Algoritmos Y Estructuras de Datos
Compendio: Sorting Algorithms (UT9)

Santiago Blanco

22-06-2025

Sorting

Insertion sort

→ En el i-ésimo recorrido se inserta el i-ésimo elemento en el lugar correcto entre los (i-1) elementos anteriores, los cuales fueron ordenados previamente.

Util para arreglos pequeños o casi ordenados. Funciona construyendo progresivamente una subsecuencia ordenada al insertar cada nuevo elemento en su lugar adecuado.

1. Se trabaja de izquierda a derecha
2. Se examina cada elemento y lo compara a los elementos de su izquierda
3. Inserto el elemento en la posición correcta del array

Se van formando particiones ordenadas y no ordenadas en el array durante el proceso.

- En el peor caso (arreglo en orden inverso), su complejidad es $O(N^2)$.
- En el mejor caso (arreglo ya ordenado), el tiempo es $O(N)$.
- El caso promedio también es $O(N^2)$.

Posición de la matriz	0	1	2	3	4	5
Estado inicial	8	5	9	2	6	3
Después de ordenar a[0..1]	5	8	9	2	6	3
Después de ordenar a[0..2]	5	8	9	2	6	3
Después de ordenar a[0..3]	2	5	8	9	6	3
Después de ordenar a[0..4]	2	5	6	8	9	3
Después de ordenar a[0..5]	2	3	5	6	8	9

Figura 8.3 Acción básica del algoritmo de ordenación por inserción (la parte sombreada está ordenada).

```
1 for i = 2 to n do
2   mover V[i] hacia la posición j <= i tal que
3   V[i].clave < V[k].clave para j<=k<i , y
4   V[i].clave >= V[j-1].clave o j=1.
```

Inserción directa

PENDIENTE

```

```

```
COM
(1) Desde i = 2 hasta N hacer
(2)   Aux <- V[i]
(3)   j=i-1
(4)   MIENTRAS j > 0 y Aux.clave < V[j].clave hacer
(5)     V[j+1] <- V[j]
(6)     j <- j-1
(7)   FIN MIENTRAS
(8)   V[j+1] <- Aux
(9) FIN DESDE
FIN
```

→ El bucle interior ("mientras") tiene distinto comportamiento según el orden del conjunto:

- **Mejor caso (conjunto ya ordenado):** Solo evalúa la condición sin entrar al bucle, por lo que es **O(1)** por iteración, resultando en **O(N)** en total.
- **Peor caso (conjunto en orden inverso):** Ejecuta el bucle $N - i$ veces por iteración, resultando en una complejidad **O(N²)**.
- **Caso promedio (claves al azar):** También tiene una complejidad **O(N²)**.

Insercion directa en listas

```
Comienzo:
(1) Mientras no (ListaDeEntrada.Vacía) hacer:
  (2) ElementoAMover ← ListaDeEntrada.Primero
  (3) ListaDeEntrada.EliminarPrimero
  (4) ListaDeSalida.InsertarOrdenado(ElementoAMover)
(5) Fin mientras
(6) Devolver ListaDeSalida.
Fin
```

- **Peor caso y caso promedio:** O(N²)
- **Mejor caso:** O(N), cuando la lista estaba ordenada al revés.

Selection sort

En la i-ésima iteración, selecciono el elemento con la clave menor entre A[i],...,A[n] y lo intercambiamos con A[i].

- $O(n^2)$ en todos los casos

1. Se busca la menor clave del conjunto, se transfiere al área de salida y se reemplaza por un valor infinito.
2. Se repite el proceso, buscando la segunda menor clave.
3. Se continúa hasta haber seleccionado todas las claves.

```
(1) Desde i = 1 hasta N - 1 hacer:
    IndiceDelMenor ← i

    (2) ClaveMenor ← V[i].clave

    (3) Desde j = i + 1 hasta N hacer:
        (4) Si V[j].clave < ClaveMenor entonces:
            (5) IndiceDelMenor ← j
            (6) ClaveMenor ← V[j].clave
```

```
10          (7) Fin si
11      (8) Fin desde
12
13      (9) (10) intercambia (V[i], V[IndiceDelMenor])
14  Fin desde
```

Shellsort

Shellsort mejora la inserción dividiendo el arreglo en subgrupos de elementos separados por una distancia `gap`. En cada fase, se realiza una inserción sobre estos subgrupos. A medida que el `gap` se reduce, los datos se ordenan progresivamente hasta que `gap = 1`, donde se finaliza con un insertion sort normal.

