

Ordenamientos y Búsquedas

Mauricio Avilés

Contenido

- Shell Sort
- Quicksort in-place
- Bin Sort
- Radix Sort
- ~~Búsqueda binaria~~
- ~~Búsqueda por interpolación~~

Lecturas

- Capítulo 7, sección 9.1
 - Shaffer, C. A. "Data Structures & Algorithm Analysis in C++" (3rd ed., Dover). Mineola, NY. 2011.
- Capítulo 11
 - Goodrich, M. T., Tamassia, R., & Mount, D. M. "Data structures and algorithms in C++" (2nd ed., Wiley). Hoboken, NJ: Wiley. 2011.
- Capítulo 8
 - Joyanes, Aguilar, & Martínez. Estructura de datos en C ++. Madrid: McGraw-Hill Interamericana. 2007.

Shell Sort

- Algoritmo de ordenamiento basado en **comparaciones**
- **No estable**
- Puede verse como una **generalización** del **Insertion** o del **Bubble**
- Empieza ordenando elementos que se encuentran **lejos** entre ellos
- **Reduce** progresivamente la distancia entre los elementos que compara
- Al iniciar con elementos separados, puede ubicar **elementos** que estén **lejos de su lugar** más rápidamente que comparando vecinos contiguos
- Su desempeño depende de la **secuencia** de distancias que se utilice

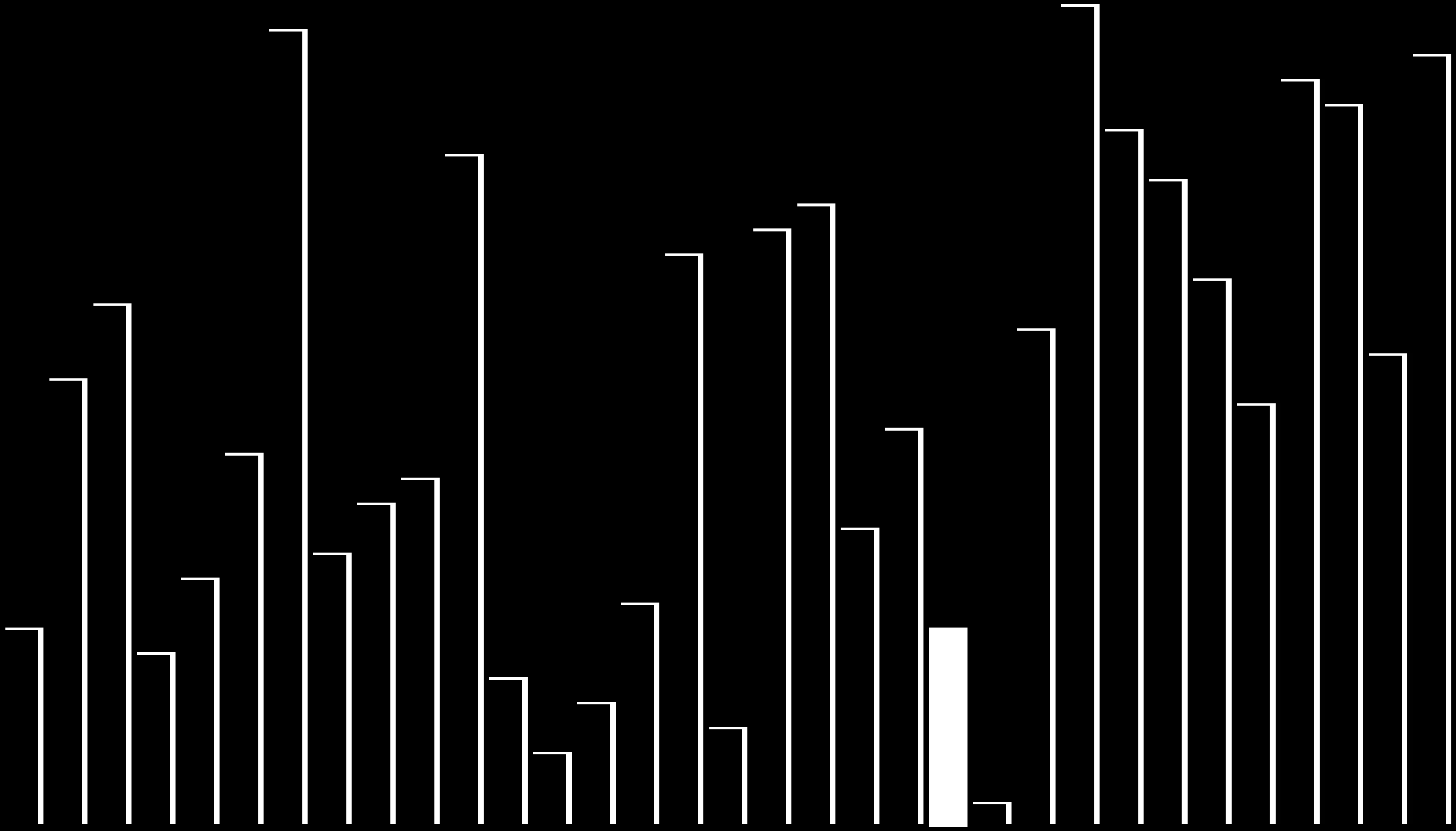
Historia



- Creado por **Donald L. Shell** en 1959
- Por muchos años el algoritmo era referido por **error** como Shell-Metzner
- **Marlene Metzner** era programadora y compartió por correo una implementación del algoritmo escrita **ensamblador** en 1961
- Algunos empezaron a llamar al algoritmo por el nombre erróneo y esto trascendió en **libros**
- Posteriormente Metzner se **percató** de ese error y manifestó públicamente que no había tenido que ver con la creación del algoritmo

Estrategia

- Hacer que la lista esté bastante ordenada
- El ordenamiento divide al arreglo en listas virtuales
- Cada sublista se ordena utilizando Insertion Sort
- Se selecciona un nuevo grupo de listas virtuales
- Se repite disminuyendo la cantidad de listas virtuales hasta llegar a una sola lista
- Insertion Sort es eficiente con listas bastante ordenadas



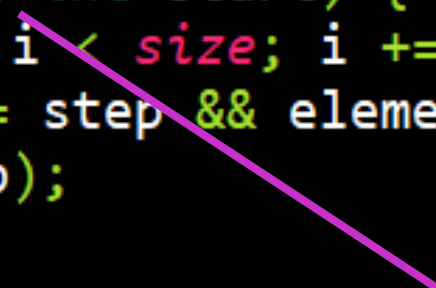
Ejemplo de implementación

- **Método** implementado en la clase **ArrayList**
- Trabaja sobre el arreglo **elements**
- Se utiliza como función auxiliar el **Insertion Sort** que recibe como parámetros el tamaño del paso y la posición de inicio de la sublista dentro del arreglo

Función principal, lleva control del incremento que inicia en la mitad del tamaño y llega hasta 1. Con cada incremento, determina el inicio de cada lista virtual y las ordena con Insertion Sort.

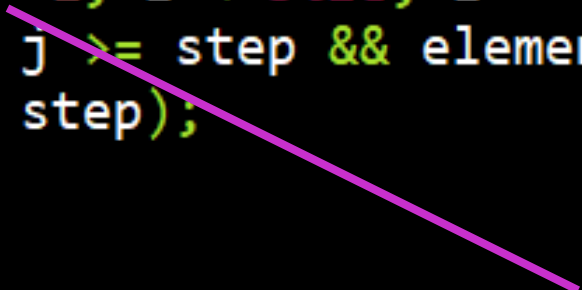
```
void shellSort() {  
    for (int step = size / 2; step > 0; step /= 2)  
        for (int start = 0; start < step; start++)  
            insertionSort(step, start);  
}  
void insertionSort(int step, int start) {  
    for (int i = start + 1; i < size; i += step)  
        for (int j = i; j >= step && elements[j] < elements[j - 1]; j -= step)  
            swap(j, j - step);  
}
```

```
void shellSort() {  
    for (int step = size / 2; step > 0; step /= 2)  
        for (int start = 0; start < step; start++)  
            insertionSort(step, start);  
}  
void insertionSort(int step, int start) {  
    for (int i = start + 1; i < size; i += step)  
        for (int j = i; j >= step && elements[j] < elements[j - 1]; j -= step)  
            swap(j, j - step);  
}
```



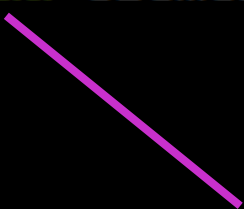
Recibe el tamaño del incremento y la posición donde inicia la lista virtual.

```
void shellSort() {  
    for (int step = size / 2; step > 0; step /= 2)  
        for (int start = 0; start < step; start++)  
            insertionSort(step, start);  
}  
void insertionSort(int step, int start) {  
    for (int i = start + 1; i < size; i += step)  
        for (int j = i; j >= step && elements[j] < elements[j - 1]; j -= step)  
            swap(j, j - step);  
}
```



