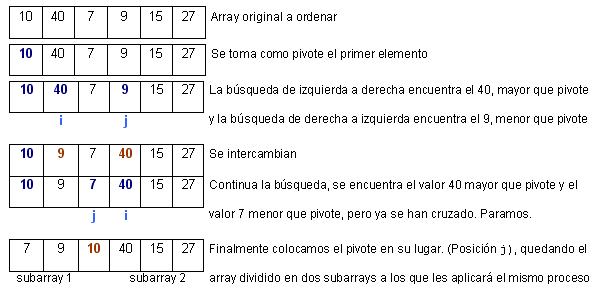
La implementación del método de ordenación Quicksort es claramente recursiva.

Suponiendo que tomamos como pivote el primer elemento, el método Java Quicksort que implementa este algoritmo de ordenación para ordenar un array de enteros se presenta a continuación. Los parámetros izq y der son el primer y último elemento del array a tratar en cada momento.

El método ordena un array A d eenteros desde la posición *izq* hasta la posición *der.* En la primera llamada recibirá los valores izq = 0, der = ELEMENTOS-1.



De forma gráfica el proceso sería el siguiente:



La elección del pivote determinará la eficiencia de este algoritmo ya que determina la partición del array. Si consideramos que el array está desordenado, podemos elegir el primer elemento y el algoritmo funcionaría de forma eficiente. Pero si el array está casi ordenado, elegir el primer elemento como pivote sería una mala solución ya que obtendríamos un subarray muy pequeño y otro muy grande. Por la misma razón, elegir el último elemento del array como pivote también es una mala idea. Pretendemos conseguir que el tamaño de los subarrays sea lo más parecido posible.

Una alternativa a elegir el primer elemento es elegir como pivote un elemento al azar de entre todos los del array.

Otra estrategia es calcular la mediana de los valores de la izquierda, centro y derecha del vector.

Por ejemplo para el vector: 9 8 1 6 10 2 3, se calcula la mediana de los elementos que ocupan el primer lugar, el último y el centro o sea 9 3 6. La mediana es 6 que determinaría las particiones {1 3 2} {6} {8 10 9}.

**En el peor caso, cuando el pivote es el elemento menor del array el tiempo de ejecución del método Quicksort es O(n2).**

**En general el tiempo medio de ejecución del Quicksort es O(n log n).**