- Utiliza una *secuencia de incrementos* (`gaps`), por ejemplo `(5, 3, 1)`.
- En cada paso se ordenan por inserción los elementos separados por ese `gap`.
- Cuando `gap = 1`, se aplica una inserción clásica sobre un arreglo ya parcialmente ordenado.

1. Elegir una secuencia decreciente de incrementos, como `n / 2, n / 4, ..., 1`.
2. Para cada incremento `gap`:
   - Se realiza una inserción con espaciado entre elementos separados por `gap`.
3. El proceso termina al llegar a `gap = 1`.

Rendimiento

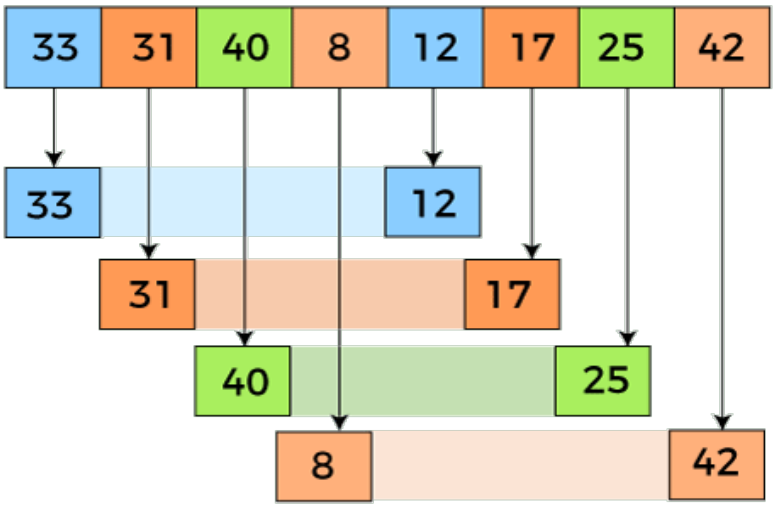
- Muy dependiente de la secuencia de incrementos.
- Con la secuencia original de Shell, el peor caso es $O(n^2)$.
- Usando secuencias mejores (como división por 2.2 o la mediana de Gonnet), se puede alcanzar $O\left(N^{\{\frac{3}{2}\}}\right)$ o incluso $O(N \log^2 N)$ en la práctica.

Ventajas

- Más rápido que la inserción directa, especialmente en arreglos medianos.
- Código simple y compacto.

Desventajas

- No garantiza el mejor rendimiento para todos los casos.
- No es un algoritmo estable.
- No recomendado para grandes volúmenes de datos.



```
1   /**
2    * Ordenación Shell, utilizando una secuencia sugerida por Gonnet.
3    */
4   public static <AnyType extends Comparable<? super AnyType>>
5   void shellsort( AnyType [ ] a )
6   {
7      for( int gap = a.length / 2; gap > 0;
8              gap = gap == 2 ? 1 : (int) ( gap / 2.2 ) )
9         for( int i = gap; i < a.length; i++ )
10        {
11           AnyType tmp = a[ i ];
12           int j = i;
13
14           for( ; j >= gap && tmp.compareTo( a[j-gap] ) < 0; j -= gap )
15              a[ j ] = a[ j - gap ];
16           a[ j ] = tmp;
17        }
18  }
```

