
# Estructuras de Datos IIS262

Profesor: Patricio Galeas

# CAPÍTULO 6 :
# Ordenamiento Avanzado

# Introducción

- Ya vimos los algoritmos simples de ordenamiento.
- Estos son simples de implementar, pero muy lentos
- Algunos algoritmos avanzados de ordenamiento: Shellsort y Quicksort.
- Ambos son más rápidos que los algoritmos simples.
- No requieren espacio extra para operar.
- Quicksort es el más rápido de todos los algoritmos de ordenamiento general.

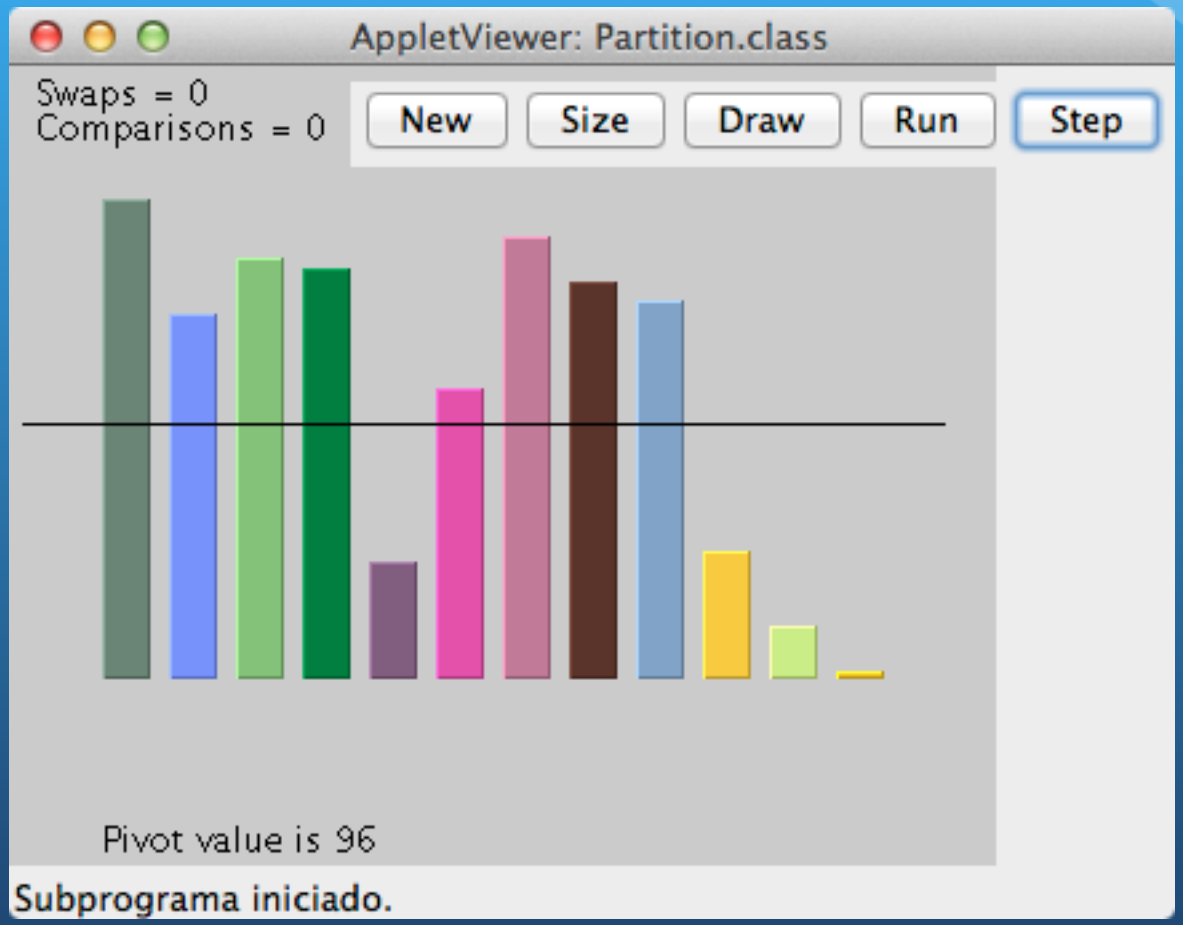
# Quicksort


http://www.youtube.com/watch?v=ywWBy6J5gz8

# Particionamiento (Quicksort)

- El particionamiento es el mecanismo que subyace al algoritmo Quicksort.
- Es un mecanismo interesante, así que lo veremos por separado.
- Particionar significa dividir datos en 2 grupos:
  - Grupo 1 : con datos mayores a cierto valor.
  - Grupo 2 : con datos menores a cierto valor.