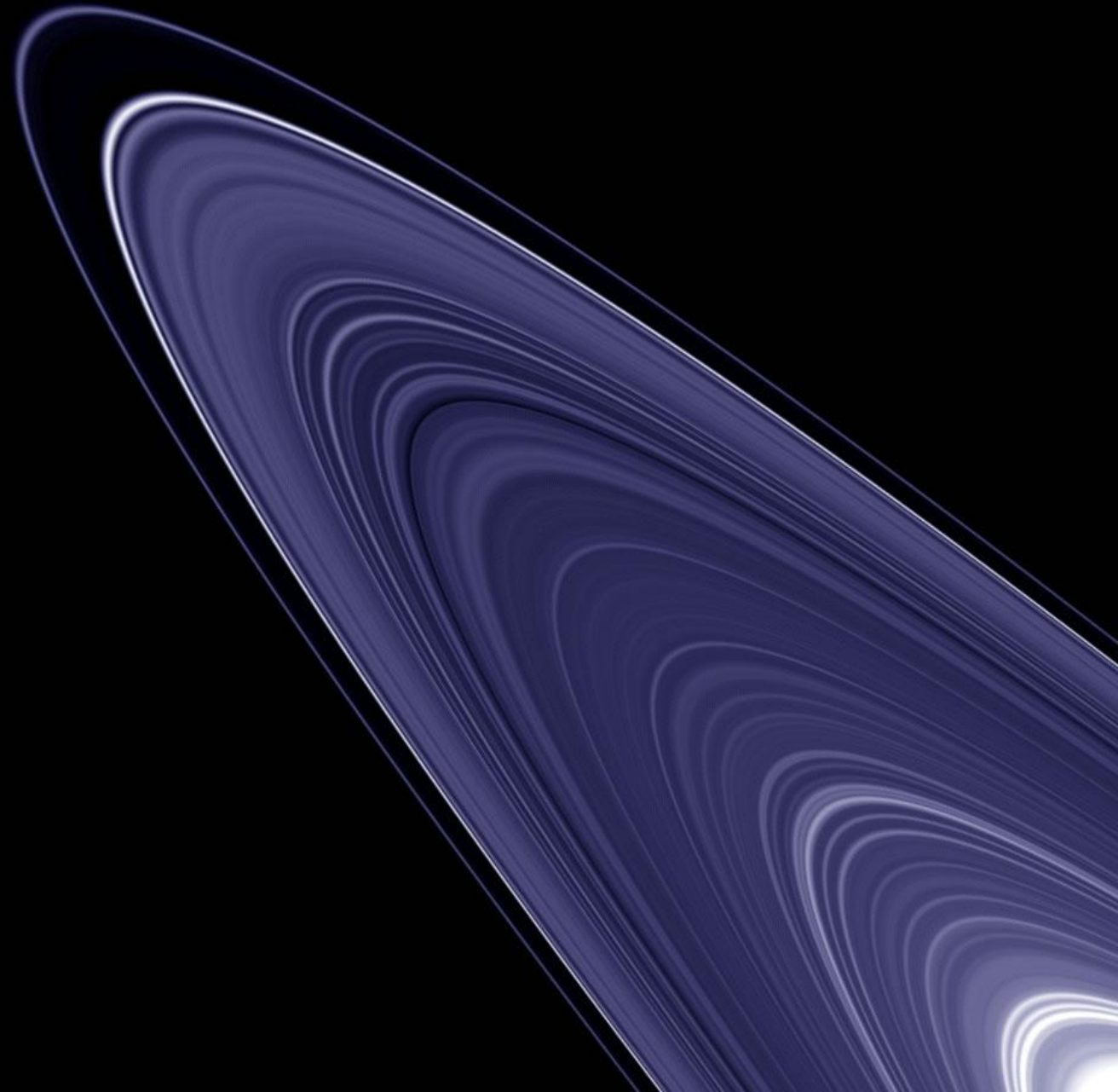
Divide la lista en parte ordenada y parte desordenada. La parte ordenada va desde el inicio hasta el contador i. La posición donde inicia la lista ordenada aumenta con el tamaño del incremento cada vez.

```
void shellSort() {  
    for (int step = size / 2; step > 0; step /= 2)  
        for (int start = 0; start < step; start++)  
            insertionSort(step, start);  
}  
void insertionSort(int step, int start) {  
    for (int i = start + 1; i < size; i += step)  
        for (int j = i; j >= step && elements[j] < elements[j - 1]; j -= step)  
            swap(j, j - step);  
}
```



El contador j se usa para ir desde la posición i hasta el inicio de la lista ordenada haciendo intercambios hasta encontrar la posición adecuada del elemento.

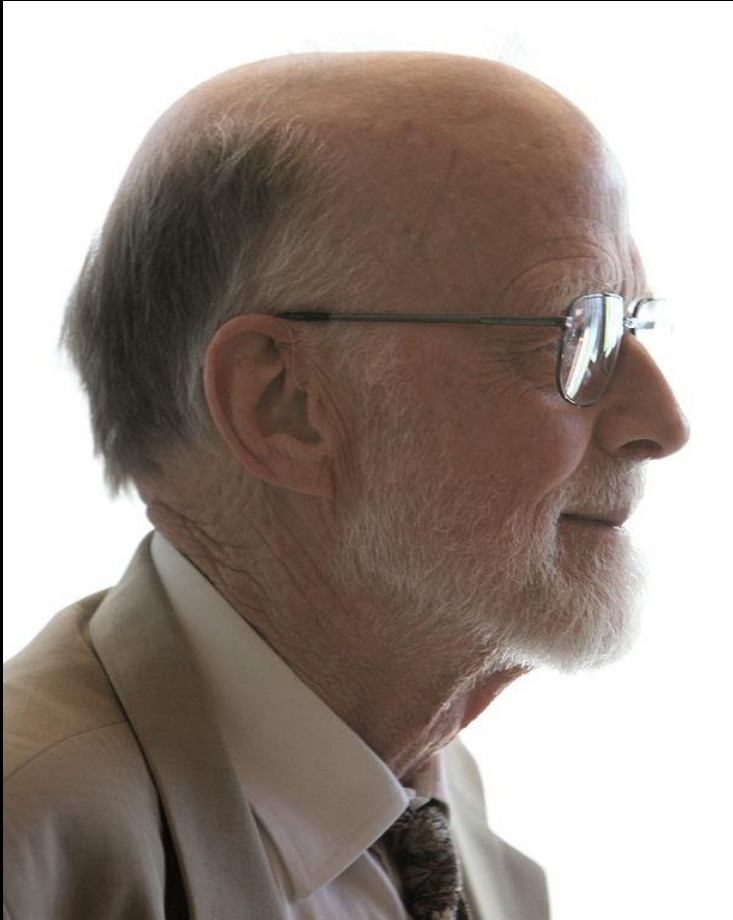
- Aunque tiene tres ciclos anidados es **más eficiente** que:
 - Bubble Sort
 - Selection Sort
 - Insertion Sort
- Se ha encontrado que dividiendo el salto por **2.2** se consigue un mejor tiempo de ejecución



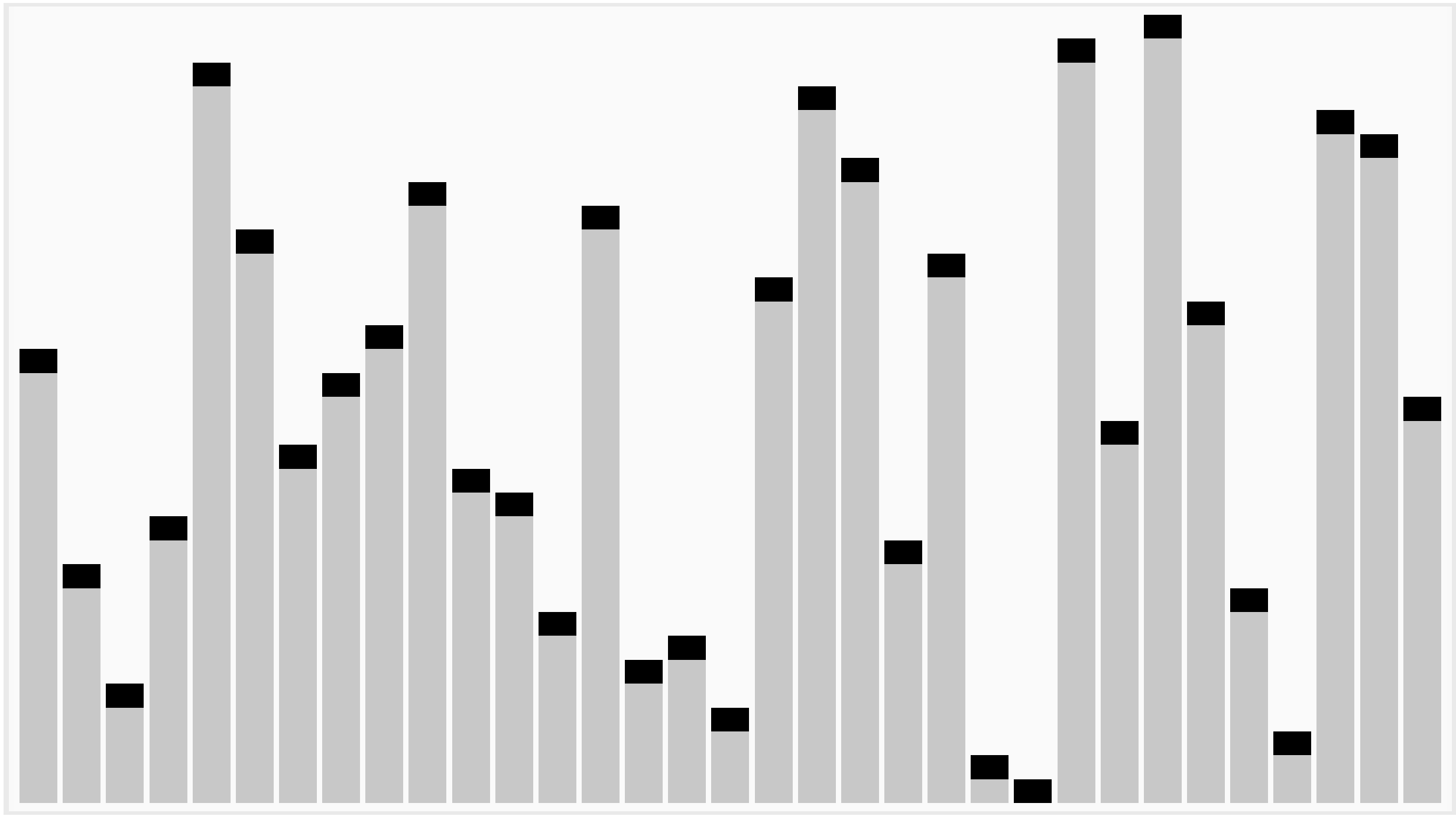
Quicksort in-place

- Es el algoritmo de ordenamiento de propósito general **más rápido** conocido
- Ordenamiento basado en **comparaciones**
- **No estable**
- **No requiere** espacio adicional en memoria para realizar el ordenamiento
- Trabaja sobre la **misma estructura** donde están almacenados los elementos que se ordenarán

Historia



- Creado por **Tony Hoare** en 1968
- Colaboraba en un proyecto de **traducción** automática como pasantía en la Universidad Estatal de Moscú
- Necesitaba **ordenar las palabras** de oraciones para luego buscarlas en un diccionario ruso-inglés que estaba en cinta magnética
- Creo el **algoritmo** y lo empezó a implementar pero sólo programó la partición
- Al regresar a Inglaterra, le encargaron programar el **Shell Sort**
- Apostó con su jefe a que tenía un algoritmo más **rápido**
- **Ganó** la apuesta
- Ha ganado gran cantidad de **premios** por esta invención y por sus aportes a la definición y diseño de lenguajes de programación



Pasos

01

Se selecciona un valor llamado pivote

02

Crear particiones:
reordenar los
contenidos de la lista

- Menores a la izquierda + pivote + mayores

Los elementos en cada partición no están ordenados

03

Ejecutar recursivamente el algoritmo en cada una de las particiones

Selección del pivote

- El **primero** o el último → simple
 - Si la lista viene ordenada inversamente, el algoritmo se degrada
- Seleccionar posición de forma **aleatoria** → es caro por los cálculos
- Seleccionar el que está en el **medio** → simple
 - Efecto parecido a seleccionar una posición aleatoria

Creación de la partición

- La idea es **reordenar** los elementos de forma que los menores estén antes que el pivote y los mayores después
- Una estrategia es crear un **método** que reciba los índices donde empieza y termina el segmento a ordenar, además de la posición del pivote
- Se ubica el **pivote al final** del segmento
- Se **busca** desde el índice menor hacia arriba un **elemento mayor** que el pivote
- Se **busca** desde el índice mayor hacia abajo un **elemento menor** que el pivote
- Luego se **intercambian** ambos elementos
- Esto se repite hasta que los **índices se encuentren** en el medio de la lista.