Quicksort

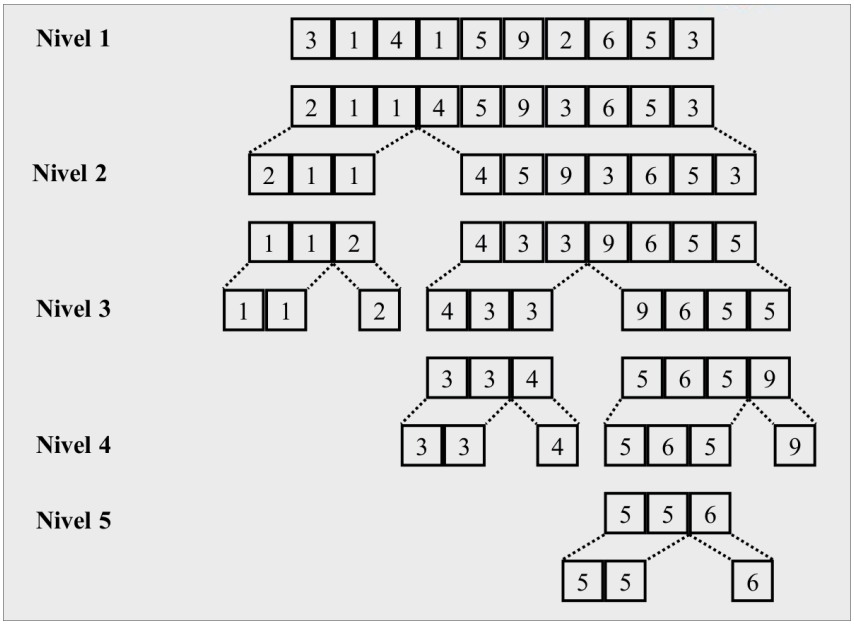
Algoritmo "Divide y vencerás". Particiona el array en dos sub-arrays, ordenando cada uno por separado. Toma un elemento v como *pivot* (se busca que sea la *mediana* del conjunto) alrededor del cual reordenar los elementos del array.

- Selecciona un pivote y particiona el arreglo en dos subarreglos: los menores y los mayores que el pivote. Luego ordena recursivamente cada subarreglo.

$\rightarrow$ Algoritmo $\boldsymbol{O(N \log N)}$ en promedio, aunque $\boldsymbol{O(N^2)}$ en el peor caso (puede ser estadísticamente improbable).

Pasos del algoritmo para Quicksort(S):

1. Si el número de elementos de S es 0 o 1, entonces callate.
2. Seleccionar cualquier elemento v de S. Lo llamamos **pivote**
3. Particionar $S - \{v\}$ (los elementos restantes de S) en dos grupos disjuntos: L : Los elementos a la izquierda de v son menores que este, y R : los que están a la derecha son mayores.
4. Devolver el resultado de Quicksort(L) seguido de v seguido de Quicksort(R)




```
1 función particion(i, j: entero; pivote: TipoClave): entero
2   // Divide V[i]..V[j] en dos grupos:
3   // claves < pivote a la izquierda y claves ≥ pivote a la derecha.
4   // Devuelve el índice donde empieza el grupo de la derecha.
5
6   variables:
7     L, R: enteros
8
9   comienzo:
10    L ← i
11    R ← j
12
13    repetir
14      intercambiar V[L], V[R]
15
16      mientras V[L].clave < pivote hacer
17        L ← L + 1
18      fin mientras
19
20      mientras V[R].clave ≥ pivote hacer
21        R ← R - 1
22      fin mientras
23    hasta que L > R
24
25    devolver L
26 fin función
```

```
1 procedimiento quicksort(i, j: entero)
2   // Clasifica V[i]..V[j] del arreglo externo V
3
4   variables:
5     pivote: TipoClave
6     IndicePivote, k: enteros
7
```

```
 8   comienzo:
 9     IndicePivote ← EncuentraPivote(i, j)
10
11     si IndicePivote ≠ 0 entonces
12       pivote ← V[IndicePivote].clave
13       k ← particion(i, j, pivote)
14       quicksort(i, k - 1)
15       quicksort(k, j)
16     fin si
17 fin procedimiento
```

Análisis

- El mejor caso se da cuando el pivote es en cada elección la mediana del conjunto.
- El peor caso se da cuando el pivote elegido es un extremo del conjunto.
- Se ejecutarán $2N - 1$ llamadas al algoritmo.

