Análisis y diseño de software

dit
UPM

Tema 2: Algoritmos /ordenación /java

José A. Mañas

http://jungla.dit.upm.es/~pepe/doc/adsw/index.html

23.2.2018

ETSIT UPM

objetivos

- algoritmos de ordenación
  - conocer nombres propios
  - conocer complejidad de cada uno
- aserciones (predicados que deben cumplirse)
- invariantes [de bucle]
- razonamientos de corrección (inducción)
- razonamientos de complejidad

referencias

- En la web
  - sorting
- dada una lista de datos, ordenarla

```
public abstract class StringSorter {
    public abstract void sort(String[] datos);
}
```

```
System.out.println(Arrays.toString(datos));
metodo.sort(datos);
System.out.println(Arrays.toString(datos));
```

```
[r, a, m, v, o, f, d, u, x, d]

[a, d, d, f, m, o, r, u, v, x]
```

auxiliar: intercambiador

```
/**
 * Intercambio.
 * Lo que hay en la posicion i pasa a la posicion j.
 * Lo que hay en la posicion j pasa a la posicion i.
 */
void swap(String[] datos, int i, int j) {
    if (i == j)
        return;
    String si = datos[i];
    String sj = datos[j];
    datos[i] = sj;
    datos[j] = si;
}
```

auxiliar: predicado

```
/**
   * Predicado.
   *
   * @param datos Strings.
   * @return TRUE si los datos estan ordenados entre [a, z).
   */
  boolean sorted(String[] datos, int a, int z) {
     for (int i = a; i + 1 < z; i++)
        if (datos[i].compareTo(datos[i + 1]) > 0)
           return false;
     return true;
  }
```

smoke test

```
// smoke test
public static void main(String[] args) {
   Random random = new Random();
   String[] datos = new String[10];
   for (int i = 0; i < datos.length; i++) {
      char ch = (char) ('a' + random.nextInt(26));
      datos[i] = String.valueOf(ch);
   }
   System.out.println(Arrays.toString(datos));

   StringSorter metodo = new ...();
   metodo.sort(datos);
   System.out.println(Arrays.toString(datos));

   if (!metodo.sorted(datos, 0, datos.length))
      System.out.println("ERROR");
}
```

http://www.guru99.com/smoke-sanity-testing.html

aserciones - invariantes

```
public class My {

   public static void assertTrue(boolean cond) {
      if (!cond)
         throw new IllegalStateException(cond + " != true");
   }

}
```

algoritmos

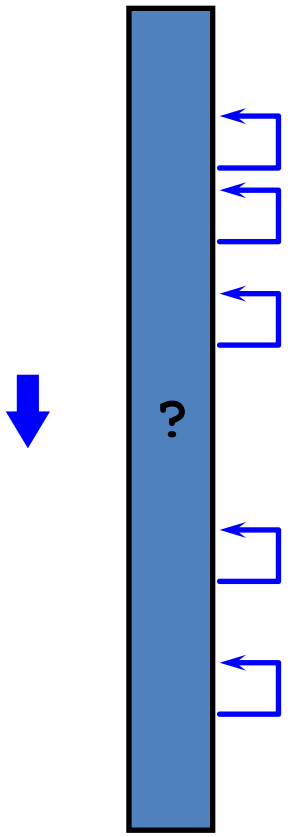
- burbuja
- selección
- inserción
- quicksort

- shell sort
- heap sort
- merge sort
- tim sort
- ...

burbuja

- en cada pasada, miramos pares adyacentes;
  si están desordenados, los ordenamos

```
@Override
  public void sort(String[] datos) {
     boolean changed;
     do {
        changed = false;
        for (int i = 0; i < datos.length - 1; i++) {
           if (datos[i].compareTo(datos[i + 1]) > 0) {
              swap(datos, i, i + 1);
              changed = true;
           }
        }
     } while (changed);
  }
```

