Algoritmos de ordenamiento

Estructuras de Datos y Algoritmos /
Algoritmos y Estructuras de Datos II
Año 2025

Dr. Pablo Ponzio
Universidad Nacional de Río Cuarto
CONICET

Sorting

Sorting es la tarea de organizar una colección de datos según un orden dado

- Podemos realizar sorting sobre cualquier tipo de datos con un orden: **int**, **char**, **Integer**, **Strings**, etc
- En general, el algoritmo de sorting acomoda los elementos de forma ascendente o descendente
- Es importante que los algoritmos de sorting sean **eficientes** ya que en la práctica se quiere ordenar una cantidad grande de elementos

Sorting en JAVA

En Java se utiliza la interface **Comparable**:

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

- Toda clase con un orden hereda de comparable.
- Permite implementar algoritmos de sorting polimórficos.
- La clase comparable provee un método CompareTo():

this.compareTo(x) > 0 ≡ this > x

this.compareTo(x) < 0 ≡ this < x

this.compareTo(x) = 0 ≡ this = x

usualmente compareTo(x) devuelve -1, 0, 1

Comparable: Ejemplo

```java
public class Employee implements Comparable<Employee> {
    String firstName;
    String lastName;

    /**
    * @post Compares this object with the specified object for order.
    *   Returns a negative integer, zero, or a positive integer as this
    *   object is less than, equal to, or greater than the specified object.
    */
    public int compareTo(Employee other)
    {
        if (this.lastName.compareTo(other.lastName) != 0) {
          // If lastNames are different, compare lastName
          return this.lastName.compareTo(other.lastName);
        } else {
          // If lastNames are the same, compare firstName
          return this.firstName.compareTo(other.firstName);
        }
    }
}
```

Importante…

Para analizar un algoritmo de sorting podemos tener en cuenta:

- **Eficiencia:** el tiempo de ejecución del algoritmo, en el peor caso, y también en el caso promedio.

- **Cantidad de comparaciones:** Cuantas veces comparamos para ordenar los elementos.

Las comparaciones pueden ser costosas

- **Cantidad de Intercambios:** Cuantos intercambios se realizan para ordenar

Los intercambios son constantes en JAVA

Estabilidad

Un algoritmo de sorting se dice estable si preserva el orden de los elementos con las mismas claves

- La clave es el campo o el atributo sobre el cual ordenamos.

- Hay algoritmos que son estables y otros no.

- En general cualquier algoritmo se puede hacer estable con algún costo extra: agregar más claves, etc.

Clase base para Sorting

```java
/**
 * Base class for sorting algorithms.
 */
public abstract class Sorting
{
    /**
     * @post Rearranges the array a in ascending order.
     */
    public abstract void sort(Comparable[] a);


    /**
     * @post Returns true iff v is smaller than w.
     */
    protected boolean less(Comparable v, Comparable w) {
        return v.compareTo(w) < 0;
    }

    /**
     * @pre 0<=i<a.length && 0<=j<a.length
     * @post Swaps the elements in positions i and j of a.
     */
    protected static void exch(Object[] a, int i, int j) {
        Object swap = a[i];
        a[i] = a[j];
        a[j] = swap;
    }
```