Mergesort

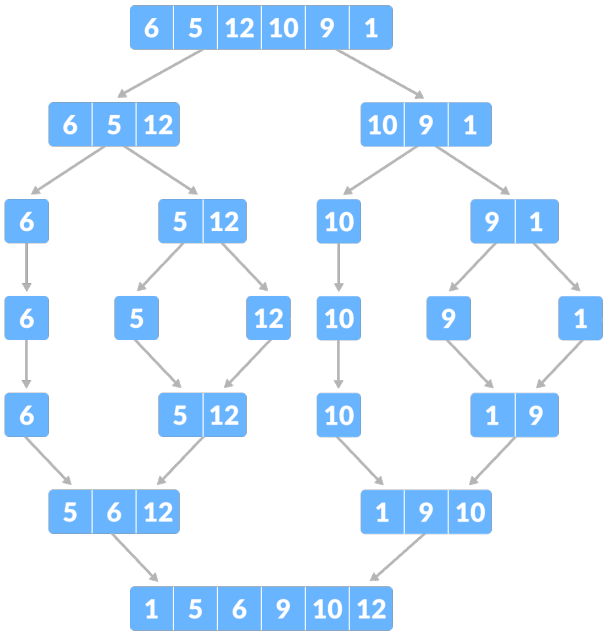
Mergesort es un algoritmo de tipo divide y vencerás. Divide el arreglo en mitades, ordena cada mitad recursivamente y luego las fusiona en un solo arreglo ordenado. Esta fusión se hace en tiempo lineal usando memoria adicional.

- Siempre tiene complejidad $O(N \log N)$.
- Es estable y funciona bien incluso en el peor caso.
- Usa memoria adicional proporcional al tamaño del arreglo.

El algoritmo está compuesto de tres etapas:

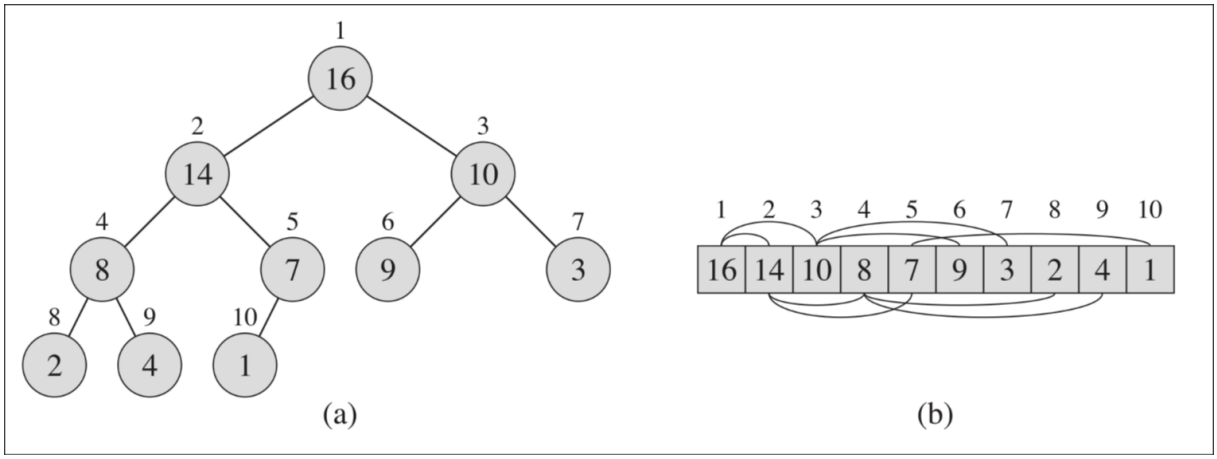
1. Si el número de elementos que hay que ordenar es 0 o 1, volver.
2. Ordenar recursivamente y por separado la primera y la segunda mitades.
3. Mezclar las dos mitades ordenadas para obtener un total ordenado.

Es ideal cuando se requiere estabilidad en la ordenación o cuando se trabaja con estructuras enlazadas donde no hay penalización por el uso de memoria adicional.