Ejemplo de implementación

- **Método** implementado en la clase **ArrayList**
- Trabaja sobre el arreglo **elements**
- Se utilizan varias funciones **auxiliares**:
 - **findPivot**
 - Recibe índices menor y mayor
 - Retorna posición del pivote
 - **partition**
 - Recibe posición del pivote, índices inferior y superior
 - Modifica la región del arreglo entre los índices inferior y superior particionada de forma que primero están los menores que el pivote, el pivote y luego los mayores o iguales que el pivote
 - Retorna la posición del pivote dentro de la región
 - **swap**
 - Recibe dos índices dentro del arreglo
 - Intercambia los elementos en esas posiciones

```
void quickSort(int low, int high) {  
    if (high - low <= 0) {  
        return;  
    }  
    int pivotIndex = findPivot(low, high);  
    pivotIndex = partition(pivotIndex, low, high);  
    quickSort(low, pivotIndex - 1);  
    quickSort(pivotIndex + 1, high);  
}
```

Función principal del algoritmo. Recibe el índice inferior y el superior de la sublista del arreglo que se está ordenando. Inicialmente se invoca con low=0 y high = size-1

```
void quickSort(int low, int high) {  
    if (high - low <= 0) {  
        return;  
    }  
    int pivotIndex = findPivot(low, high);  
    pivotIndex = partition(pivotIndex, low, high);  
    quickSort(low, pivotIndex - 1);  
    quickSort(pivotIndex + 1, high);  
}
```

Si la lista tiene un elemento, entonces low y high son iguales. Si la lista tiene cero elementos, entonces low es mayor que high. En ambos casos la resta de los índices es menor o igual a cero. En dado caso, no ordenar la lista.

```
void quickSort(int low, int high) {  
    if (high - low <= 0) {  
        return;  
    }  
    int pivotIndex = findPivot(low, high);  
    pivotIndex = partition(pivotIndex, low, high);  
    quickSort(low, pivotIndex - 1);  
    quickSort(pivotIndex + 1, high);  
}
```

Se dan el índice inferior y superior. La función findPivot se encarga de calcular la posición del pivote en esa lista.

```
void quickSort(int low, int high) {  
    if (high - low <= 0) {  
        return;  
    }  
    int pivotIndex = findPivot(low, high);  
    pivotIndex = partition(pivotIndex, low, high);  
    quickSort(low, pivotIndex - 1);  
    quickSort(pivotIndex + 1, high);  
}
```

El método partition se encarga de particionar la parte del arreglo que empieza en la posición low y termina en high. Se le indica cuál es la posición del pivote actualmente.

El método reordena esa región del arreglo de modo que quedan primero los menores al pivote, luego el pivote y de último los mayores.

Al reordenar, el pivote pudo haber cambiado su posición, por lo que la retorna.


```
void quickSort(int low, int high) {  
    if (high - low <= 0) {  
        return;  
    }  
    int pivotIndex = findPivot(low, high);  
    pivotIndex = partition(pivotIndex, low, high);  
    quickSort(low, pivotIndex - 1);  
    quickSort(pivotIndex + 1, high);  
}
```

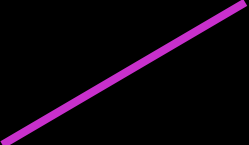
Se ordena recursivamente la sublista de menores, que va desde la posición low hasta la posición anterior al pivote.

Se ordena recursivamente la sublista de mayores, que va desde la posición siguiente al pivote y hasta la posición high.

No es necesario “unir” las dos listas porque todo está ocurriendo en el mismo arreglo sin generar listas nuevas.

```
int findPivot(int low, int high) {  
    return (low + high) / 2;  
}
```

```
void swap(int i, int j) {  
    E temp = elements[i];  
    elements[i] = elements[j];  
    elements[j] = temp;  
}
```



Se elige como pivote el elemento que se encuentra en el medio de la región delimitada por low y high. El cálculo de esta posición es la suma de ambos índices dividido entre dos.

```
int findPivot(int low, int high) {  
    return (low + high) / 2;  
}
```

```
void swap(int i, int j) {  
    E temp = elements[i];  
    elements[i] = elements[j];  
    elements[j] = temp;  
}
```

Este método es utilizado por partition. Recibe dos índices e intercambia los elementos en esas posiciones.

```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

Se ubica el pivote en el último espacio para reordenar el resto de elementos.

```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

Se actualiza la posición del pivote para recordar dónde se encuentra.

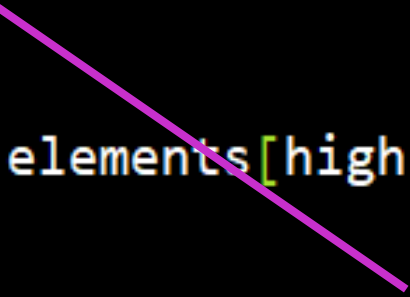
```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

Se actualiza la posición superior para que no tome en cuenta al pivote.

```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

El ciclo se repite mientras los índices inferior y superior no se encuentren. El índice low va a ir aumentando, mientras que high va a ir disminuyendo

```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```



Este ciclo aumenta el índice low hasta que se encuentre algún elemento que no sea menor que el pivote.


```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

Este ciclo disminuye el índice high hasta que se encuentra un elemento que sea menor que el pivote o se encuentre con el índice low.

```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

En este punto pueden suceder dos cosas (1) low apunta a un elemento mayor o igual que el pivote y high a uno menor, o (2) los dos apuntan al mismo elemento y ya se encontraron.

```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

Si los índices no se han encontrado, entonces se intercambian los elementos para ponerlos en una posición adecuada.

```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

Al terminar el ciclo, todos los elementos menores están al inicio y los mayores al final. Los índices low y high quedan apuntando al primer elemento mayor. Lo que hace falta es poner al pivote en el medio.

```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

En caso de que todos los elementos fueran menores que el pivote, no es necesario moverlo de su ubicación actual.

```
int partition(int pivotIndex, int low, int high) {  
    swap(pivotIndex, high);  
    pivotIndex = high;  
    high--;  
    while (low < high) {  
        while (elements[low] < elements[pivotIndex]) {  
            low++;  
        }  
        while (low < high && elements[high] >= elements[pivotIndex]) {  
            high--;  
        }  
        if (low < high) {  
            swap(low, high);  
        }  
    }  
    if (elements[low] > elements[pivotIndex]) {  
        swap(low, pivotIndex);  
    }  
    return low;  
}
```

Se retorna la posición donde finalmente quedó ubicado el pivote.

Llamada inicial al algoritmo QuickSort.

```
void sort() {  
    quickSort(0, size - 1);  
}
```

Bin Sort

- **Distribuye** elementos en una estructura de baldes o urnas (bins, buckets)
- Esta estructura consiste en un **arreglo de listas enlazadas**
- Dependiendo de su implementación, puede ser **estable**
- Evidentemente, necesita espacio en **memoria adicional**
- En ciertos casos específicos, puede ser **más rápido** que QuickSort
- Existen diferentes **variantes** del algoritmo

Versión estándar (Bucket Sort)

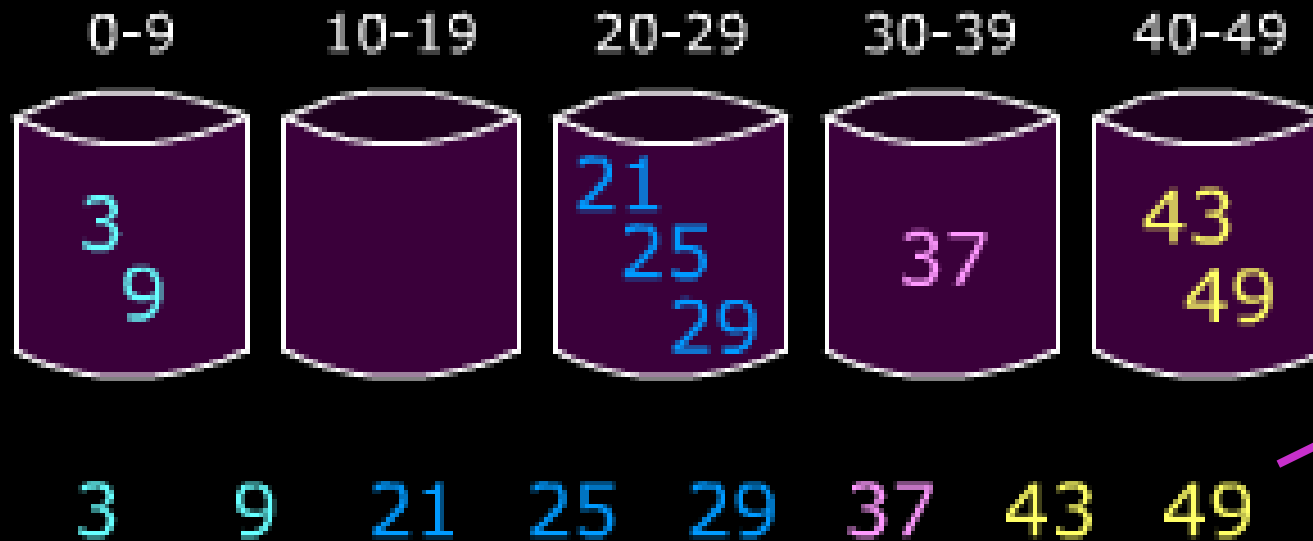
- Es necesario conocer el **rango de valores** a ordenar
- Se define una cantidad **n** de baldes a utilizar
- Se divide el rango de valores en **n rangos más pequeños**, cada uno corresponde a un balde
- Se recorre la lista de elementos, insertando cada uno en el balde que **corresponde** a su valor
- Cada balde se ordena utilizando **otro algoritmo** de ordenamiento como **Insertion Sort**, también puede aplicarse Bin Sort **recursivamente**
- Una vez ordenados, se recorren los baldes, **recolectando** elementos y agregándolo a la lista resultado

29 25 3 49 9 37 21 43



El rango de los valores a ordenar es de 0 a 49. Es decir, 50 posibles valores.