# Particionamiento (Quicksort)




Partition

applet

# Particionamiento (Quicksort)

- Revisemos un ejemplo en Java

```
public int partitionIt(int left, int right, long pivot)
   {
   int leftPtr = left - 1;           // right of first elem
   int rightPtr = right + 1;         // left of pivot
   while(true)
      {
      while(leftPtr < right &&       // find bigger item
            theArray[++leftPtr] < pivot)
         ;  // (nop)

      while(rightPtr > left &&       // find smaller item
            theArray[--rightPtr] > pivot)
         ;  // (nop)
      if(leftPtr >= rightPtr)        // if pointers cross,
         break;                      //    partition done
      else                           // not crossed, so
         swap(leftPtr, rightPtr);    //    swap elements
      }  // end while(true)
   return leftPtr;                   // return partition
   }  // end partitionIt()
```

Partition.java

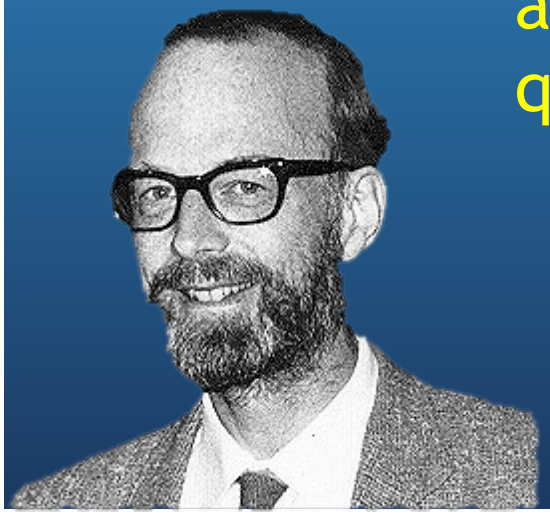
code

# Eficiencia del Particionamiento

- El algoritmo de particionamiento corre en un tiempo O(N).
- Las comparaciones y permutaciones son proporcionales al número de ítems.
- El numero de comparaciones es independiente a como estan organizados los items.
- El número de permutaciones si depende de cómo estan organizados los items. Si los items estan invertidos (peor caso), necesita N/2 permutaciones.
- Luego, hay menos permutaciones que comparaciones, pero ambas son proporcionales a N => O(N).

# Quicksort

- Descubierto por, Charles A. Hoare, en 1962.
- Por su rapidez, es actualmente el más popular de los algoritmos de búsqueda.
- Básicamente opera "particionando" el arreglo en dos y aplica recursivamente quicksort a los dos sub-arreglos.

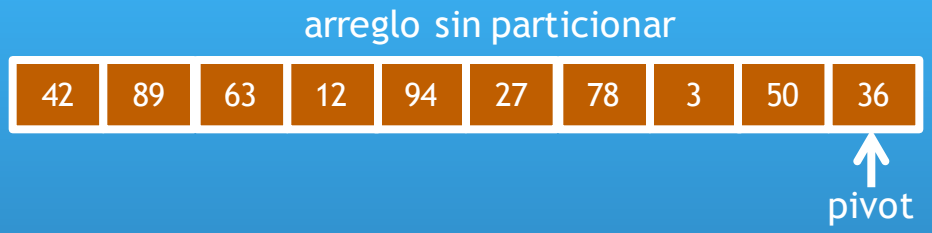


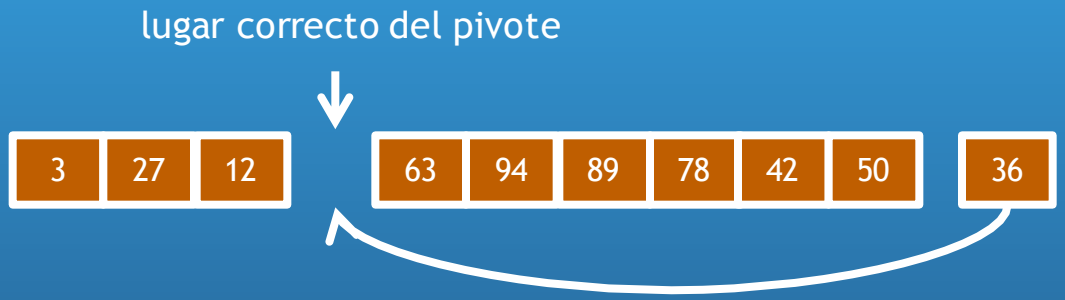
# El algoritmo Quicksort

```
public void recQuickSort(int left, int right)
   {
   if(right-left <= 0)        // if size is 1,
       return;                //    it's already sorted
   else                       // size is 2 or larger
      {
                                          // partition range
      int partition = partitionIt(left, right);
      recQuickSort(left, partition-1);   // sort left side
      recQuickSort(partition+1, right);  // sort right side
      }
   }
```

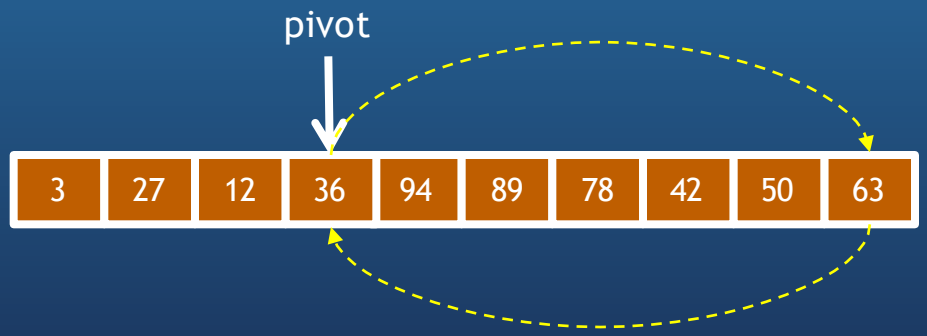

1. Dividir el arreglo o sub-arreglo un grupo izquierdo (valores menores) y derecho (valores mayores).
2. Se llama a si mismo para ordenar el grupo izquierdo.
3. Se llama a si mismo para ordenar el grupo derecho.