Heapsort

Heapsort es un algoritmo de ordenación con complejidad $O(n \log n)$ tanto en el peor como en el caso promedio. Utiliza una estructura auxiliar —un montículo o heap— para organizar los elementos antes de ordenarlos.



- Se parte de un conjunto E de elementos a ordenar.
- Se utiliza una estructura auxiliar S (el heap) para almacenar los elementos ordenados parcialmente.
- El algoritmo se puede describir mediante las siguientes operaciones:
  - `INSERTA(x, S)` — Inserta un elemento en el heap.
  - `MIN(S)` — Devuelve el mínimo (o máximo) del heap.
  - `SUPRIME(y, S)` — Elimina el elemento y del heap.
  - `VACIA(S)` — Verifica si el heap está vacío.

El pseudocódigo puede representarse así:

```
para x en E hacer
  INSERTA(x, S)

mientras no VACIA(S) hacer
  y ← MIN(S)
  procesar(y) // Por ejemplo, guardar en lista final
  SUPRIME(y, S)
```

Cada inserción y eliminación en el heap toma $O(\log n)$, y como se hacen n operaciones de cada tipo, el costo total es $O(n \log n)$.

El heap se implementa en un vector V de tamaño n, donde:

- La raíz está en $V[1]$.
- El hijo izquierdo de $V[i]$ está en $V[2i]$, y el derecho en $V[2i + 1]$.
- Los nodos internos están en las posiciones 1 a $\lfloor \frac{n}{2} \rfloor$, y las hojas en el resto.

Construcción del heap (Heapify)

Para transformar un arreglo desordenado en un heap:

```
para i desde ⌊n/2⌋ hasta 1 hacer
  DesplazaElemento(i, n)
```

La operación `DesplazaElemento` garantiza que el subárbol con raíz en la posición `i` cumple la propiedad del heap.

Ordenación
Luego de construir el heap, se procede a extraer el mínimo (o máximo) repetidamente:

```
1 para i desde n hasta 2 hacer
2   intercambiar V[1] con V[i]
3   DesplazaElemento(1, i - 1)
```

Después de cada paso, el heap está entre las posiciones $[1, i-1]$, y la parte ordenada entre $[i, n]$.

Procedimiento DesplazaElemento

```
 1 DesplazaElemento(Primero, Último)
 2   Actual ← Primero
 3   mientras Actual ≤ ⌊Último / 2⌋ hacer
 4     si Último = 2 * Actual entonces
 5       // Solo tiene hijo izquierdo
 6       si V[Actual].clave > V[2*Actual].clave entonces
 7         intercambiar(V[Actual], V[2*Actual])
 8       fin si
 9       Actual ← Último
10     sino
11       Menor ← índice del menor entre V[2*Actual] y V[2*Actual+1]
12       si V[Actual].clave > V[Menor].clave entonces
13         intercambiar(V[Actual], V[Menor])
14         Actual ← Menor
15       sino
16         Actual ← Último
17       fin si
18     fin si
19   fin mientras
```

Propiedades
- **Complejidad temporal:** $O(n \log n)$ en todos los casos.
- **Memoria adicional:** No requiere, se realiza in-place.
- **Estabilidad:** No es estable.
- **Ventaja:** Rendimiento predecible, ideal cuando se requiere robustez ante el peor caso.

Bin sort / Bucket sort

Bin sort (o bucket sort) es un algoritmo de ordenación que puede alcanzar una complejidad de $O(n)$ si se conoce de antemano el rango de las claves. A diferencia de algoritmos como quicksort o mergesort, que tienen una cota mínima de $O(n \log n)$, bin sort aprovecha conocimiento adicional sobre los datos.

- Supone que las claves son enteras dentro de un rango conocido (por ejemplo, $1..m$).
- Si hay n claves diferentes dentro de ese rango, y sin duplicados, puede lograrse una ordenación en $O(n)$.
- Cuando existen duplicados, se utilizan **listas enlazadas** en cada "cubo" para almacenar múltiples elementos con la misma clave.