Se usan $n=5$ baldes, por lo que al dividir los 50 valores, se obtienen 5 rangos de 10.



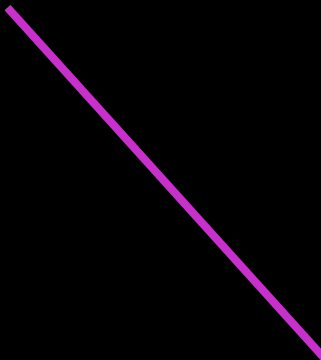
Cada balde se ordena utilizando algún ordenamiento eficiente para listas pequeñas.

Se recolectan los elementos de cada balde ubicándolos de nuevo en la lista original.

Versión rápida

- **Requiere** que los elementos a ordenar sean una **permutación** de los valores en **$0..n-1$** , donde n es la cantidad de elementos
- La estructura de baldes no requiere listas enlazadas, si no **un espacio para cada elemento**
- Se recorren los elementos a ordenar utilizando **el mismo valor del elemento** como índice de la estructura de baldes e insertando cada uno en su posición
- Se **recolectan** los elementos de la estructura de baldes que ya estarían ordenados
- Esta versión es más **eficiente** que el QuickSort, pero es restrictiva
- Modificación
 - Si las llaves pueden repetirse, entonces puede adaptarse para que cada balde sí utilice **listas enlazadas**
 - Agrega complejidad al recolectar las llaves, porque debe preguntarse si la lista está **vacía**

| | | | | | | | | | | | |
|----|---|---|----|---|---|---|---|---|---|---|---|
| 10 | 1 | 8 | 11 | 3 | 9 | 0 | 7 | 4 | 5 | 2 | 6 |
|----|---|---|----|---|---|---|---|---|---|---|---|



Los elementos a ordenar son una permutación, están todos los valores de 0 a 11 y no se repite ninguno.

| | | | | | | | | | | | |
|----|---|---|----|---|---|---|---|---|---|---|---|
| 10 | 1 | 8 | 11 | 3 | 9 | 0 | 7 | 4 | 5 | 2 | 6 |
|----|---|---|----|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|
| | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

Los baldes no requieren listas enlazadas. Hay un espacio para cada elemento a ordenar.

| | | | | | | | | | | | |
|----|---|---|----|---|---|---|---|---|---|---|---|
| 10 | 1 | 8 | 11 | 3 | 9 | 0 | 7 | 4 | 5 | 2 | 6 |
|----|---|---|----|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|
| | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

Los baldes no requieren listas enlazadas. Hay un espacio para cada elemento a ordenar.

| | | | | | | | | | | | |
|--|---|---|----|---|---|---|---|---|---|---|---|
| | 1 | 8 | 11 | 3 | 9 | 0 | 7 | 4 | 5 | 2 | 6 |
|--|---|---|----|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|
| | | | | | | | | | | 10 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

El valor del elemento se utiliza como índice para ubicarlo en la estructura.

| | | | | | | | | | | | |
|--|--|---|----|---|---|---|---|---|---|---|---|
| | | 8 | 11 | 3 | 9 | 0 | 7 | 4 | 5 | 2 | 6 |
|--|--|---|----|---|---|---|---|---|---|---|---|



| | | | | | | | | | | | |
|--|---|--|--|--|--|--|--|--|--|----|--|
| | 1 | | | | | | | | | 10 | |
|--|---|--|--|--|--|--|--|--|--|----|--|

0

1

2

3

4

5

6

7

8

9

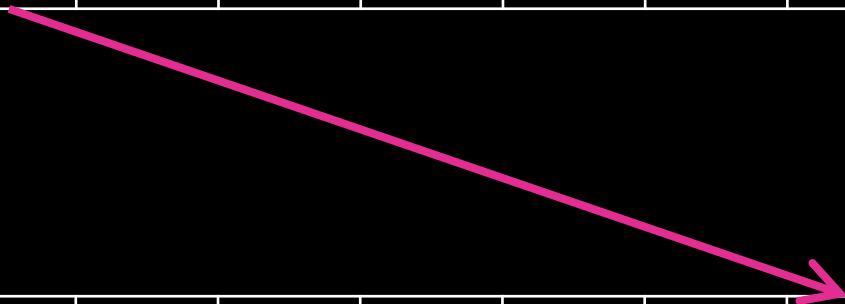
10

11

| | | | | | | | | | | | |
|--|--|--|----|---|---|---|---|---|---|---|---|
| | | | 11 | 3 | 9 | 0 | 7 | 4 | 5 | 2 | 6 |
|--|--|--|----|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | |
|--|---|--|--|--|--|--|--|---|--|----|--|
| | 1 | | | | | | | 8 | | 10 | |
|--|---|--|--|--|--|--|--|---|--|----|--|

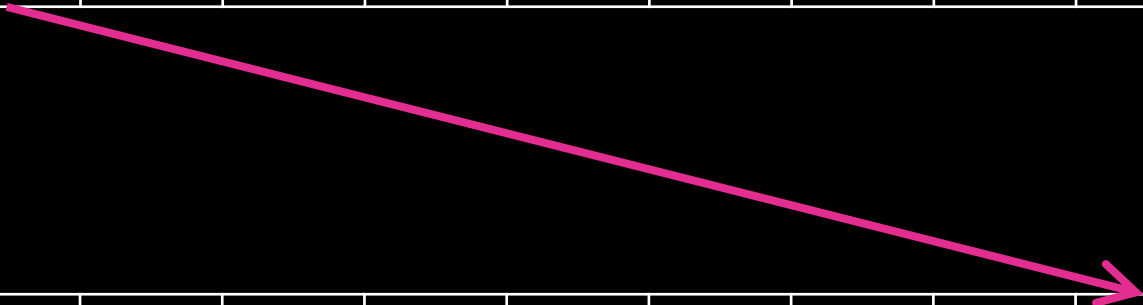
0 1 2 3 4 5 6 7 8 9 10 11



| | | | | | | | | | | | |
|--|--|--|--|---|---|---|---|---|---|---|---|
| | | | | 3 | 9 | 0 | 7 | 4 | 5 | 2 | 6 |
|--|--|--|--|---|---|---|---|---|---|---|---|

| | | | | | | | | | | | |
|--|---|--|--|--|--|--|--|---|--|----|----|
| | 1 | | | | | | | 8 | | 10 | 11 |
|--|---|--|--|--|--|--|--|---|--|----|----|

0 1 2 3 4 5 6 7 8 9 10 11



| | | | | | | | | | | | |
|--|--|--|--|--|---|---|---|---|---|---|---|
| | | | | | 9 | 0 | 7 | 4 | 5 | 2 | 6 |
|--|--|--|--|--|---|---|---|---|---|---|---|

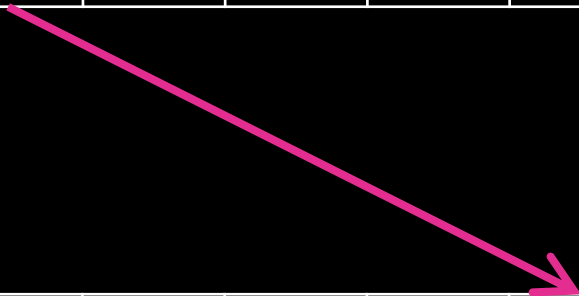


| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|
| | 1 | | 3 | | | | | 8 | | 10 | 11 |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |

| | | | | | | | | | | | |
|--|--|--|--|--|--|---|---|---|---|---|---|
| | | | | | | 0 | 7 | 4 | 5 | 2 | 6 |
|--|--|--|--|--|--|---|---|---|---|---|---|

| | | | | | | | | | | | |
|--|---|--|---|--|--|--|--|---|---|----|----|
| | 1 | | 3 | | | | | 8 | 9 | 10 | 11 |
|--|---|--|---|--|--|--|--|---|---|----|----|

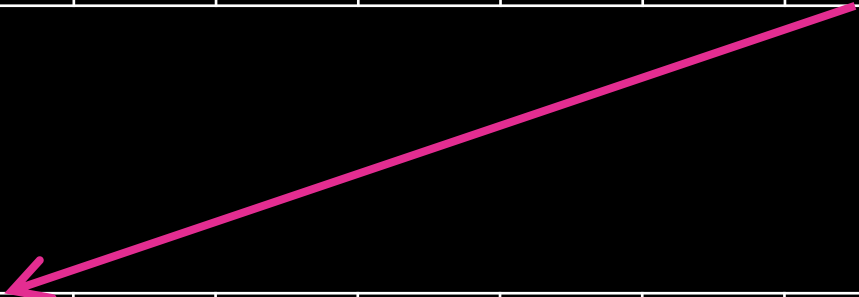
0 1 2 3 4 5 6 7 8 9 10 11



| | | | | | | | | | | | |
|--|--|--|--|--|--|--|---|---|---|---|---|
| | | | | | | | 7 | 4 | 5 | 2 | 6 |
|--|--|--|--|--|--|--|---|---|---|---|---|

| | | | | | | | | | | | |
|---|---|--|---|--|--|--|--|---|---|----|----|
| 0 | 1 | | 3 | | | | | 8 | 9 | 10 | 11 |
|---|---|--|---|--|--|--|--|---|---|----|----|