Selection sort

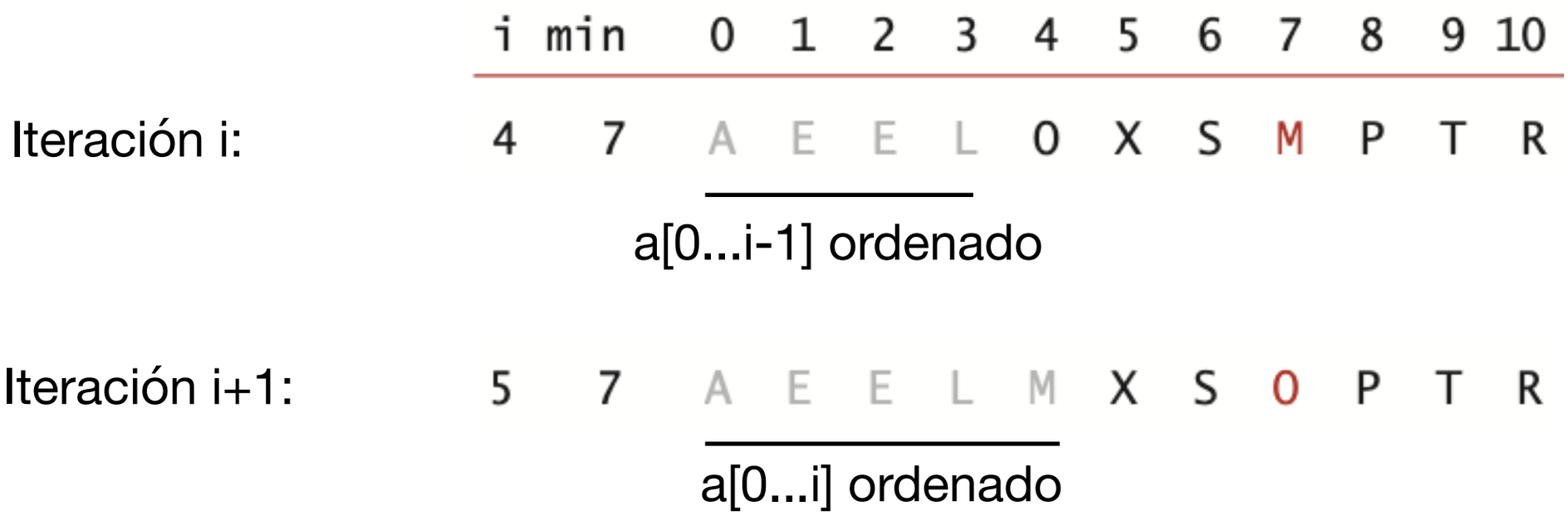
- Elegir el elemento más chico del arreglo e intercambiarlo con el elemento en la primera posición
- Elegir el segundo elemento más chico del arreglo e intercambiarlo con el elemento en la segunda posición
- y así sucesivamente...

		a[]										
i	min	0	1	2	3	4	5	6	7	8	9	10
		S	O	R	T	E	X	A	M	P	L	E
0	6	S	O	R	T	E	X	A	M	P	L	E
1	4	A	O	R	T	E	X	S	M	P	L	E
2	10	A	E	R	T	O	X	S	M	P	L	E
3	9	A	E	E	T	O	X	S	M	P	L	R
4	7	A	E	E	L	O	X	S	M	P	T	R

...

Selection sort

- Invariante: Selection sort mantiene ordenado el arreglo a[0...i-1]
- En cada iteración, intercambia el elemento en i por el elemento más chico en a[i...n-1]
  - De esta manera, se obtiene que el arreglo a[0...i] está ordenado



- Encapsular los algoritmos en objetos nos va a permitir intercambiar las implementaciones de los algoritmos de Sorting sin modificar los clientes
  - Esto se corresponde con el patrón de diseño llamado Strategy
- Para ordenar un arreglo con selection sort crearemos un objeto de la clase Selection:
  - Sorting s = new Selection(); s.sort(arr);
- Para ordenar con insertion sort ejecutaremos:
  - Sorting s = new Insertion(); s.sort(arr);

```java
public class Selection extends Sorting {

    /**
     * @post Creates an object that encapsulates the selection
     *   sort algorithm.
     */
    public Selection() { }

    /**
     * @post Rearranges the array a in ascending order
     *   using the selection sort algorithm.
     */
    public void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            // Invariant: a[0...i-1] is sorted
            assert isSorted(a, 0, i-1);
            // Exchange a[i] with smallest entry in a[i...n-1]
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(a[j], a[min]))
                    min = j;
            }
            exch(a, i, min);
            // Now a[0...i] is sorted
            assert isSorted(a, 0, i);
        }
        assert isSorted(a);
    }
}
```