Estructura

- Se utiliza un arreglo `B[0..m-1]` de listas vacías, que actúan como "cubos" (buckets).
- Cada elemento de entrada se distribuye en uno de los cubos en base a su clave.
- Cada cubo contiene una lista de elementos que comparten el mismo valor de clave (o un rango en caso de bucket sort más general).
- Opcionalmente, cada lista puede ordenarse internamente (por ejemplo, por inserción).
- Finalmente, se concatenan todas las listas en orden para obtener el resultado final.

Pseudocódigo general

```
Binsort(entrada, m)
  urnas ← nuevo array de m listas vacías
  para i de 1 a n hacer
    insertar entrada[i] en urnas[clave(entrada[i])]

  para i de 0 a m-1 hacer
    ordenar(urnas[i]) // si se desea ordenar dentro de cada cubo

  salida ← concatenar urnas[0..m-1]
  devolver salida
```

Complejidad

- Distribuir los elementos en cubos: $O(n)$
- Ordenar cada lista: $O(m)$ (si hay pocos elementos por cubo o se usa inserción)
- Concatenación: $O(m)$
- En total: $O(n + m)$

Esto implica que si el número de cubos m es proporcional a n (y los datos están distribuidos uniformemente), el rendimiento puede ser lineal.

Observaciones

- Es útil cuando las claves son numéricas y se conocen de antemano.
- El uso de listas enlazadas para manejar duplicados permite eficiencia sin colisiones.
- Puede ser implementado también con arreglos si se conoce la cantidad exacta de repeticiones.

Radix sort

Radix sort es un algoritmo de ordenación no comparativo que clasifica registros con claves compuestas por varios campos (o dígitos) ordenando iterativamente por cada campo, del menos significativo al más significativo.

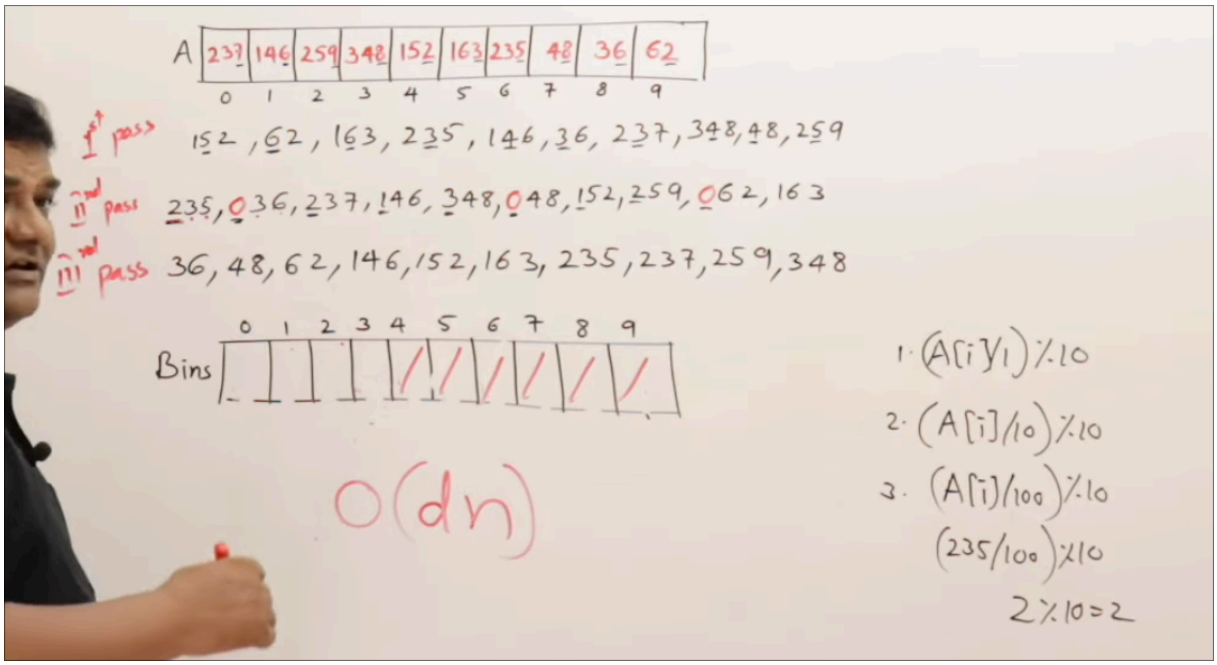
Supuestos

- Las claves de los registros están formadas por k componentes: $f_1, f_2, ..., f_k$, donde cada f_i es de un tipo t_i.
- Se desea clasificar los registros en **orden lexicográfico** respecto a esas claves.
- Ejemplos de claves:
  - Una fecha: `{día: 1..31, mes: 1..12, año: 1900..1999}`
  - Una cadena de texto: `array[1..10] of char`

