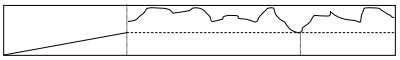
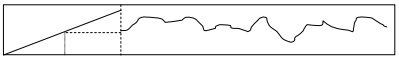
Problema. Ordenar un arreglo de n objetos (comparables)
`void ordenar(Comparable[]x,int n)`
Algoritmo de selección y reemplazo ($O(n^2)$)



ip: índice del primero (de los desordenados)
im: índice del menor (de los desordenados)

Algoritmo de inserción ($O(n^2)$)
¿idea? Cada vez se inserta un elemento entre los ya ordenados.



i: índice de elemento que se debe insertar entre los ya ordenados
j: índice de inserción (de primer elemento > x[i])

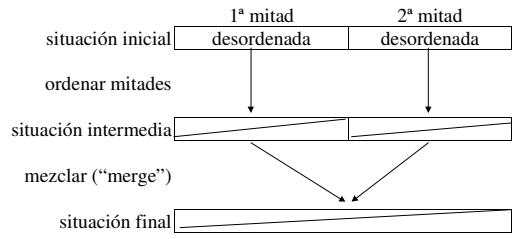
Algoritmo de la burbuja ($O(n^2)$)
¿Idea? En cada una de las pasadas "hacer subir la burbuja (el mayor) al último lugar".

algoritmo
repetir n-1 veces (para los n, n-1, ..., 2 primeros elementos)
- recorrer todos los elementos (salvo el último)
  - comparar elemento con siguiente
  - intercambiar si están fuera de orden, es decir, si elemento es mayor que siguiente

ejemplo: String[]x={"D","C","B","E","A"};

recorrido	1ª comp	2ª comp	3ª comp	4ª comp	x
1°	**DC**BEA	C**DB**EA	CB**DE**A	CBD**EA**	CBDA**E**
2°	**CB**DA	B**CD**A	BC**DA**		BCA**D**E
3°	**BC**A	B**CA**			BA**C**DE
4°(n-1)	**BA**				A**B**CDE

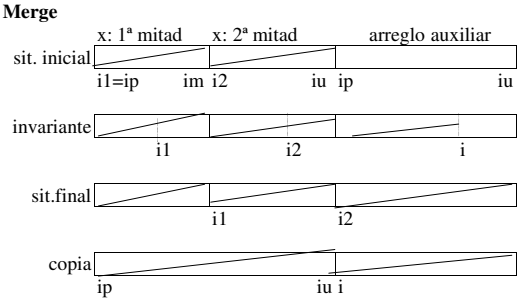
MergeSort: algoritmo $O(n\log_2 n)$




```
static public
void mergesort(Comparable[]x,int ip,int iu)
{
   if(ip>=iu) return;     //caso base

   int im=(ip+iu)/2;      //índice de mitad

   mergesort(x,ip,im);    //ordenar 1ª mitad

   mergesort(x,im+1,iu); //ordenar 2ª mitad

   merge(x,ip,im,iu);     //mezclar mitades
}
```

invocación
`mergesort(a,0,a.length-1);`

Merge




```
void merge(Comparable[]x,int ip,int im,int iu){
  //arreglo auxiliar e indices
  Comparable[]a=new Comparable[iu+1];
  int i1=ip,    //indice de primera mitad
      i2=im+1, //indice de segunda mitad
      i=ip;    //indice de arreglo auxiliar
  //repetir hasta terminar una mitad
  while(i1<=im  &&  i2<=iu)
     //seleccionar y copiar el más pequeño
     a[i++]=(x[i1].compareTo(x[i2])<0) ?
            x[i1++] : x[i2++];
  //traspasar restantes 1ª mitad (si quedan)
  while(i1<=im) a[i++]=x[i1++];
  //traspasar restantes 2ª mitad (si quedan)
  while(i2<=iu) a[i++]=x[i2++];
  //copiar arreglo auxiliar en original
  for(int i=ip; i<=iu; ++i) x[i]=a[i];
}
```

Versión 2: una iteración

```
void merge(Comparable[]x,int ip,int im,int iu)
{
  //recorrer mitades y arreglo auxiliar
  Comparable[]a=new Comparable[iu+1];
  for(int i=ip, i1=ip, i2=im+1; i<=iu; ++i)
    //si izq<der (o se terminó 2ª mitad)
    if(i1<=im
    &&(i2>iu || x[i1].compareTo(x[i2])<0))
      a[i]=x[i1++];
    else
      a[i]=x[i2++];
  //copiar arreglo auxiliar
  for(int i=ip; i<=iu; ++i)
    x[i]=a[i];
}
```

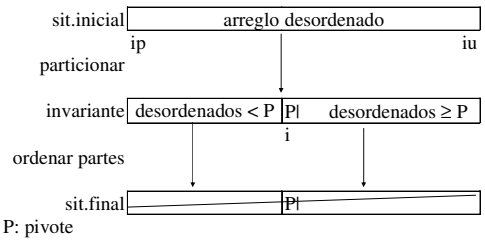
Análisis informal

- Mergesort se invoca recursivamente $2\log_2 n$ veces (tantas veces como se puede dividir el arreglo por la mitad)
- Mergesort invoca a merge una vez
- Merge realiza O(n) comparaciones (en cada recorrido)
- En total: $O(n\log_2 n)$ comparaciones

¿Espacio?