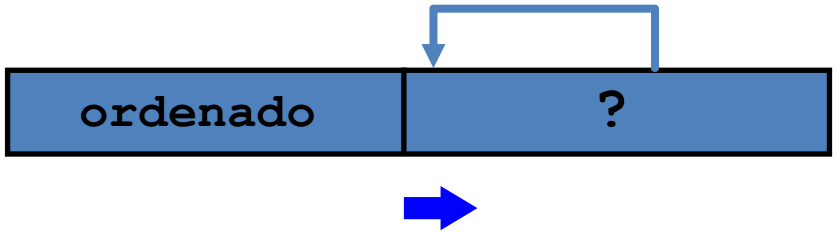


caso mejor / caso peor

- caso mejor
  - datos ordenados: 1 pasada
  - O(n)
- caso peor
  - datos al revés: N pasadas
  - recurrencia: $T(n) = cn + T(n-1) \rightarrow O(n^2)$

selección

- en cada pasada, elegimos el menor de los datos que quedan y lo metemos al final de los datos procesados




```
for (int i = 0; i < datos.length - 1; i++) {
   ...  ...  ...
   My.assertTrue(sorted(datos, 0, i + 1));
}
```

selección

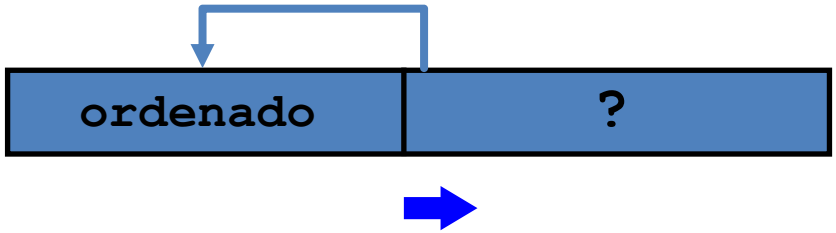
```
@Override
public void sort(String[] datos) {
    for (int i = 0; i < datos.length - 1; i++) {
        int j = minimo(datos, i, datos.length);
        swap(datos, i, j);
        My.assertTrue(sorted(datos, 0, i + 1));
    }
}

private int minimo(String[] datos, int a, int z) {
    int min = a;
    for (int i = a; i < z; i++) {
        if (datos[i].compareTo(datos[min]) < 0)
            min = i;
    }
    return min;
}
```

complejidad

- contando
  - la primera vez elegimos entre N
  - la segunda, entre N-1
  - …
  - $T(n) = n + n-1 + n-2 + \ldots + 1 \rightarrow O(n^2)$
- función recurrente
  - $T(n) = O(n) + T(n-1) \rightarrow O(n^2)$

inserción

- en cada pasada, elegimos el primero de los datos que quedan y lo metemos en su sitio en los datos procesados




```
for (int i = 0; i < datos.length - 1; i++) {
   ...  ...  ...
   My.assertTrue(sorted(datos, 0, i + 1));
}
```

