Unidad: Estructuras de datos básicas

Búsqueda y ordenamiento

Introducción a la Programación

Noviembre de 2025

Caso historico: Censo 1890 y tarjetas perforadas (Hollerith)

Contexto. El censo de 1880 tardo muchos años en procesarse manualmente. En 1890, Herman Hollerith introduce tarjetas perforadas y una maquina tabuladora para capturar, clasificar y resumir datos.
Innovación. Inspira su diseño en el telar de Jacquard: perforaciones representan atributos (edad, sexo, estado civil). La maquina lee eléctricamente las perforaciones y actualiza contadores.
Funcionamiento. Cada tarjeta representa a una persona; clasificar primero por campos (orden) hace que consultar y resumir luego sea rapido (menos pasos).
Eficiencia. El uso de tabuladoras acelera de forma drastica el conteo y el analisis del censo de 1890 frente al proceso manual previo.
Impacto. La comercializacion de estas maquinas lleva a la Tabulating Machine Company, que mas tarde formaria parte de IBM.
Fuentes:

- IBM Heritage: The punched card tabulator. ibm.com/history/punched-card-tabulator
- U.S. Census Bureau: Herman Hollerith and mechanical tabulation. census.gov/.../january-2016.html

Caso historico: Catalogos en fichas y Sistema Dewey

Contexto. A finales del siglo XIX y durante gran parte del XX, las bibliotecas usaron catalogos en fichas para localizar libros. Melvil Dewey propuso en 1876 un esquema de clasificacion numerica por materias.
Idea clave. *Ordenar* primero (por materia/autor/titulo) convierte la *busqueda* fisica en un recorrido corto y sistematico: de miles de libros a unas pocas fichas.
Funcionamiento. Cada ficha describe una obra (autor, titulo, materia). Las gavetas mantienen fichas *ordenadas* (alfabetico o numerico). El usuario recorre el segmento correcto y encuentra rapido.
Eficiencia. Sin orden previo, buscar un libro especifico implicaria revisar anaqueles uno por uno. Con orden, la consulta se reduce a muy pocos pasos.
Impacto. El catalogo en fichas estandarizo la organizacion del conocimiento en bibliotecas y anticipo principios modernos de indices y metadatos para busqueda.
Fuentes:

- Library of Congress: Card Catalogs (historia y transicion). loc.gov/item/prn-13-041
- OCLC: Dewey Decimal Classification overview. oclc.org/en/dewey/resources.html

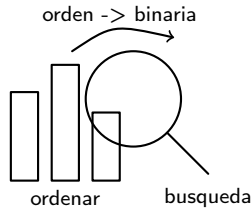
Caso reciente: Clasificacion de paquetes en e-commerce

Contexto. En redes de comercio electronico, millones de paquetes se mueven cada dia. La rapidez depende de *ordenar* primero por destino para luego *buscar/encaminar* con muy pocos pasos.
Funcionamiento. En centros de clasificacion, los paquetes pasan por escaner; el sistema *lee la etiqueta* y *asigna una chuta* o linea segun su destino (por ejemplo, codigo postal). Asi se consolidan cargas por ruta.
Idea clave. El orden previo (por zona/codigo) convierte la posterior asignacion de rutas en recorridos cortos y mecanicos: menos comparaciones y menos movimientos.
Eficiencia. Al llegar a la estacion de entrega, el personal ya recibe lotes ordenados por recorrido, lo que acelera la carga del vehiculo y reduce re-trabajos.
Impacto. Esta capa de ordenamiento es una pieza critica para cumplir ventanas de entrega y escalar operaciones.
Fuentes:

- Amazon (operacion y centros de clasificacion). aboutamazon.com/.../how-do-amazon-packages-get-delivered
- UPS Worldport (clasificacion a gran escala). howstuffworks.com/ups.htm

Sesion 3 — objetivos y agenda

- Conectar recorridos -> consultas de busqueda.
- Aplicar busqueda lineal y binaria (precondicion de orden).
- Diferenciar seleccion, insercion y burbuja (con corte).
- Reconocer estable vs no estable e in-place.
- Actividad en aula.



Mapa rapido: busquedas y ordenamientos

Búsquedas

- Lineal: sirve sin ordenar; primera ocurrencia.
- Binaria: requiere arreglo ordenado asc.

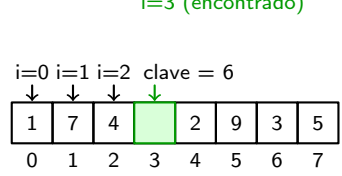
Ordenamientos básicos

- Seleccion: in-place; no siempre estable; simple.
- Insercion: in-place; estable; muy buen rendimiento en casi ordenado.
- Burbuja (con corte): in-place; estable; detecta ya-ordenado.