# Quicksort: eligiendo el pivote



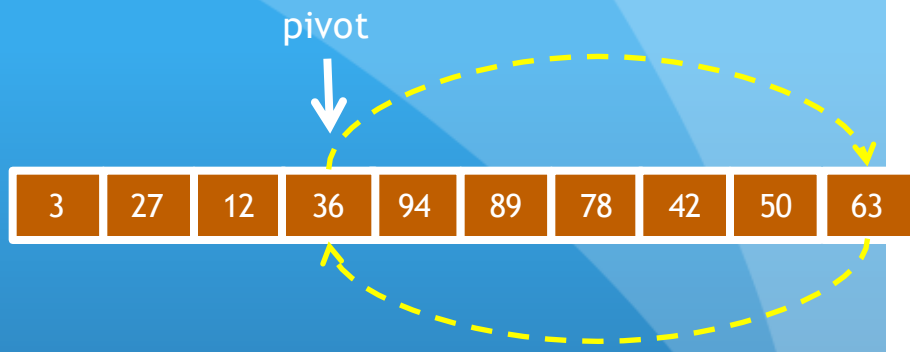

Mover los elementos es muy costoso, por eso se permuta el pivote, por el ítem más a la izquierda del arreglo de la derecha (63).




1. El pivote puede ser el valor de algún elemento del arreglo.
2. Entonces , el pivote se puede elegir en forma mas o menos aleatoria.
3. Por simplicidad se elige el último elemento de la derecha.
4. Después de la partición el pivote es insertado entre los arreglos del lado izquierdo y derecho, y será su posición final.

# Quicksort: eligiendo el pivote

Hay que modificar levemente recQuickSort() y partitionIt().




```
public void recQuickSort(int left, int right)
   {
   if(right-left <= 0)              // if size <= 1,
       return;                      //    already sorted
   else                             // size is 2 or larger
      {
      long pivot = theArray[right];      // rightmost item
                                         // partition range
      int partition = partitionIt(left, right, pivot);
      recQuickSort(left, partition-1);   // sort left side
      recQuickSort(partition+1, right);  // sort right side
      }
   }  // end recQuickSort()
```

```
while(leftPtr < right &&          //
      theArray[++leftPtr] < pivot)
   ;  // (nop)
```

- Aquí se elimina una condición, que mejora el rendimiento del algoritmo.
- Si se toma el pivote de otra posición, no se puede implementar esta mejora

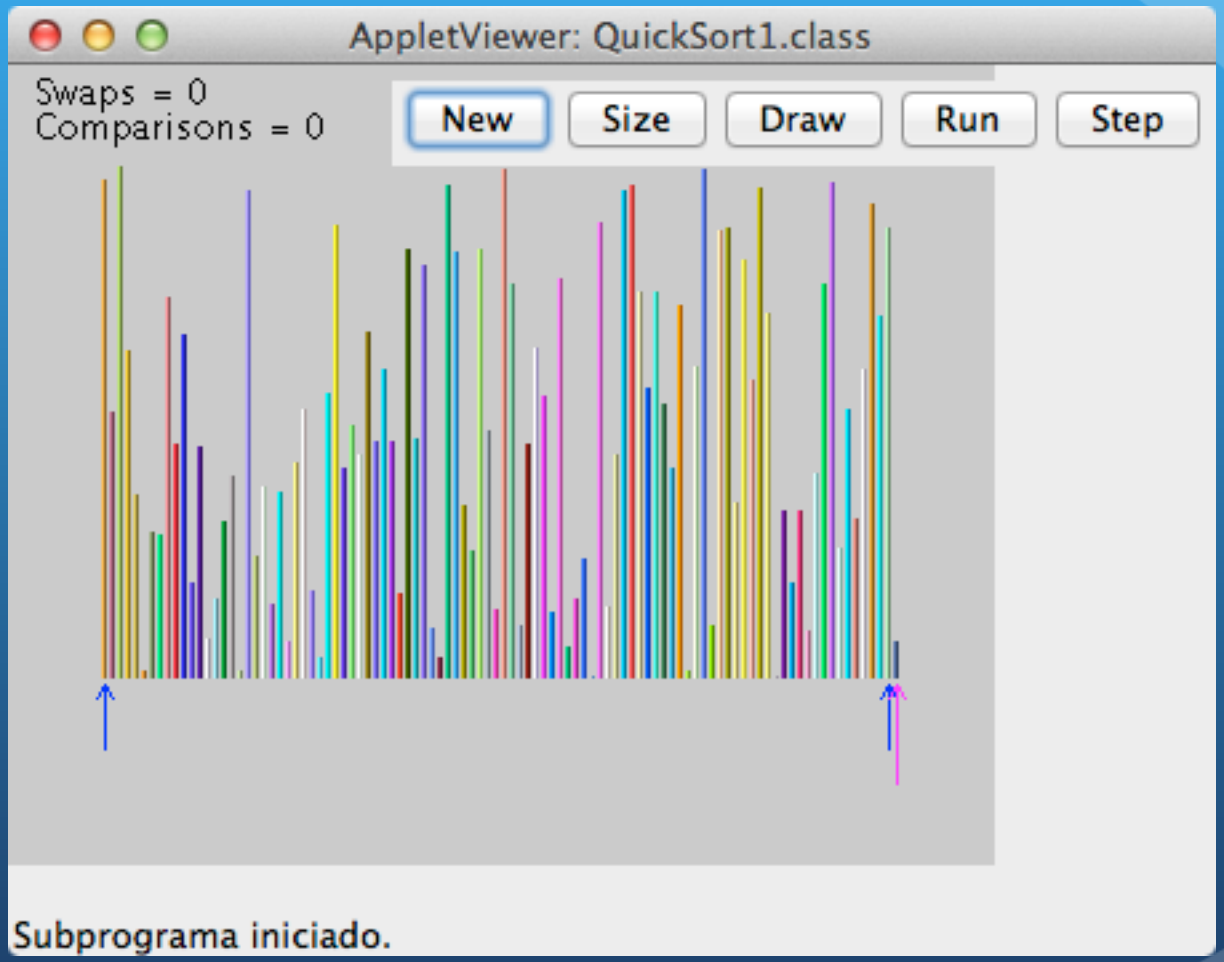
```
public int partitionIt(int left, int right, long pivot)
   {
   int leftPtr = left-1;            // left    (after ++)
   int rightPtr = right;            // right-1 (after --)
   while(true)
      {                             // find bigger item
      while( theArray[++leftPtr] < pivot )
         ;  // (nop)
                                    // find smaller item
      while(rightPtr > 0 && theArray[--rightPtr] > pivot)
         ;  // (nop)

      if(leftPtr >= rightPtr)       // if pointers cross,
         break;                     //    partition done
      else                          // not crossed, so
         swap(leftPtr, rightPtr);   //    swap elements
      }  // end while(true)
   swap(leftPtr, right);            // restore pivot
   return leftPtr;                  // return pivot location
   }  // end partitionIt()
```