inserción

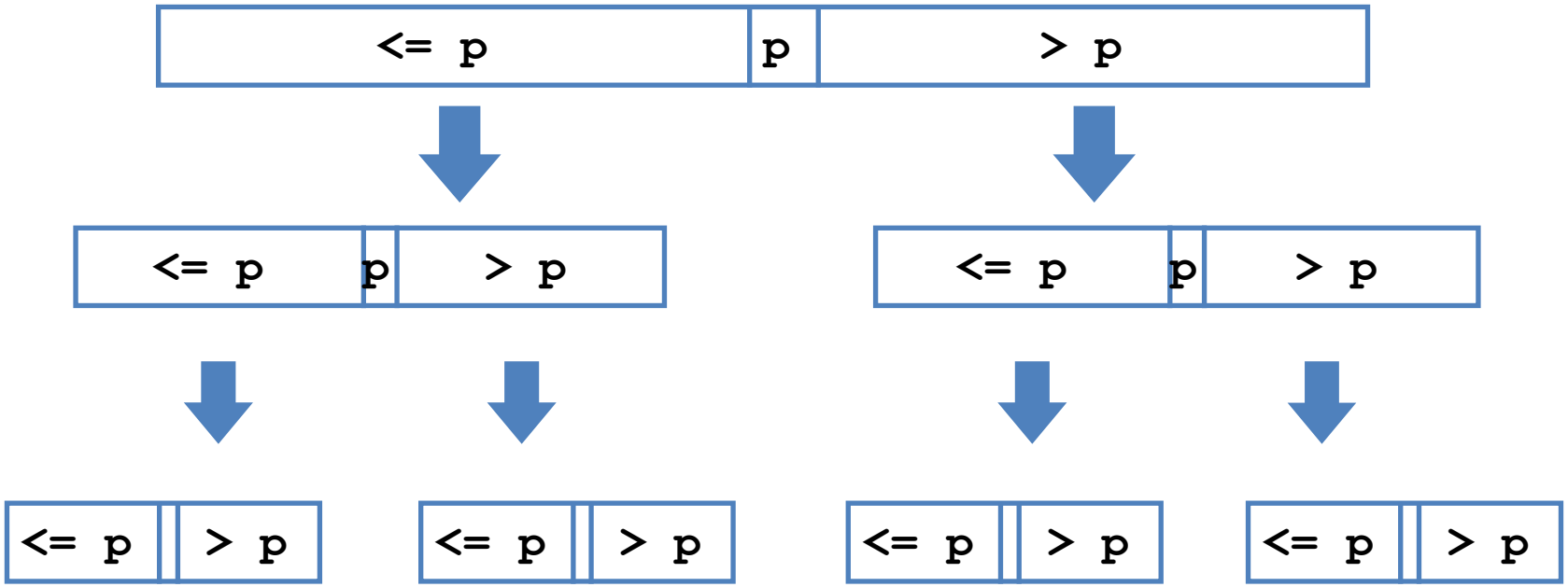
```
@Override
public void sort(String[] datos) {
    for (int i = 1; i < datos.length; i++) {
        inserta(datos, i, datos[i]);
        My.assertTrue(sorted(datos, 0, i + 1));
    }
}

// inserta v en datos, entre 0 y z-1.
private void inserta(String[] datos, int z, String v) {
    int j = z;
    while (0 < j && v.compareTo(datos[j - 1]) < 0)
        j--;
    System.arraycopy(datos, j, datos, j + 1, z - j);
    datos[j] = v;
}
```

caso mejor / caso peor

- caso mejor: ordenados
  - un recorrido, sin intercambios
  - O(n)
- caso peor: al revés
  - hay que recorrer todo para inserter
  - $T(n) = O(n) + T(n-1) \rightarrow O(n^2)$

quickSort

- divide y vencerás: ordenar subvectores



se ordenan los casos triviales, y queda ordenado el conjunto

invariante

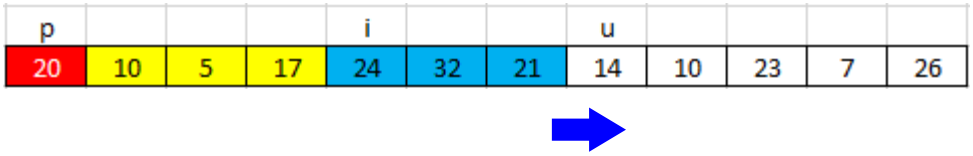
```
private void sort(String[] datos, int a, int z) {
    if (z <= a)
        return;
    .... .... ....
    sort(datos, a, x);
    sort(datos, x, z);
    My.assertTrue(sorted(datos, a, z);
}
```

quicksort / tony hoare

```
private void sort(String[] datos, int a, int z) {
      if (z <= a)
         return;
      String pivot = datos[(a + z) / 2];
      int izq = a;
      int der = z - 1;
      while (izq <= der) {
         while (datos[izq].compareTo(pivot) < 0)
            izq++;
         while (datos[der].compareTo(pivot) > 0)
            der--;
         if (izq <= der)
            swap(datos, izq++, der--);
      }
      sort(datos, a, der + 1);
      sort(datos, izq, z);
   }
```

quicksort / jon bentley

- Usamos D[a] como pivote:
  - a la izquierda de i están los elementos que son menores que D[a].
  - Entre i y u están los elementos que son mayores de D[a].
  - A la derecha de u están los elementos que aún no hemos considerado.




```
int i = a;
for (int u = a + 1; u < z; u++) {
    ...  ...  ...
    for (int j = a; j < i; j++)
      My.assertTrue(datos[j].compareTo(pivot) <= 0);
   for (int j = i + 1; j < z; j++)
      My.assertTrue(datos[i].compareTo(pivot) >= 0);}
```