Busquedas

Busqueda lineal (idea y seguimiento)

- Recorre desde indice 0 hasta n-1.
- Sirve sin ordenar; devuelve primera ocurrencia.
- Se detiene al encontrar la clave.
- Si no encuentra, retorna -1.



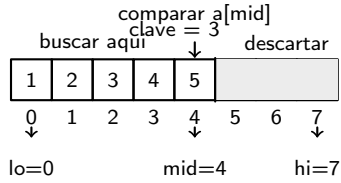
De recorrido a consulta de busqueda

Lineal (recorrido secuencial):

```
int buscarLineal(int[] a, int clave){
  for(int i=0;i<a.length;i++){
    if(a[i]==clave) return i;  // predicado satisface -> parar
  }
  return -1; // se agoto el recorrido
}
```

Busqueda binaria (idea y seguimiento)

- Requiere arreglo ordenado ascendente.
- Indices: lo, mid, hi.
- Si clave $<$ a[mid] -> mover hi $=$ mid - 1.
- Si clave $>$ a[mid] -> mover lo $=$ mid $+$ 1.
- Si a[mid] == clave -> retornar mid.



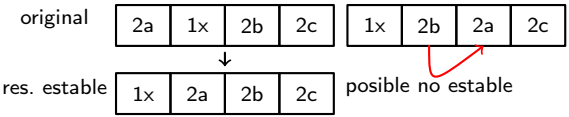
Consulta sobre un rango: busqueda binaria

```
int buscarBinaria(int[] a, int clave){
  int lo=0, hi=a.length-1;
  while(lo<=hi){
    int mid = lo+(hi-lo)/2;
    if(a[mid]==clave) return mid;       // exito
    if(clave < a[mid]) hi = mid-1;      // reducir a izquierda
    else lo = mid+1;                    // reducir a derecha
  }
  return -1; // sin exito
}
```

Ordenacion

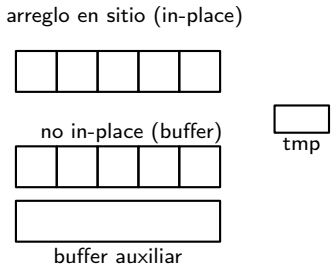
Propiedades: estabilidad e in-place

Estabilidad (orden relativo de duplicados)



- insercion y burbuja: **estables**
- seleccion: *no siempre* estable (puede cruzar 2a y 2b)

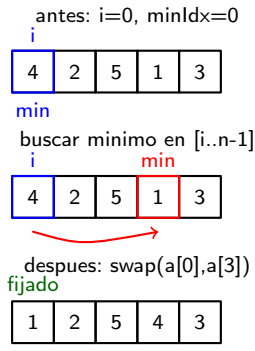
Uso de memoria (in-place)



- seleccion, insercion, burbuja: **in-place**
- algoritmos con arreglo extra: *no in-place*

Seleccion: seguimiento y minimos

- Para i de 0 a n-2: fijar i como inicio del segmento.
- Buscar el indice del minimo en [i..n-1] (minIdx).
- Si minIdx != i, hacer swap(a[i], a[minIdx]).
- Simple e in-place; *no siempre* estable.
- Util cuando n es pequeno o como base didactica.

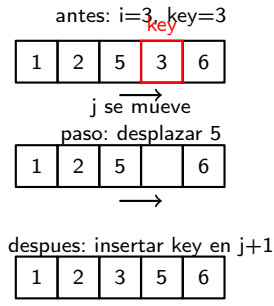


Seleccion: codigo minimo (in-place)

```
static void swap(int[] a, int i, int j){
  if(i==j) return;
  int t = a[i]; a[i] = a[j]; a[j] = t;
}
static void seleccion(int[] a){
  for(int i = 0; i < a.length - 1; i++){
    int min = i;
    for(int j = i + 1; j < a.length; j++){
      if(a[j] < a[min]) min = j;
    }
    swap(a, i, min);
  }
}
```

Insercion: seguimiento y desplazamientos

- i avanza de 1 a n-1; key = a[i]
- Mientras a[j] > key: desplazar a[j] a j+1 y decrementar j
- Insertar key en j+1
- Casi ordenado: pocos desplazamientos
- Estable e in-place