- Mergesort requiere espacio adicional para los n elementos
- Se puede ahorrar espacio adicional a costa del tiempo de ejecución (propuesto)

QuickSort (Hoare, 1962): $O(n\log_2 n)$ sin arreglo auxiliar

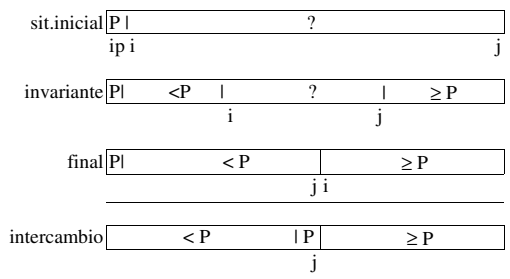


P: pivote

- elemento del arreglo que se utiliza para separar (distribuir) los otros elementos en menores y mayores o iguales
- puede elegirse cualquiera (por ejemplo el primero)
- las particiones resultan de distintos largos
- ¿$\log_2 n$? en promedio particiones de largos iguales (≈mergesort)

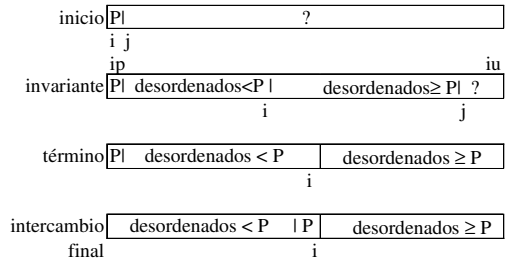
```
static public
void quicksort(Comparable[]x,int ip,int iu)
{
  //caso base(1 elemento)
  if(ip>=iu) return;

  //particionar (y obtener indice de pivote)
  int i=particionar(x,ip,iu);

  //ordenar 1ª parte
  quicksort(x,ip,i-1);

  //ordenar 2ª parte
  quicksort(x,i+1,iu);
}
```

Particionamiento original de Hoare: algoritmo O(n)




```
int particionar(Comparable[]x,int ip,int iu){
  //elegir pivote (por ejemplo: el primero)
  Comparable pivote=x[ip];
  //repetir hasta que indices se superen
  int i=ip+1,j=iu;
  while(i<=j){
    //avanzar índice de menores
    while(i<=j && x[i].compareTo(pivote)<0)
      ++i;
    //retroceder índice de mayores o iguales
    while(i<=j && x[j].compareTo(pivote)>=0)
      --j;
    //intercambiar menor con mayor (si corr)
    if(i<=j) intercambiar(x,i++,j--);
  }
  //pivote a su posición final
  x[ip]=x[j]; x[j]=pivote; return j;
}
```

Problema: programar el particionamiento de acuerdo a las reglas indicadas en las siguientes figuras




```
int particionar(Comparable[]x,int ip,int iu){
  //elegir pivote (ej. Primero)
  Comparable pivote=x[ip];
  //inicializar índices
  int i=ip;//ultimo de los menores que pivote
  int j=ip+1;//primero de los desconocidos
  //recorrer todos los desconocidos
  while(j<=iu){
    //si < pivote,intercambiar con 1ºde mayores
    if(x[j].compareTo(pivote)<0){
      i=i+1; intercambiar(x,i,j);
    }
    j=j+1;//avanzar indice desconocidos
  }
  //pivote a su posición final
  x[ip]=x[i]; x[i]=pivote;
  //devolver indice de pivote
  return i;
}
```

```
int particionar(Comparable[]x,int ip,int iu)
{
  //elegir pivote
  Comparable pivote = x[ip];
  //inicializar índice de último de menores
  int i=ip;
  //recorrer elementos
  for(int j=ip+1; j<=iu; ++j)
    //si menor que pivote juntarlo con menores
    if( x[j].compareTo(pivote) < 0 )
      intercambiar(x,++i,j);
  //pivote a su posición final
  intercambiar(x,ip,i);
  return i;
}
```

Propuesto. Ordenar un archivo
(suponiendo que sólo N líneas caben en memoria)