partición /jon bentley

```
private void sort(String[] datos, int a, int z) {
      if (z <= a)
         return;
      String pivot = datos[a];
      int i = a;
      for (int u = a + 1; u < z; u++) {
         if (datos[u].compareTo(pivot)) < 0) {
            i++;
            swap(datos, i, u);
         }
      }
      swap(datos, a, i);
      // invariantes
      sort(datos, a, i);
      sort(datos, i + 1, z);
   }
```

John Bentley.
Programming Pearls.
2nd ed. 1999.

caso mejor / caso peor

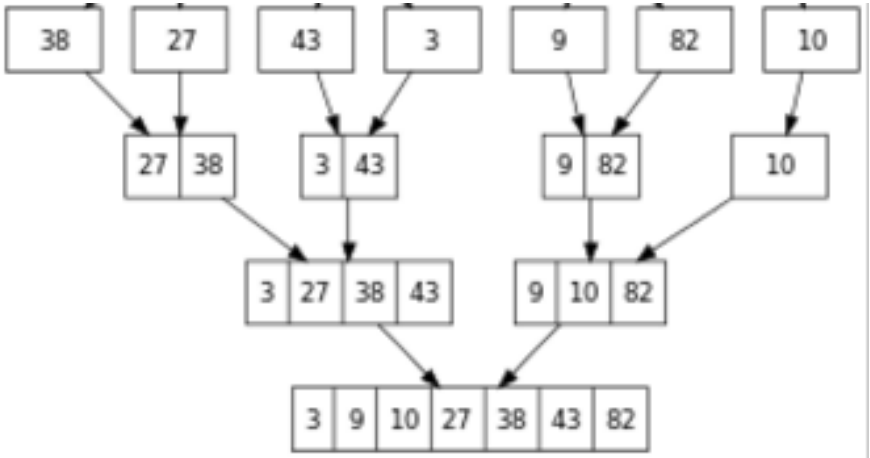
- caso mejor
  - pivote en el centro → bipartición
  - $T(n) = O(n) + 2\,T(n/2) \rightarrow O(n \log n)$
- caso peor
  - pivote en un extremo → árbol degenerado
  - $T(n) = O(n) + T(n-1) \rightarrow O(n^2)$

método híbrido

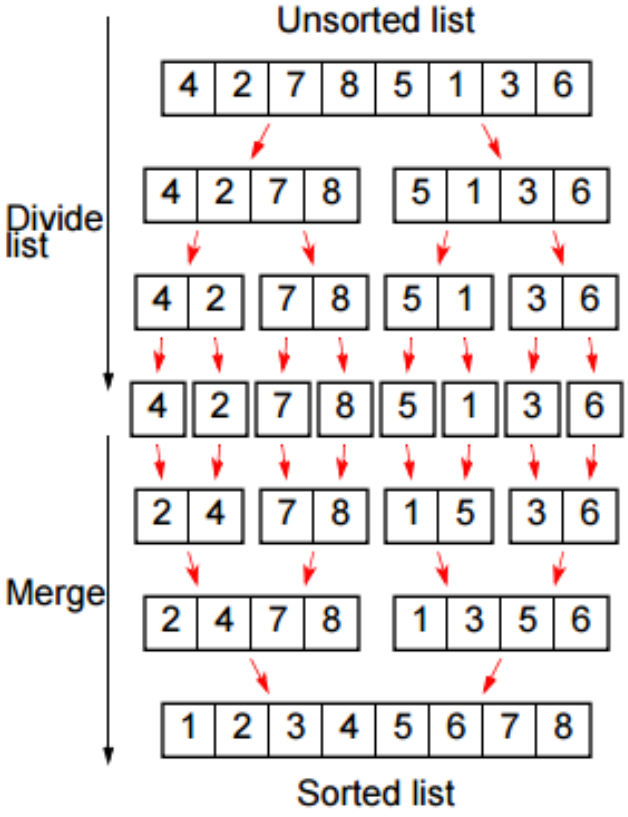
- fragmentamos hasta un cierto umbral
  - suele ser un número bajo, ~10
- por debajo, aplicamos inserción