0 1 2 3 4 5 6 7 8 9 10 11

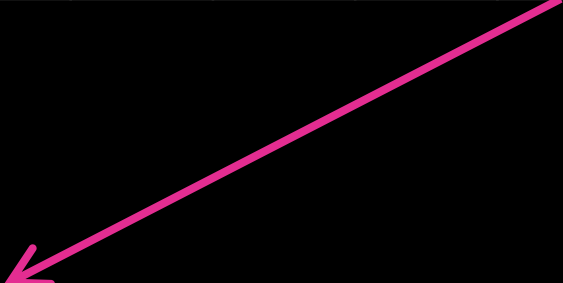
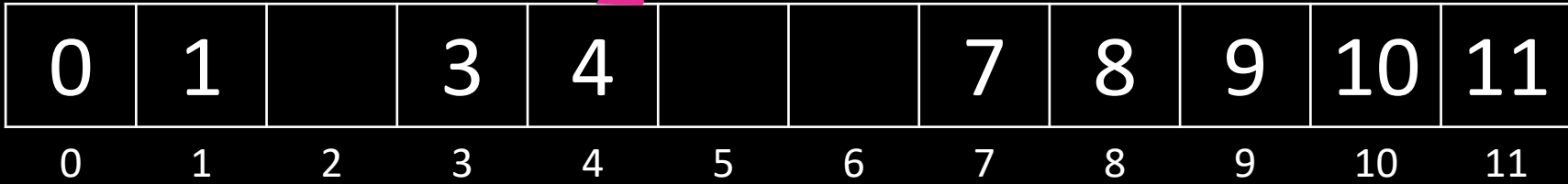
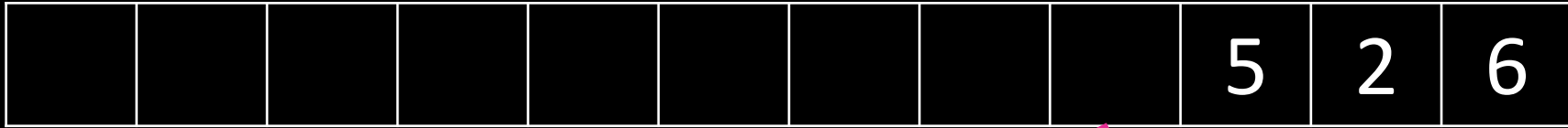


| | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|---|---|---|---|
| | | | | | | | | 4 | 5 | 2 | 6 |
|--|--|--|--|--|--|--|--|---|---|---|---|



| | | | | | | | | | | | |
|---|---|--|---|--|--|--|---|---|---|----|----|
| 0 | 1 | | 3 | | | | 7 | 8 | 9 | 10 | 11 |
|---|---|--|---|--|--|--|---|---|---|----|----|

0 1 2 3 4 5 6 7 8 9 10 11



| | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|---|---|
| | | | | | | | | | | 2 | 6 |
|--|--|--|--|--|--|--|--|--|--|---|---|

| | | | | | | | | | | | |
|---|---|--|---|---|---|--|---|---|---|----|----|
| 0 | 1 | | 3 | 4 | 5 | | 7 | 8 | 9 | 10 | 11 |
|---|---|--|---|---|---|--|---|---|---|----|----|

0

1

2

3

4

5

6

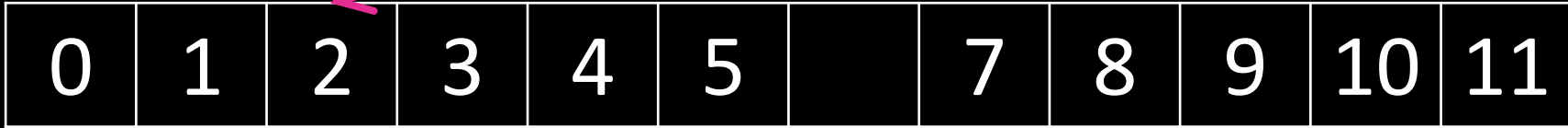
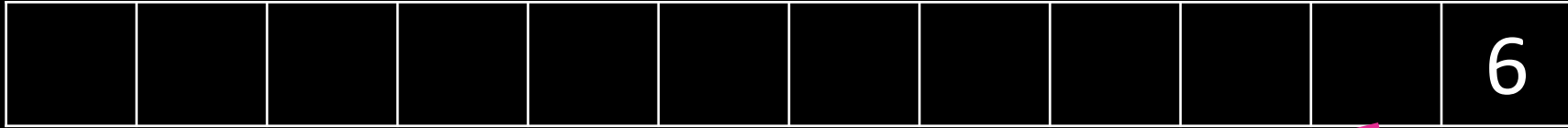
7

8

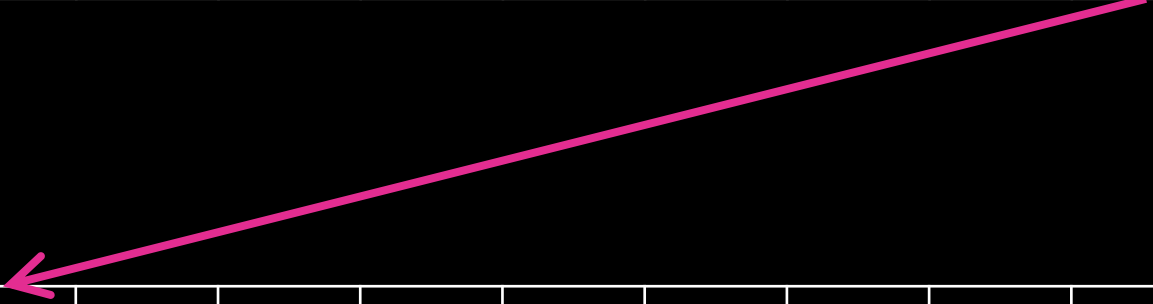
9

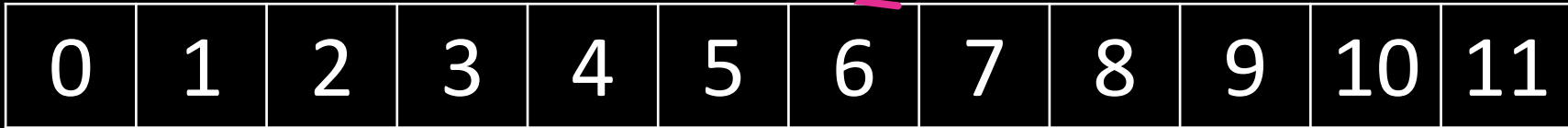
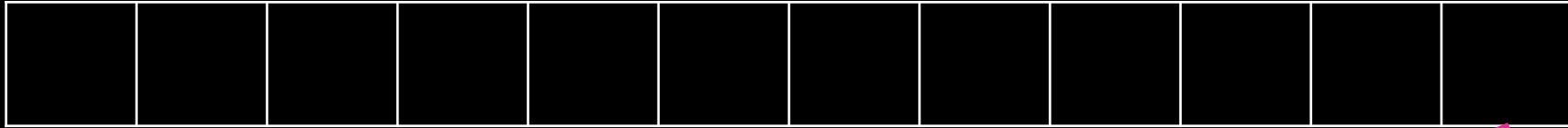
10

11

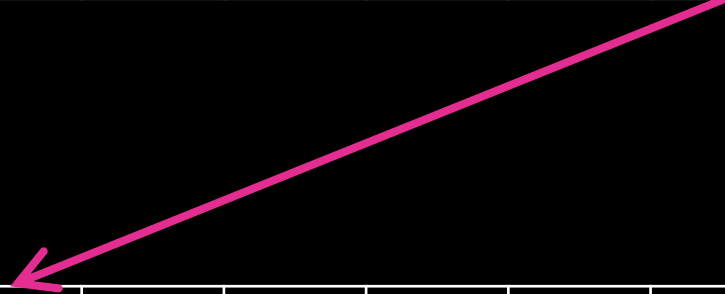


0 1 2 3 4 5 6 7 8 9 10 11





0 1 2 3 4 5 6 7 8 9 10 11



| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 |
|---|---|---|---|---|---|---|---|---|---|----|----|

| | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|--|--|
| | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|--|--|

0 1 2 3 4 5 6 7 8 9 10 11

Se recorre la estructura moviendo los elementos a la lista original.

Versión con más libertad en las llaves

- En lugar de requerir que los elementos sean una permutación de los valores $0..n-1$, se acepta que estén en un rango $0..\text{maxKeyValue}$
- La estructura de baldes tiene un espacio para cada posible valor de las llaves, aunque no necesariamente se vaya a utilizar
- Se recorren los elementos y se insertan en el balde correspondiente
- Se recolectan los elementos y se ubican en la lista original

| | | | | | | | | | |
|---|----|----|---|---|----|---|---|----|----|
| 7 | 16 | 13 | 6 | 0 | 18 | 3 | 2 | 10 | 11 |
|---|----|----|---|---|----|---|---|----|----|

Los elementos se encuentran en el rango 0-19, pero no es necesario que estén todos los valores.

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

| | | | | | | | | | |
|---|----|----|---|---|----|---|---|----|----|
| 7 | 16 | 13 | 6 | 0 | 18 | 3 | 2 | 10 | 11 |
|---|----|----|---|---|----|---|---|----|----|

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| | | | | | | | | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

Los baldes tienen un espacio para cada posible valor de la llave.

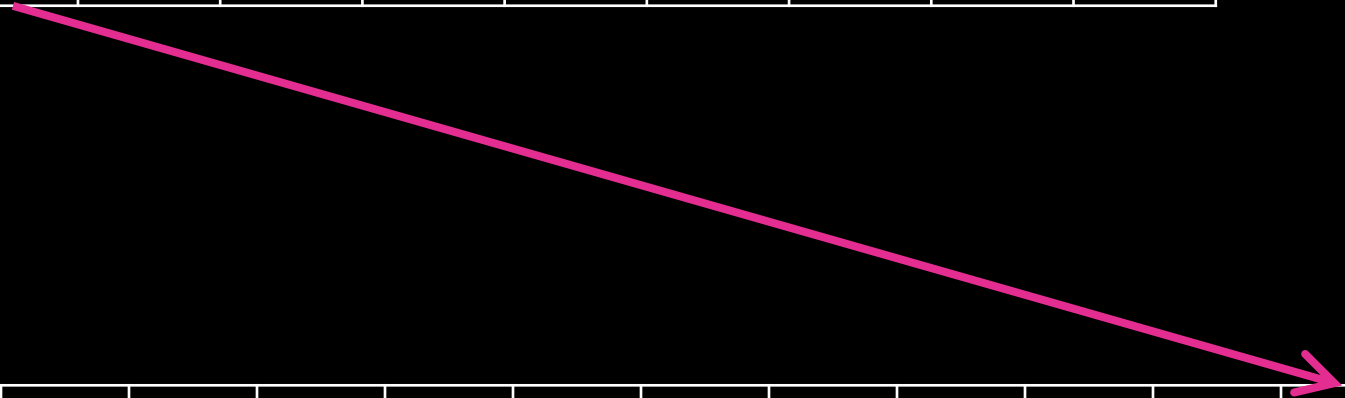
| | | | | | | | | | |
|--|----|----|---|---|----|---|---|----|----|
| | 16 | 13 | 6 | 0 | 18 | 3 | 2 | 10 | 11 |
|--|----|----|---|---|----|---|---|----|----|