Idea principal

Radix sort funciona aplicando **Bin Sort** o algún método estable sobre cada componente de la clave, comenzando por el **menos significativo** (f_k) hasta el más significativo (f_1).

Cada paso asegura que el orden previo se conserva gracias a que se utiliza una ordenación estable.



Pasos del algoritmo

```
procedure RadixSort
  para i desde k hasta 1 hacer
    para cada valor v de tipo tᵢ hacer
      vaciar Bi[v] // inicializar urnas

    para cada registro R en A hacer
      insertar R al final de Bi[valor(R.fᵢ)]

    A ← arreglo vacío
    para cada valor v de tipo tᵢ, en orden creciente hacer
      concatenar Bi[v] al final de A
  fin para
fin
```

Donde:

- `Bi` es un arreglo de listas, con una lista para cada valor posible de t_i.
- Se garantiza la **estabilidad** insertando los elementos **al final** de cada lista.

Análisis de complejidad

- Supongamos que s_i es la cantidad de valores distintos del tipo t_i.
- Para cada iteración i (de k a 1):
  - Vaciar las urnas: $O(s_i)$
  - Distribuir los n registros en las urnas: $O(n)$
  - Concatenar las urnas en orden: $O(s_i)$

Entonces, el costo total es:

```
T(n) = ∑ ( O(sᵢ) + O(n) + O(sᵢ) ) = O(k·n + ∑ sᵢ)
```

Si cada s_i es acotado y pequeño en comparación con n, la complejidad es esencialmente $O(kn)$, es decir, **lineal** respecto al número de elementos.

Consideraciones

- El algoritmo es eficiente si los campos de las claves tienen un rango limitado.
- Es importante que el método usado en cada fase sea **estable**, como bin sort.
- No se basa en comparaciones, lo que lo diferencia de quicksort o heapsort.
- Es particularmente útil para claves de longitud fija, como enteros grandes, fechas o strings.

Bubblesort

Algoritmo feo. Se basa en recorrer repetidamente el arreglo comparando pares de elementos adyacentes y **permutándolos si están en orden incorrecto**. Este proceso se repite hasta que no se necesitan más intercambios.

Funcionamiento

- En cada pasada, los elementos "más grandes" se van desplazando hacia el final del arreglo, como burbujas que suben.
- Después de la i-ésima pasada, los últimos i elementos ya están en su posición final.
- Puede detectarse si el arreglo está ordenado antes de tiempo usando una bandera que registre si hubo intercambios.

Pseudocódigo

```
procedure BubbleSort(A[1..n])
  para i desde 1 hasta n - 1 hacer
    intercambiado ← falso
    para j desde 1 hasta n - i hacer
      si A[j] > A[j+1] entonces
        intercambiar A[j] y A[j+1]
        intercambiado ← verdadero
    si no intercambiado entonces
      salir // el arreglo ya está ordenado
fin
```

Complejidad

- **Mejor caso** (arreglo ya ordenado):
  - Comparaciones: $O(n)$
  - Intercambios: 0
- **Caso promedio**:
  - Comparaciones: $O(n^2)$
  - Intercambios: $O(n^2)$
- **Peor caso** (arreglo en orden inverso):
  - Comparaciones: $O(n^2)$
  - Intercambios: $O(n^2)$

Características

- **Estable**: no cambia el orden de elementos iguales.
- **In-place**: no requiere memoria adicional.
- Muy **ineficiente** para grandes volúmenes de datos.
- Fácil de implementar y comprender.