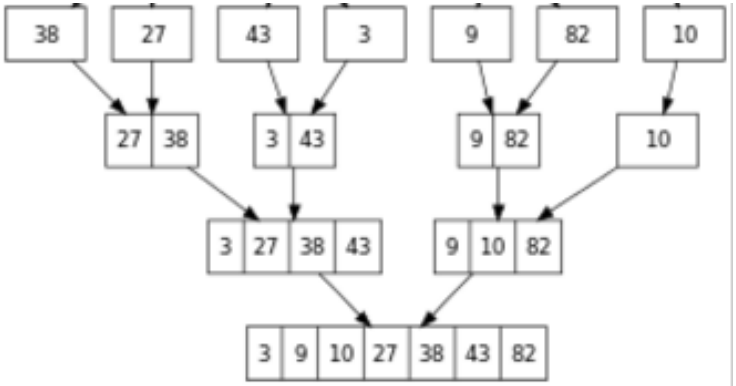
mergesort

- tomamos datos de 2 en 2 y los ordenamos
- tomamos datos de 4 en 4 y los ordenamos
- ...
- tomamos datos de 2^k en 2^k y los ordenamos



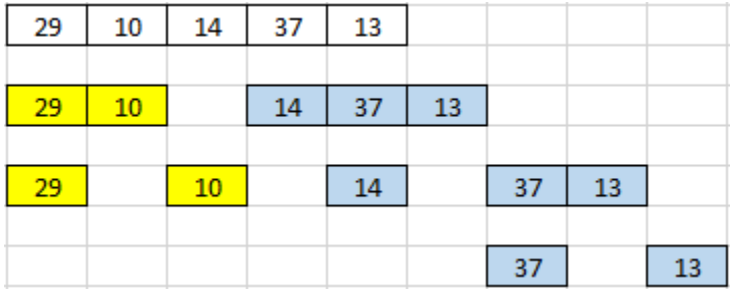
recursivo - iterativo





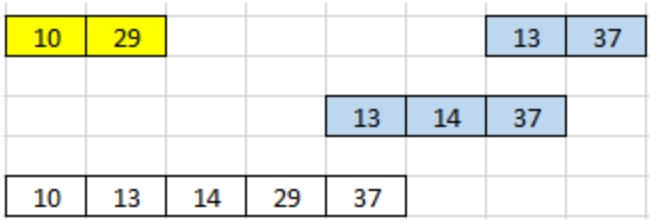
mergesort for dummies

```
@Override
public void sort(String[] datos) {
    List<String> list = new ArrayList<>();
    Collections.addAll(list, datos);
    sort(list);
    list.toArray(datos);
}

private void sort(List<String> list) {
    if (list.size() < 2)
        return;
    int m = list.size() / 2;
    List<String> izq = new ArrayList<>(list.subList(0, m));
    List<String> der = new ArrayList<>(list.subList(m, list.size()));

    sort(izq);
    sort(der);
```



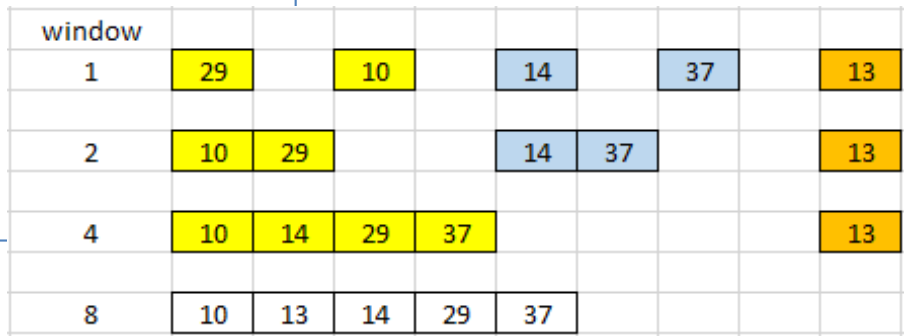
mergesort for dummies

```
    sort(izq);
    sort(der);

    list.clear();
    while (izq.size() > 0 && der.size() > 0) {
        String si = izq.get(0);
        String sd = der.get(0);
        if (si.compareTo(sd) < 0)
            list.add(izq.remove(0));
        else
            list.add(der.remove(0));
    }
    list.addAll(izq);
    list.addAll(der);
}
```