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| | | | | | | | 7 | | | | | | | | | | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

Cada elemento se utiliza como índice para ubicarlo en el la estructura.

| | | | | | | | | | |
|--|--|----|---|---|----|---|---|----|----|
| | | 13 | 6 | 0 | 18 | 3 | 2 | 10 | 11 |
|--|--|----|---|---|----|---|---|----|----|

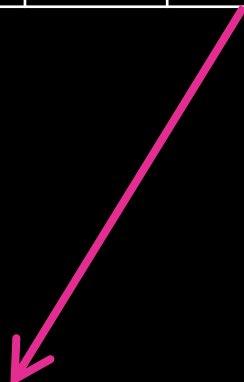
| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| | | | | | | | 7 | | | | | | | | | 16 | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |



| | | | | | | | | | |
|--|--|--|---|---|----|---|---|----|----|
| | | | 6 | 0 | 18 | 3 | 2 | 10 | 11 |
|--|--|--|---|---|----|---|---|----|----|

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| | | | | | | | 7 | | | | | | 13 | | | 16 | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

| | | | | | | | | | |
|--|--|--|--|---|----|---|---|----|----|
| | | | | 0 | 18 | 3 | 2 | 10 | 11 |
|--|--|--|--|---|----|---|---|----|----|



| | | | | | | | | | | | | | | | | | | | | |
|--|--|--|--|--|--|---|---|--|--|--|--|--|--|----|--|--|----|--|--|--|
| | | | | | | 6 | 7 | | | | | | | 13 | | | 16 | | | |
|--|--|--|--|--|--|---|---|--|--|--|--|--|--|----|--|--|----|--|--|--|

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

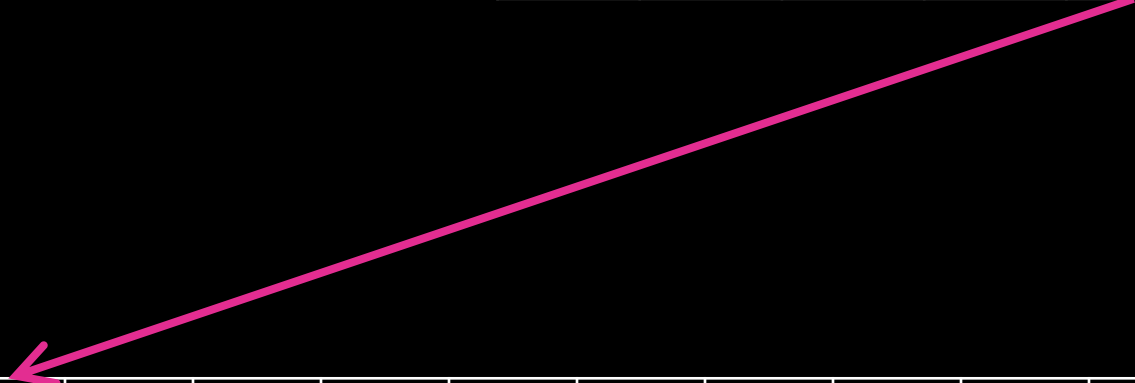
17

18

19

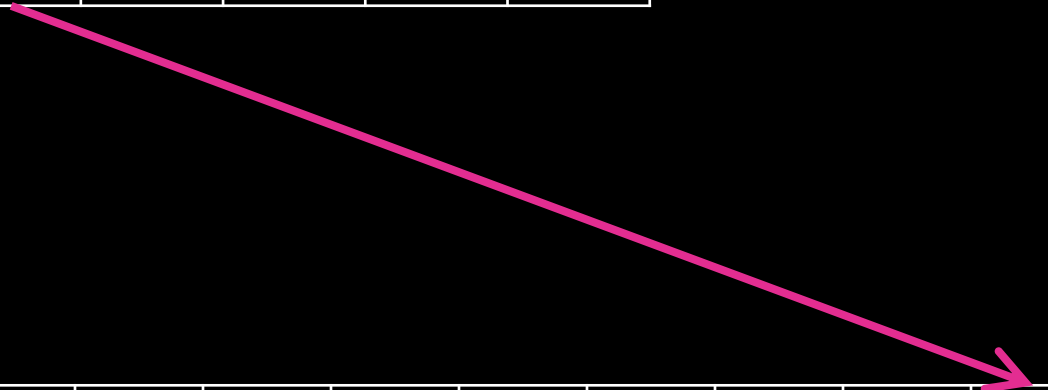
| | | | | | | | | | |
|--|--|--|--|--|----|---|---|----|----|
| | | | | | 18 | 3 | 2 | 10 | 11 |
|--|--|--|--|--|----|---|---|----|----|

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | | | | | | 6 | 7 | | | | | | 13 | | | 16 | | | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |



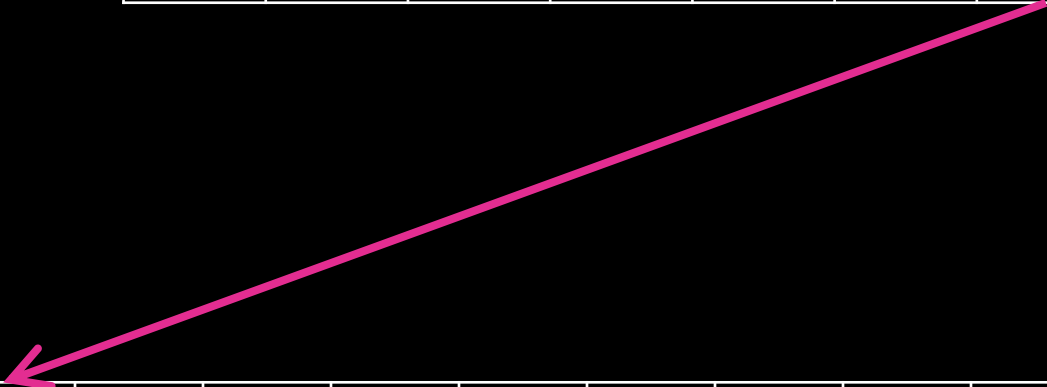
| | | | | | | | | | |
|--|--|--|--|--|--|---|---|----|----|
| | | | | | | 3 | 2 | 10 | 11 |
|--|--|--|--|--|--|---|---|----|----|

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | | | | | | 6 | 7 | | | | | | 13 | | | 16 | | 18 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |



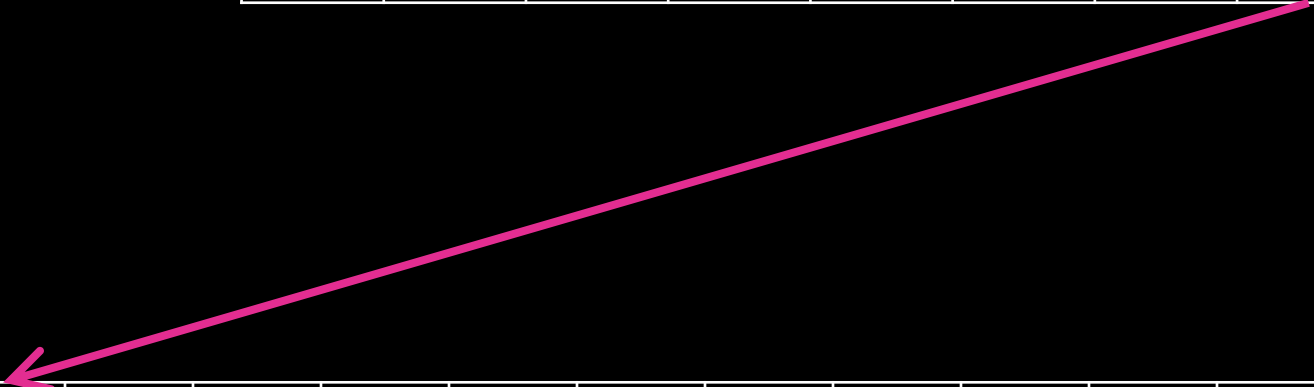
| | | | | | | | | | |
|--|--|--|--|--|--|--|---|----|----|
| | | | | | | | 2 | 10 | 11 |
|--|--|--|--|--|--|--|---|----|----|

| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | | | 3 | | | 6 | 7 | | | | | | 13 | | | 16 | | 18 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

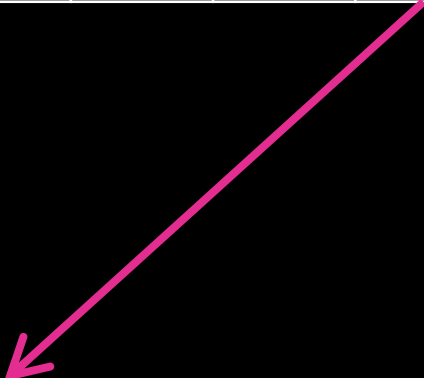


| | | | | | | | | | |
|--|--|--|--|--|--|--|--|----|----|
| | | | | | | | | 10 | 11 |
|--|--|--|--|--|--|--|--|----|----|