# Quicksort: el código final

Quicksort1.java

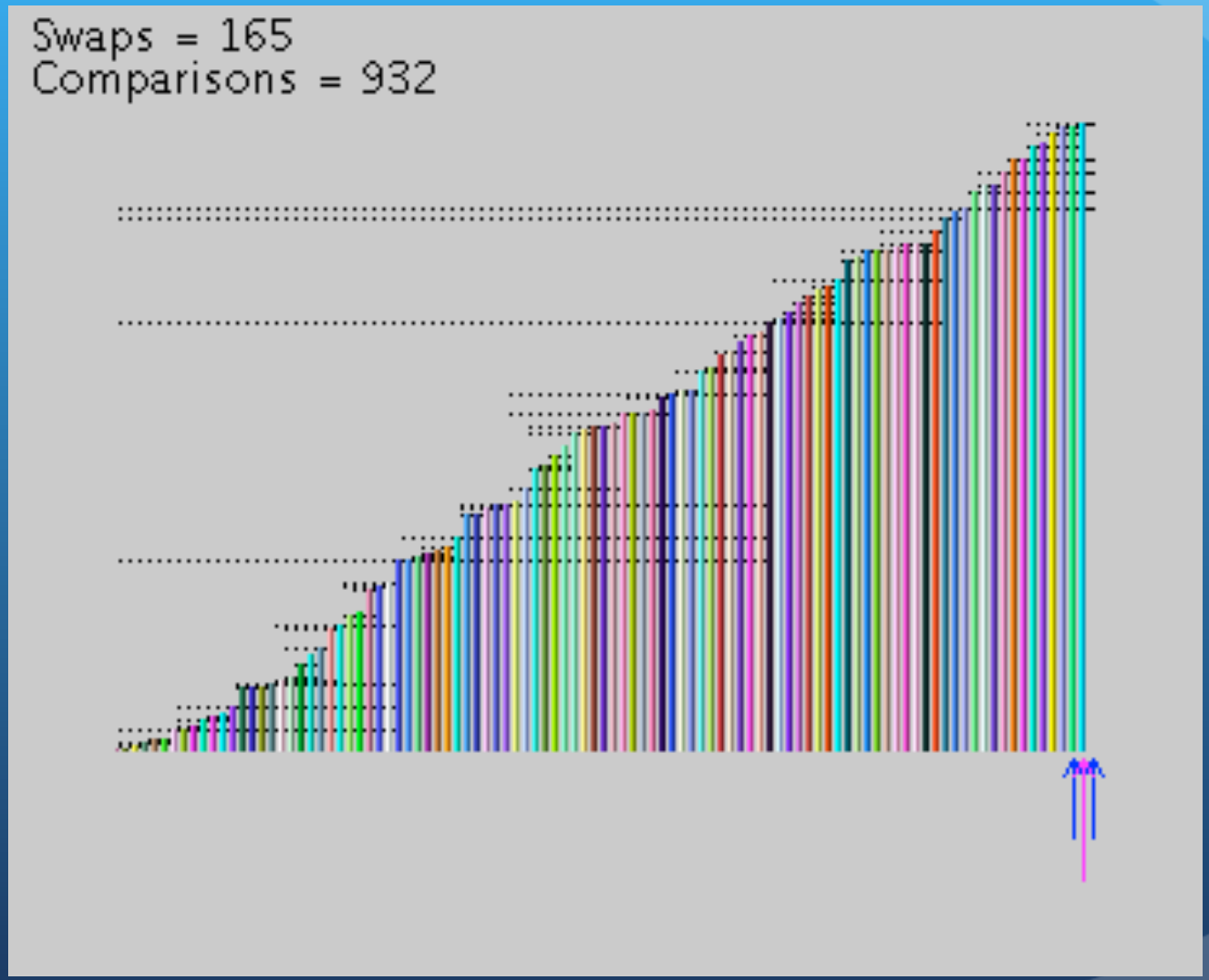
code

# Quicksort: el applet con 100 ítems




applet

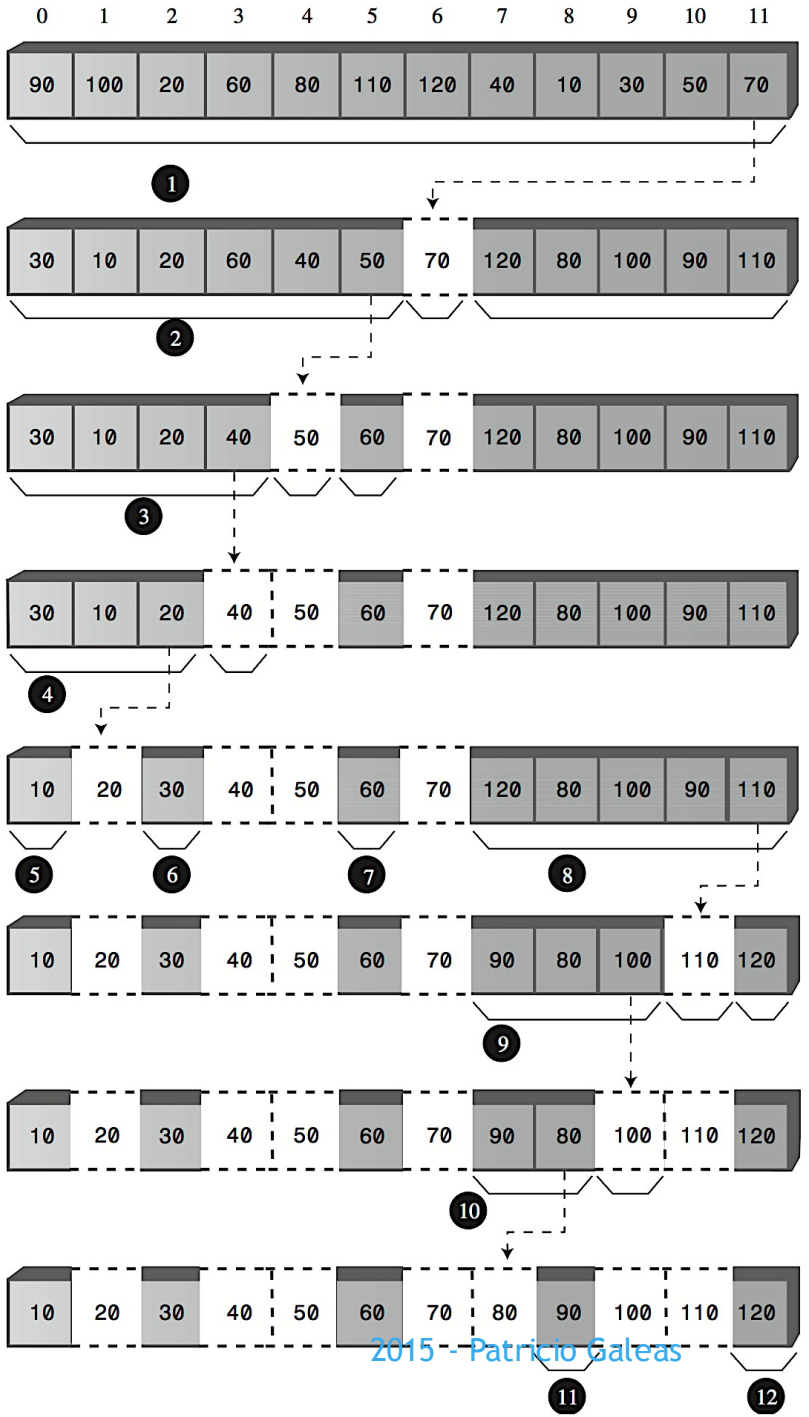
# Quicksort: el applet con 100 ítems




applet

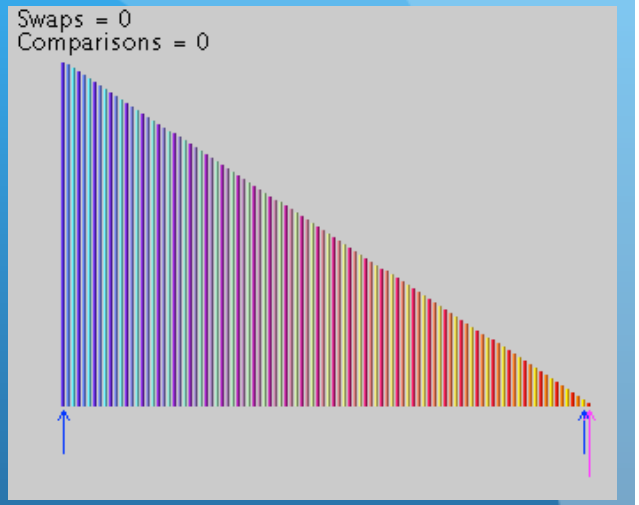
# Quicksort: en detalle

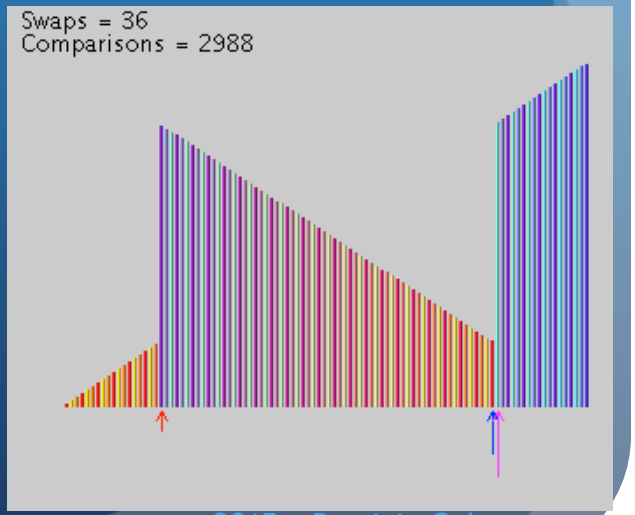
1. Los paréntesis horizontales indican que arreglo es particionado en cada paso.
2. Los circulos muestran el orden en el que la particiones son creadas.
3. La permutación del pivote a su posición final es demarcada con las flechas segmentadas.
4. La posición “final” del pivote esta marcada por una celda segmentada.
5. Los parentesis horizontales en las celdas unitarias (5, 6, 7, 11 y 12) son casos base del algoritmo (retornan inmediatamente).