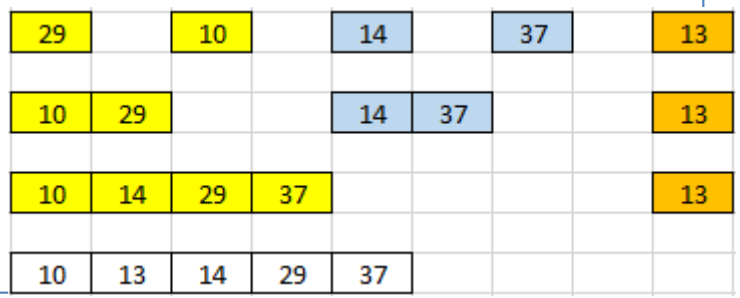
mergesort (iterativo)

```
@Override
  public void sort(String[] datos) {
     bottomUpSort(datos, new String[datos.length]);
  }

  private void bottomUpSort(String[] datos, String[] aux) {
     int n = datos.length;
     for (int window = 1; window < n; window *= 2) {
        for (int i = 0; i < n; i += 2 * window) {
           int iIzq = i;
           int iDer = Math.min(i + window, n);
           int iEnd = Math.min(i + 2 * window, n);
           bottomUpMerge(datos, iIzq, iDer, iEnd, aux);
        }
        System.arraycopy(aux, 0, datos, 0, n);
     }
  }
```



mergesort (iterativo)

```
private void bottomUpMerge(String[] datos, int iIzq, int iDer, int iEnd, String[] aux) {
    My.assertTrue(sorted(datos, iIzq, iDer));
    My.assertTrue(sorted(datos, iDer, iEnd));
    int i0 = iIzq;
    int i1 = iDer;
    int dst= iIzq;
    while (i0 < iDer && i1 < iEnd) {
        if (OpMeter.compareTo(datos[i0], datos[i1]) <= 0)
            aux[dst++] = datos[i0++];
        else
            aux[dst++] = datos[i1++];
    }
    while (i0 < iDer)
        aux[dst++] = datos[i0++];
    while (i1 < iEnd)
        aux[dst++] = datos[i1++];
    My.assertTrue(sorted(aux, iIzq, iEnd));
}
```



complejidad

- a costa de tener un array auxiliary que consume N datos en RAM ...
- caso mejor = caso peor = caso medio
- $T(n) = 2\,T(n/2) + O(n) \rightarrow O(n \log n)$

Análisis de complejidad

método	mejor	*medio*	peor
burbuja	O(n)	$O(n^2)$	$O(n^2)$
selección	$O(n^2)$	$O(n^2)$	$O(n^2)$
inserción	O(n)	$O(n^2)$	$O(n^2)$
quicksort	O(n log n)	$O(n \log n)$	$O(n^2)$
mergesort	O(n log n)	$O(n \log n)$	O(n log n)

Array Sorting Algorithms

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	Θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	Θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	Θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	Θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	Θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	Θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	Θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	Θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	Θ(n log(n))	O(n log(n))	O(n)

http://bigocheatsheet.com/

estabilidad

- algoritmo de ordenación que respeta el orden relative de elementos con la misma clave

(4, 0), (4, 1), (1, 7)

estables

- burbuja
- inserción
- mergesort

(1, 7), (4, 0), (4, 1)

no estables

- selección
- quicksort
- heapsort

(1, 7), (4, 1), (4, 0)

resumiendo ...

- hay muchos algoritmos de ordenación
  - hemos visto algunos muy utilizados
- burbuja: rápido y bueno para arrays ya ordenados
- selección nunca es buena
- inserción es bueno para arrays casi ordenados
- quicksort suele funcionar en O(n log n)
  - el híbrido mejora notoriamente
- mergesort gasta el doble de memoria pero garantiza O(n log n)