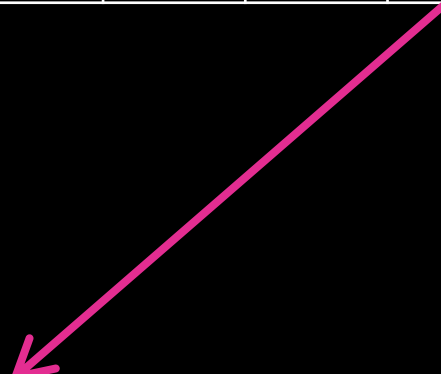
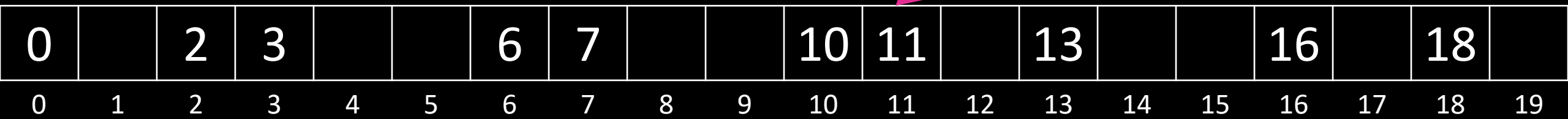
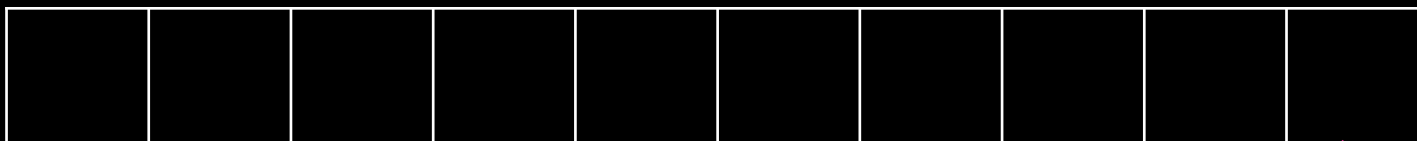
| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | | 2 | 3 | | | 6 | 7 | | | | | | 13 | | | 16 | | 18 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

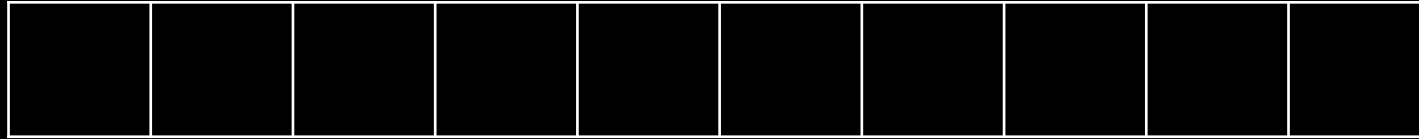


| | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|----|
| | | | | | | | | | 11 |
|--|--|--|--|--|--|--|--|--|----|



| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | | 2 | 3 | | | 6 | 7 | | | 10 | | | 13 | | | 16 | | 18 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |





| | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|
| 0 | | 2 | 3 | | | 6 | 7 | | | 10 | 11 | | 13 | | | 16 | | 18 | |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |

Se procede a recolectar los valores de la estructura de baldes. El inconveniente de esta versión consiste en que es necesario preguntar si en cada balde hay o no un elemento, lo cual degrada al algoritmo. Otro inconveniente es que se requiere mucha memoria para la estructura si el rango de valores es muy amplio.

| | | | | | | | | | |
|---|---|---|---|---|----|----|----|----|----|
| 0 | 2 | 3 | 6 | 7 | 10 | 11 | 13 | 16 | 18 |
|---|---|---|---|---|----|----|----|----|----|



| | | | | | | | | | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|
| | | | | | | | | | | | | | | | | | | | |
|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|--|

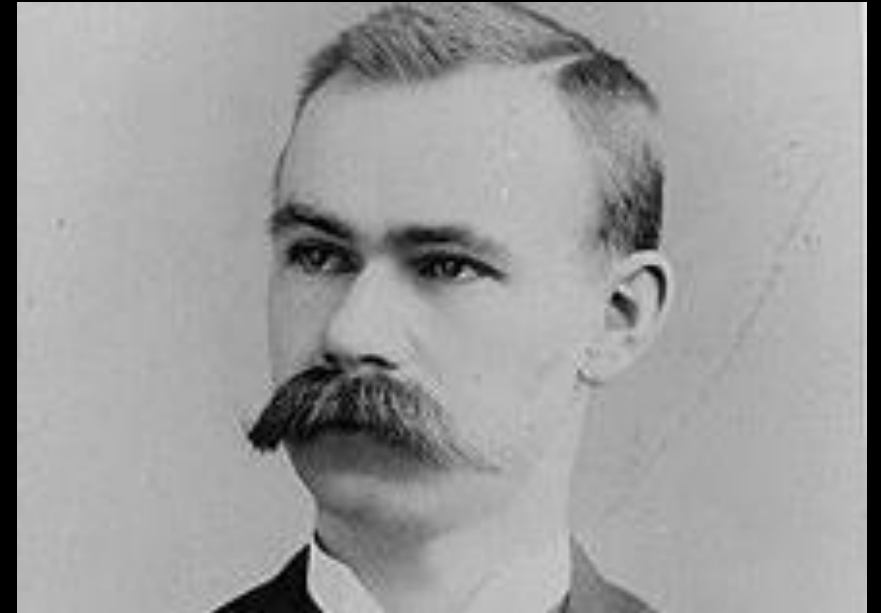
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Radix Sort

- Algoritmo no basado en comparaciones
- Requiere que los valores a ordenar sean números enteros
- Se basa en una notación posicional
- Puede adaptarse para utilizar tipos de datos que no sean enteros
- Estable
- Es un caso especial del Bin Sort
- También utiliza una estructura de baldes

Historia

- Fue utilizado por **Herman Hollerith** desde 1887 para ordenar tarjetas perforadas
- Desarrolló un mecanismo electromecánico que podía **ordenar tarjetas** de acuerdo a los agujeros que tenían en diferentes columnas
- Esto lo utilizó para compilar **estadísticas** y **procesar datos** en general
- La primera versión programada fue creada por **Harold H. Seward** en 1954 en el MIT
- La popularidad del algoritmo ha variado a través del tiempo, hoy es considerado una **alternativa de alto desempeño** a los algoritmos basados en comparaciones



Pasos

Es necesario conocer la cantidad máxima de dígitos que tienen los elementos a ordenar, de eso depende la cantidad de veces que se repite el procedimiento.

Los elementos pueden analizarse desde la perspectiva de cualquier base numérica, la cual puede parametrizarse y determina la cantidad de baldes a utilizar.

01

Se recorren los elementos a ordenar analizando uno de sus dígitos. Se empieza por el menos significativo

02

Según el valor del dígito, se elimina de la lista y se inserta en el balde correspondiente

03

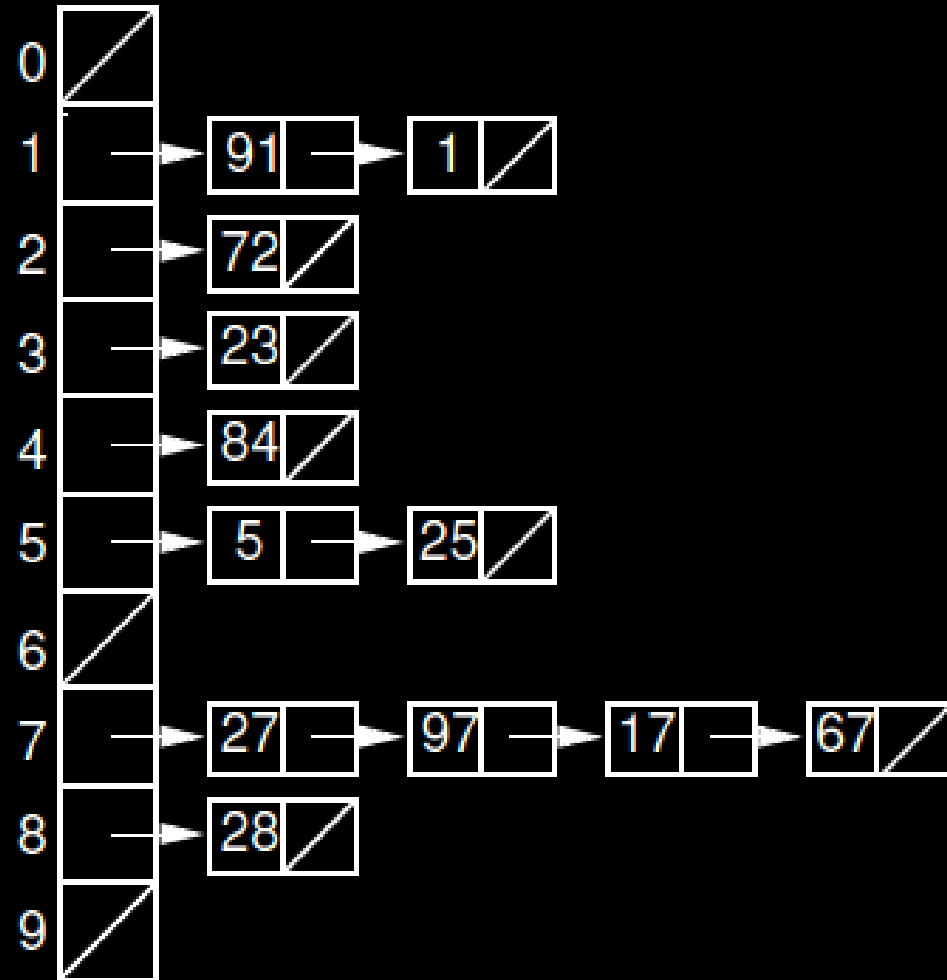
Una vez ubicados todos los elementos en su balde, se recolectan en orden ubicándolos en la lista original

04

Se repiten los pasos con cada uno de los dígitos hasta llegar al más significativo