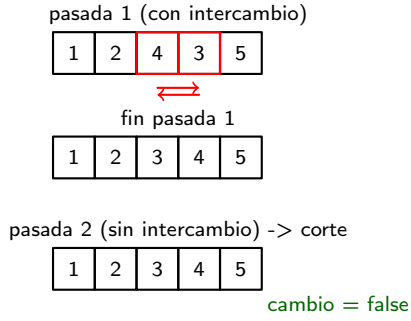


Insercion: codigo minimo

```
static void insercion(int[] a){
  for(int i = 1; i < a.length; i++){
    int key = a[i];
    int j = i - 1;
    while(j >= 0 && a[j] > key){
      a[j + 1] = a[j];
      j--;
    }
    a[j + 1] = key;
  }
}
```

Burbuja con corte: idea y corte temprano

- Comparar pares adyacentes y hacer swap si estan fuera de orden.
- En cada pasada, el mayor pendiente "burbujea" al final.
- **Corte temprano**: si en una pasada no hubo intercambios, la lista ya esta ordenada.
- Estable e in-place.



Burbuja: codigo minimo (con corte)

```
static void swap(int[] a, int i, int j){
  if(i==j) return;
  int t = a[i]; a[i] = a[j]; a[j] = t;
}
static void burbujaConCorte(int[] a){
  for(int i = 0; i < a.length - 1; i++){
    boolean cambio = false;
    for(int j = 0; j < a.length - 1 - i; j++){
      if(a[j] > a[j+1]){
        swap(a, j, j+1);
        cambio = true;
      }
    }
    if(!cambio) break; // ya esta ordenado
  }
}
```

Instrumentacion: proposito y uso

- **Que es**: agregar contadores simples al codigo para medir *que tanto hace* un algoritmo (sin cambiar su logica).
- **Que medimos aqui**: **comps** (comparaciones) y **swaps** (intercambios/desplazamientos).
- **Como lo haremos**: contadores **locales** dentro del metodo; se reinician al inicio y se imprimen al final.
- **Para que sirve**:
  - Comparar metodos (seleccion vs insercion vs burbuja) en casos reales.
  - Detectar *casi ordenado* (insercion hace pocos desplazamientos).
  - Ver efecto del *corte temprano* en burbuja.
- **Reglas practicas**:
  - No modificar la logica del algoritmo (solo contar).
  - Reiniciar contadores por prueba.
  - Probar varios casos: aleatorio, ordenado, reverso, con duplicados.

Instrumentacion simple (en el metodo)

```java
// Insercion con contadores locales
static void insercionContada(int[] a) {
  int comps = 0;   // comparaciones
  int swaps = 0;   // intercambios (desplazamientos)
  for (int i = 1; i < a.length; i++) {
    int key = a[i];
    int j = i - 1;
    // Contamos la comparacion a[j] > key en cada iteracion que evaluamos
    while (j >= 0) {
      comps++;                  // comparamos a[j] > key
      if (a[j] > key) {
        a[j + 1] = a[j];        // desplazamiento (lo contamos en swaps)
        swaps++;
        j--;
      } else { break; }// no hay mas desplazamientos para esta key
    }
    a[j + 1] = key;             // insercion final
  }
  System.out.println("comps=" + comps + ", swaps=" + swaps);
}
```

ejemplo de medición

```
static void print(int[] a){
  for(int i=0;i<a.length;i++){
    System.out.print(a[i] + (i+1<a.length?" ":"\n"));
  }
}
public static void main(String[] args){
  int[] casi = {1,2,4,3,5,6};
  int[] reverso = {6,5,4,3,2,1};
  System.out.println("Caso: casi ordenado");
  print(casi);
  insercionContada(casi);       // imprime comps y swaps
  print(casi);
  System.out.println("\nCaso: reverso");
  print(reverso);
  insercionContada(reverso);    // imprime comps y swaps
  print(reverso);
}
```

Decidir busqueda (reglas simples)

- **Arreglo no ordenado** o **pocas consultas**: usar **lineal**.
- **Arreglo ordenado** y **muchas consultas**: usar **binaria**.
- Si primero vas a **ordenar** para luego buscar varias veces: conviene **ordenar una vez** y despues **binaria**.
- Con **duplicados**:
  - si necesitas la *primera* posicion, tras encontrar, retrocede mientras sea igual;
  - si necesitas la *ultima*, avanza mientras sea igual.

Mini glosario

- **clave**: valor que se busca en el arreglo.
- **lo, mid, hi**: indices usados en busqueda binaria (inicio, medio, fin).
- **pasada**: recorrido parcial de un algoritmo de ordenamiento.
- **swap**: intercambio de dos posiciones del arreglo.
- **in-place**: trabaja sobre el mismo arreglo (sin arreglo extra).
- **estable**: mantiene el orden relativo de elementos iguales.
- **corte temprano**: terminar antes si ya no hay cambios.