6. A veces, como en los pasos 4 y 10, el pivote termina en su posición original, al lado derecho del arreglo que esta siendo ordenado. En esta situación, sólo hay un solo sub-arreglo para ser ordenado: el de la izquierda al pivote. No hay sub-arreglo al lado derecho del pivote.
7. El orden en que las particiones son creadas corresponde al número del paso (círculos) y no al nivel de recursión. Es decir, no todas las particiones del primer de recursión se realizan antes de las del segundo nivel, etc.



# Quicksort: caso mas desfavorable

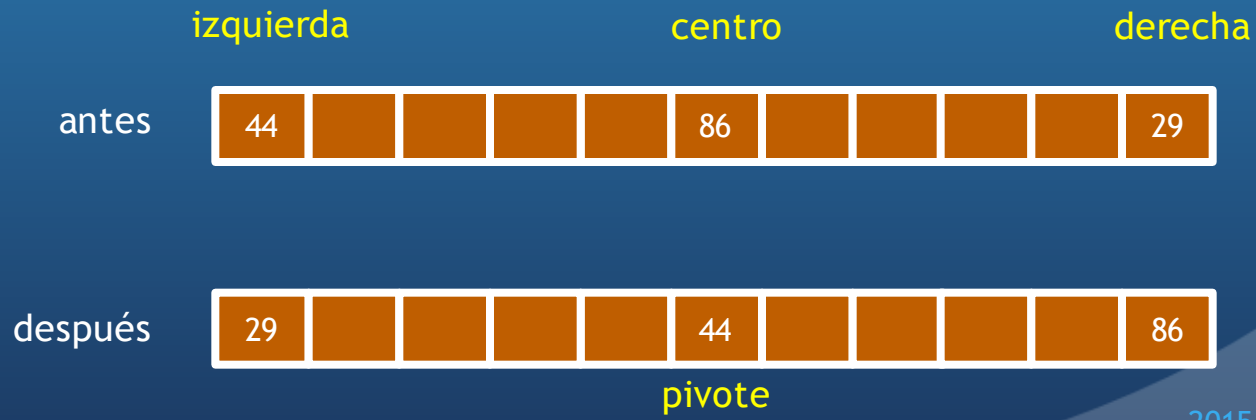
1. Quicksort a 100 ítems ordenados inversamente es bastante lento.
2. El problema es la elección del pivote
3. El pivote "ideal" es la Mediana de los ítems, ya que esta genera dos arreglos de igual tamaño (Quicksort óptimo)
4. Cuando un arreglo es mas grande que el otro implica que el primero debe ser particionado más veces.
5. El peor caso ocurre cuando la división se traduce en un arreglo de un elemento y el otro de (N-1) elementos (Paso 3 y 9 de lámina anterior).
6. Si esta división entre 1 y (N-1) ítems ocurre en todos las particiones, entonces cada elemento requerirá un paso de partición.
7. Esto es el caso cuando los ítems están ordenados inversamente. En cada sub-arreglo, el pivote es el menor item, luego cada partición resulta en (N-1) elementos en un sub-arreglo y el pivote en el otro sub-arreglo.
8. ¿Se puede mejorar la forma de seleccionar el pivote?