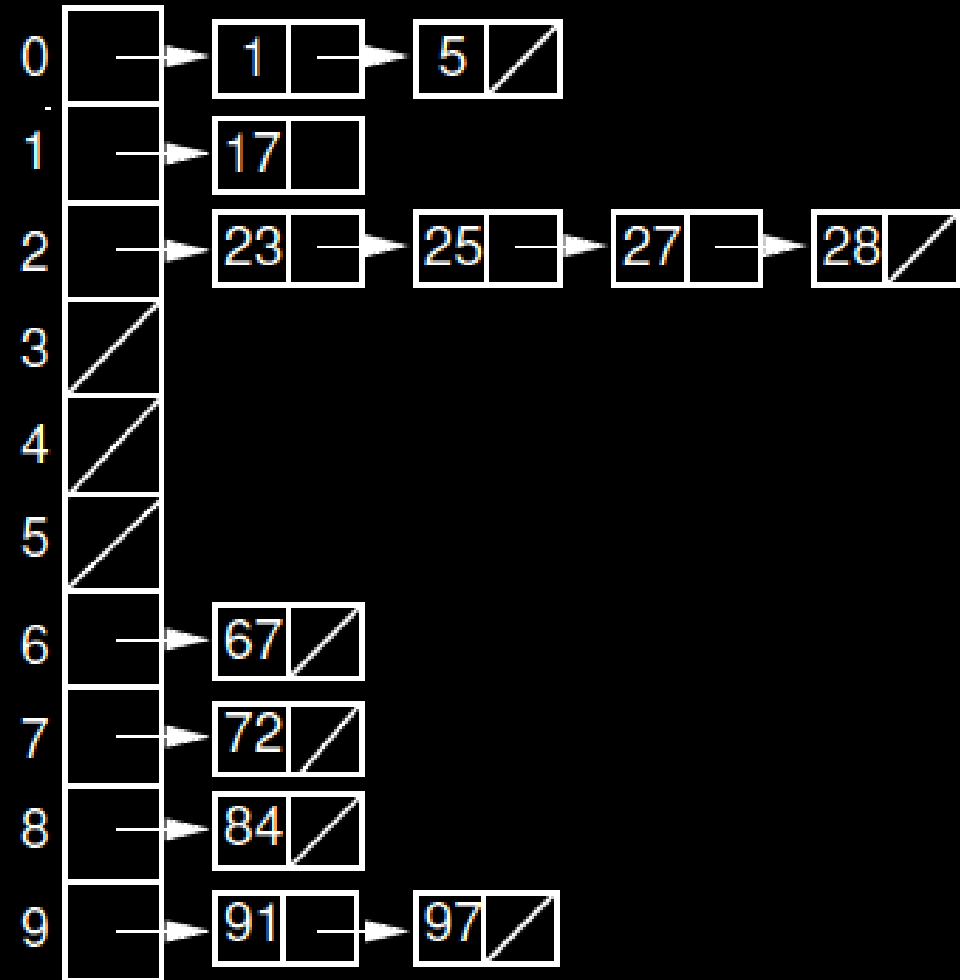
Initial List: 27 91 1 97 17 23 84 28 72 5 67 25

First pass
(on right digit)



Result of first pass: 91 1 72 23 84 5 25 27 97 17 67 28

Second pass
(on left digit)



Result of second pass: 1 5 17 23 25 27 28 67 72 84 91 97

Ejemplo de implementación

- **Método** implementado en la clase **ArrayList**
- Trabaja sobre el arreglo **elements**
- Se utiliza una función auxiliar que calcula la **cantidad de iteraciones** que deben hacerse en el algoritmo
- Esto puede recibirse por **parámetro** en lugar de calcularlo cada vez si se conocen de antemano los posibles valores de los elementos a ordenar
- El ordenamiento funciona solamente si la plantilla E es de tipo **entero**

Recibe por parámetro la base numérica a utilizar.

```
void radixSort(int radix) {
    int digits = getMaxDigits(radix);
    LinkedList<E> *buckets = new LinkedList<E>[radix];
    for (int iter = 0; iter < digits; iter++) {
        for (int i = 0; i < size; i++) {
            int pos = int(elements[i] / pow(radix, iter)) % radix;
            buckets[pos].append(elements[i]);
        }
        int i = 0;
        for (int pos = 0; pos < radix; pos++) {
            while (buckets[pos].getSize() > 0) {
                elements[i] = buckets[pos].remove();
                i++;
            }
        }
    }
    delete [] buckets;
}
```


Se calcula la cantidad máxima de dígitos en los valores a ordenar. Este valor podría recibirse por parámetro.

```
void radixSort(int radix) {
    int digits = getMaxDigits(radix);
    LinkedList<E> *buckets = new LinkedList<E>[radix];
    for (int iter = 0; iter < digits; iter++) {
        for (int i = 0; i < size; i++) {
            int pos = int(elements[i] / pow(radix, iter)) % radix;
            buckets[pos].append(elements[i]);
        }
        int i = 0;
        for (int pos = 0; pos < radix; pos++) {
            while (buckets[pos].getSize() > 0) {
                elements[i] = buckets[pos].remove();
                i++;
            }
        }
    }
    delete [] buckets;
}
```

```
void radixSort(int radix) {
    int digits = getMaxDigits(radix);
    LinkedList<E> *buckets = new LinkedList<E>[radix];
    for (int iter = 0; iter < digits; iter++) {
        for (int i = 0; i < size; i++) {
            int pos = int(elements[i] / pow(radix, iter)) % radix;
            buckets[pos].append(elements[i]);
        }
        int i = 0;
        for (int pos = 0; pos < radix; pos++) {
            while (buckets[pos].getSize() > 0) {
                elements[i] = buckets[pos].remove();
                i++;
            }
        }
    }
    delete [] buckets;
}
```

Creación de la estructura de baldes. Uno por cada dígito presente en la base numérica a utilizar.

```
void radixSort(int radix) {
    int digits = getMaxDigits(radix);
    LinkedList<E> *buckets = new LinkedList<E>[radix];
    for (int iter = 0; iter < digits; iter++) {
        for (int i = 0; i < size; i++) {
            int pos = int(elements[i] / pow(radix, iter)) % radix;
            buckets[pos].append(elements[i]);
        }
        int i = 0;
        for (int pos = 0; pos < radix; pos++) {
            while (buckets[pos].getSize() > 0) {
                elements[i] = buckets[pos].remove();
                i++;
            }
        }
    }
    delete [] buckets;
}
```

Ciclo que se repite por cada dígito presente en los elementos a ordenar.

```

void radixSort(int radix) {
    int digits = getMaxDigits(radix);
    LinkedList<E> *buckets = new LinkedList<E>[radix];
    for (int iter = 0; iter < digits; iter++) {
        for (int i = 0; i < size; i++) {
            int pos = int(elements[i] / pow(radix, iter)) % radix;
            buckets[pos].append(elements[i]);
        }
        int i = 0;
        for (int pos = 0; pos < radix; pos++) {
            while (buckets[pos].getSize() > 0) {
                elements[i] = buckets[pos].remove();
                i++;
            }
        }
    }
    delete [] buckets;
}


```

Se recorre el arreglo de elementos obteniendo el dígito en la posición de interés según el número de iteración. Se utiliza ese dígito para seleccionar el balde respectivo y agregarlo en la lista.

```
void radixSort(int radix) {  
    int digits = getMaxDigits(radix);  
    LinkedList<E> *buckets = new LinkedList<E>[radix];  
    for (int iter = 0; iter < digits; iter++) {  
        for (int i = 0; i < size; i++) {  
            int pos = int(elements[i] / pow(radix, iter)) % radix;  
            buckets[pos].append(elements[i]);  
        }  
        int i = 0;  
        for (int pos = 0; pos < radix; pos++) {  
            while (buckets[pos].getSize() > 0) {  
                elements[i] = buckets[pos].remove();  
                i++;  
            }  
        }  
    }  
    delete [] buckets;  
}
```

Una vez que se recorrieron todos los elementos. Se recorren los baldes.

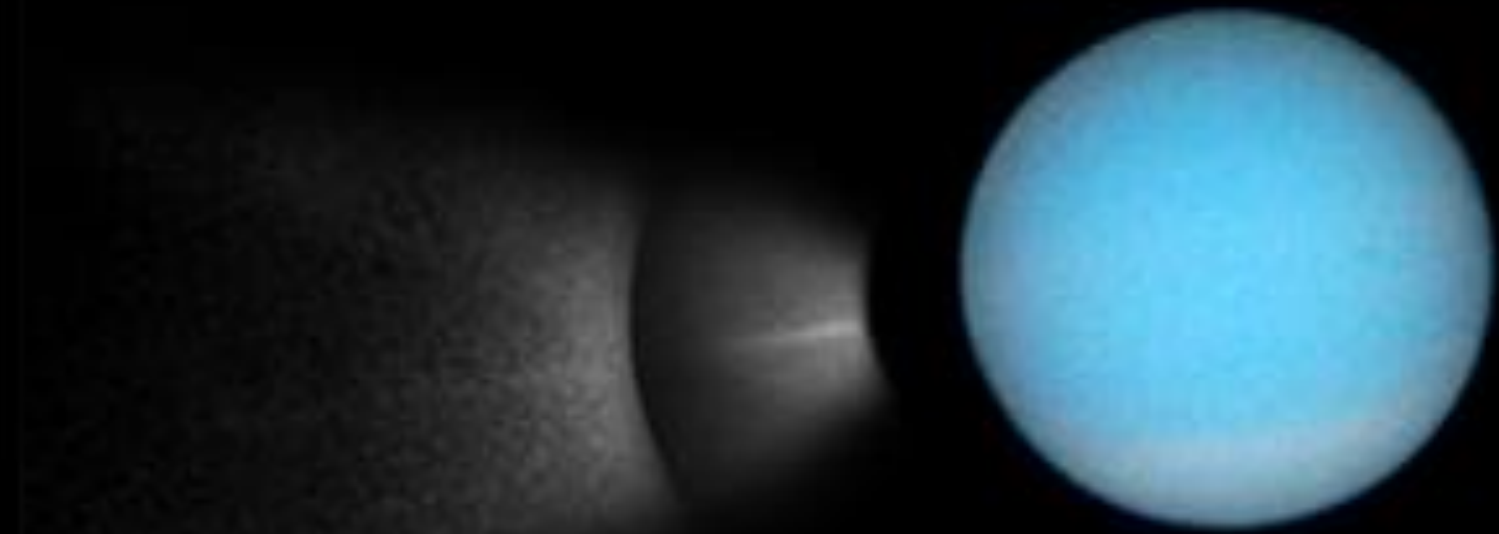
```
void radixSort(int radix) {
    int digits = getMaxDigits(radix);
    LinkedList<E> *buckets = new LinkedList<E>[radix];
    for (int iter = 0; iter < digits; iter++) {
        for (int i = 0; i < size; i++) {
            int pos = int(elements[i] / pow(radix, iter)) % radix;
            buckets[pos].append(elements[i]);
        }
        int i = 0;
        for (int pos = 0; pos < radix; pos++) {
            while (buckets[pos].getSize() > 0) {
                elements[i] = buckets[pos].remove();
                i++;
            }
        }
    }
    delete [] buckets;
}
```



Cada elemento que se encuentra en cada balde se agrega a la lista original.

```
int getMaxDigits(int radix) {  
    int greater = elements[0];  
    int maxDigits = 0;  
    for (int i = 1; i < size; i++)  
        if (greater < elements[i])  
            greater = elements[i];  
    while (greater > 0) {  
        greater /= radix;  
        maxDigits++;  
    }  
    return maxDigits;  
}
```

Función auxiliar que recorre el arreglo de elementos buscando el elemento mayor para determinar la cantidad de dígitos que tiene.



Ordenamientos y Búsquedas

Mauricio Avilés