# Quicksort: Mejoras para elegir el pivote

- Hay varios intentos por mejorar el mecanismo de selección del pivote.
- Ya dijimos que la Mediana es el pivote "ideal", pero su cálculo es muy costoso.
- Una solución: Mediana del primer, medio y último ítem: "Mediana de Tres".



# Quicksort: Beneficios de la Mediana de Tres

Aparte de poder seleccionar el pivote de manera más eficiente, la mediana de tres tiene otros beneficios.

1. Como los valores left, center y right "ya están ordenados", no es necesario incluirlos en el proceso de partición (left y right ya se encuentran en el lado correcto).

2. La condición rightPtr>left del 2° while no es mas necesaria, ya que el valor del puntero rightPtr en el borde izquierdo es siempre menor al pivote (valor central de la Mediana).

Luego, la Mediana de Tres no sólo evita el caso más desfavorable $O(N^2)$, (seleccionando un pivote más adecuado), sino que también mejora el rendimiento de los ciclos (while) internos de la partición.

```
public int partitionIt(int left, int right, long pivot)
   {
   int leftPtr = left-1;           // left    (after ++)
   int rightPtr = right;           // right-1 (after --)
   System.out.println("left:"+left+" right:"+right);
   while(true)
      {                            // find bigger item
      while( theArray[++leftPtr] < pivot )
         ;  // (nop)
                                   // find smaller item
      while(rightPtr > left && theArray[--rightPtr] > pivot)
         ;  // (nop)

      if(leftPtr >= rightPtr)      // if pointers cross,
         break;                    //    partition done
      else                         // not crossed, so
         swap(leftPtr, rightPtr);  //    swap elements
      }  // end while(true)
   swap(leftPtr, right);           // restore pivot
   return leftPtr;                 // return pivot location
   }  // end partitionIt()
```

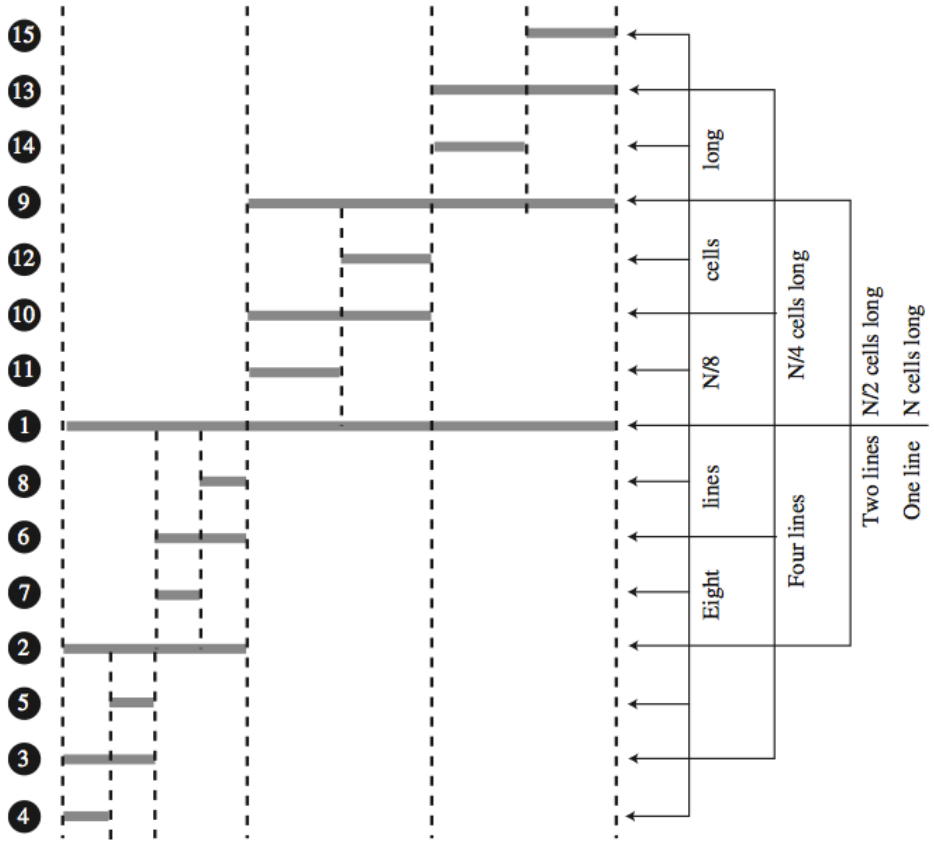
```
public int partitionIt(int left, int right, long pivot)
   {
   int leftPtr = left;             // right of first elem
   int rightPtr = right - 1;       // left of pivot

   while(true)
      {
      while( theArray[++leftPtr] < pivot )  // find bigger
         ;                                  //    (nop)
      while( theArray[--rightPtr] > pivot ) // find smaller
```

# Eficiencia de Quicksort

- Dijimos que Quicksort opera en tiempo O(N*logN).
- Esto es relativamente cierto para los algoritmos "Dividir para Conquistar" en donde un método recursivo divide un rango de items en dos grupos y luego se llama a si mismo para resolver cada grupo: $O(N*\log_2 N)$.
- Uno se puede hacerse una idea de la validez de esta estimación revisando las lineas punteadas que deja el Applet cuando ordena 100 elementos.
- Cada línea punteada representa la partición de un arreglo o sub-arreglo: donde los punteros leftScan y rightScan se mueven en direcciones opuestas comparando y permutando items.
- Además vimos que una partición opera en tiempo O(N).
- Es decir, el largo de estas lineas es proporcional al tiempo de ejecución de Quicksort.
- Pero ¿Cómo medir esas líneas? … con el numero de ítems asociados.

# Eficiencia de Quicksort

- Las líneas sólidas = líneas punteadas del Applet
- N/8, N/4, N/2, etc., es el número promedio de elementos en cada línea.
- Los círculos representan el orden en que las líneas son creadas.
- Si partimos con 100 ítems, obtenemos la serie de largos de línea: 100, 50 ,25, 12, 6, 3, 1.
- Que sumados dan un largo total de 652 items.




$100*\log_2 100 = 665$

| Recursión | Paso | Largo prom. Línea (items) | Nro. de líneas | Largo total |
|---|---|---|---|---|
| 1 | 1 | 100 | 1 | 100 |
| 2 | 2,9 | 50 | 2 | 100 |
| 3 | 3,6,10,13 | 25 | 4 | 100 |
| 4 | 4, 5, 7, 11, 12, 14,15 | 12 | 8 | 96 |
| 5 | No mostrado | 6 | 16 | 96 |
| 6 | No mostrado | 3 | 32 | 94 |
| 7 | No mostrado | 1 | 64 | 64 |
| | | | TOTAL | 652 |

# Resumen

- Shellsort aplica el ordenamiento por inserción con elementos mas espaciados, disminuyendo el espacio en cada iteración.
- Una secuencia de números, llamada "secuencia de intervalo", es usada para determinar los intervalos de ordenamiento en Shellsort.
- La "secuencia de intervalo" puede ser calculada con la expresión recursiva : h=3*h+1.
- El orden de Shellsort es $O(N*(logN)^2)$, pero es difícil de calcular. Este es mucho más rápido que los algoritmos $O(N^2)$ como Inserción, pero más lento que Quicksort que corre a O(N*logN).
- Particionar un arreglo es dividirlo en dos sub-arreglos. Uno con ítems menores a un determinado valor y el otro sub-arreglo con ítems mayores al valor definido.
- El pivote es un valor que termina en que grupo irá un ítem durante el proceso de partición. Ítems menores que el pivote can al grupo de la izquierda y los ítems mayores al grupo de la derecha.

# Resumen

- En el algoritmo de particionamiento, dos punteros (controlados con dos while), escanean el arreglo en direcciones opuestas para encontrar ítems que deben ser permutados.
- Cuando un puntero encuentra un elemento que necesita ser permutados, interrumpe el loop while.
- Cuando ambos loops (while) se interrumpen, los ítems son permutados.
- Cuando ambos loops (while) se interrumpen y los punteros se cruzan la partición esta completa.
- El particionamiento opera en tiempo lineal O(N). Haciendo (N+1) o (N+2) comparaciones y (N/2) permutaciones.
- El particionamiento requiere una condición extra en los while internos, para evitar que los punteros salgan de los límites del arreglo.

# Resumen

- Quicksort particiona un arreglo y luego se llama dos veces recursivamente para ordenar los sub-arreglos resultantes de la partición.
- Sub-arreglos de un-elemento están automáticamente ordenados (caso base de Quicksort)
- El valor del pivote en Quicksort es el valor de un item específico del arreglo.
- Una versión simple de Quicksort, utiliza el último elemento de la derecha como pivote.
- Durante la partición, el pivote permanece al lado derecho y no es considerado. Después es permutado entre los dos sub-arreglos, tomando su posición final (ordenado).
- En versión simple de Quicksort, el tiempo para items inversamente ordenados en $O(N^2)$.
- En una versión más avanzada se utiliza la mediana-de-tres para elegir el pivote. Lo cual elimina el problema de $O(N^2)$ en datos inversamente ordenados.
- Mediana-de-tres permite eliminar las condiciónes de "fin de arreglo" en los while.
- Quicksort opera en tiempo O(N*logN), excepto en arreglos inversamente ordenados.
- Arreglos menores a cierto tamaño, pueden ser ordenados con un método distinto a Quicksort.

# Experimentos

- Encontrar que pasa cuando usamos el applet Partition con 100 ítems inversamente ordenados. ¿esta el resultado casi ordenado?
- Modifique shellSort.java, para que imprima el contenido del arreglo después de completar cada ordenamiento (n-sort). El arreglo debe ser lo suficientemente pequeño para que su contenido se pueda ver en una línea. Analice los pasos intermedio para determinar si el algoritmo opera en la forma que Ud. tenía en mente.
- Modifique shellSort.java y quickSort3.java, para que ordenen arreglos grandes y compare sus velocidades. También compare sus velocidades con los algoritmos de ordenamiento simple.