- Encapsular los algoritmos en objetos nos va a permitir intercambiar las implementaciones de los algoritmos de Sorting sin modificar los clientes
  - Esto se corresponde con el patrón de diseño llamado Strategy
- Para ordenar un arreglo con selection sort crearemos un objeto de la clase Selection:
  - Sorting s = new Selection();
-

```java
public class Selection extends Sorting {

    /**
     * @post Creates an object that encapsulates the selection
     *   sort algorithm.
     */
    public Selection() { }

    /**
     * @post Rearranges the array a in ascending order
     *   using the selection sort algorithm.
     */
    public void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            // Invariant: a[0...i-1] is sorted
            assert isSorted(a, 0, i-1);
            // Exchange a[i] with smallest entry in a[i...n-1]
            int min = i;
            int j = i+1; j < n; j++) {
            (less(a[j], a[min]))
                min = j;

            a, i, min);
            a[0...i] is sorted
            isSorted(a, 0, i);

            Sorted(a);

    }
```

```java
@Test
public void test0()
{
    Integer [] arr = new Integer [] {3,2,1};
    Integer [] expected = new Integer [] {1,2,3};
    Sorting s = new Selection();
    s.sort(arr);
    assertTrue(Arrays.equals(arr, expected));
    assertTrue(s.isSorted(arr));
}
```

- Encapsular los algoritmos en objetos nos va a permitir intercambiar las implementaciones de los algoritmos de Sorting sin modificar los clientes
  - Esto se corresponde con el patrón de diseño llamado Strategy
- Para ordenar un arreglo con selection sort crearemos un objeto de la clase Selection:
  - Sorting s = new Selection();
-

```java
public class Selection extends Sorting {

    /**
     * @post Creates an object that encapsulates the selection
     *   sort algorithm.
     */
    public Selection() { }

    /**
     * @post Rearranges the array a in ascending order
     *   using the selection sort algorithm.
     */
    public void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            // Invariant: a[0...i-1] is sorted
            assert isSorted(a, 0, i-1);
            // Exchange a[i] with smallest entry in a[i...n-1]
            int min = i;
            int j = i+1; j < n; j++) {
```

```java
@Test
public void test0()
{
    Integer [] arr = new Integer []
    Integer [] expected = new Intege
    Sorting s = new Selection();
    s.sort(arr);
    assertTrue(Arrays.equals(arr, ex
    assertTrue(s.isSorted(arr));
}
```

```java
@Test
public void test1()
{
    Integer [] arr = new Integer [] {2,3,1,5,6,9,6,3};
    Integer [] expected = new Integer [] {1,2,3,3,5,6,6,9};
    Sorting s = new Selection();
    s.sort(arr);
    assertTrue(Arrays.equals(arr, expected));
    assertTrue(s.isSorted(arr));
}
```

Selection sort: Características

- El peor caso es cuando el arreglo está ordenado en forma decreciente
- El tiempo de ejecución en el peor caso es:
  - $T(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} c \quad \in \theta(n^2)$
- La cantidad de comparaciones que realiza en el peor caso es:
  - $\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 \quad \in \theta(n^2)$
- La cantidad de intercambios que realiza en el peor caso es:
  - $T(n) = \sum_{i=0}^{n-1} 1 \quad \in \theta(n)$
- Esta implementación es estable

```java
public void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
        // Invariant: a[0...i-1] is sorted
        assert isSorted(a, 0, i-1);
        // Exchange a[i] with smallest entry in a[i...n-1]
        int min = i;
        for (int j = i+1; j < n; j++) {
            if (less(a[j], a[min]))
                min = j;
        }
        exch(a, i, min);
        // Now a[0...i] is sorted
        assert isSorted(a, 0, i);
    }
    assert isSorted(a);
}
```

Bubble sort

- Intercambiar los pares de elementos adyacentes en a[n-1...0] que no estén ordenados
  - Esto deja el elemento más chico en a[0]
- Intercambiar los pares de elementos adyacentes en a[n-1...1] que no estén ordenados
  - Esto deja el segundo elemento más chico en a[1]
- y así sucesivamente...

[S, O, R, T, E, X, A, M, P, L, E]
[A, S, O, R, T, E, X, E, M, P, L]
[A, E, S, O, R, T, E, X, L, M, P]
[A, E, E, S, O, R, T, L, X, M, P]
[A, E, E, L, S, O, R, T, M, X, P]

Bubble sort

- Invariante: Bubble sort mantiene ordenado el arreglo a[0...i-1]
- En cada iteración, intercambia los pares de elementos adyacentes en a[n-1...i] que no están ordenados, y así logra que el menor de a[n-1...i] quede en a[i]
  - De esta manera, se obtiene que el arreglo a[0...i] está ordenado

Iteración i: [A, E, E, L, S, O, R, T, M, X, P]

a[0...i-1] ordenado

Iteración i+1: [A, E, E, L, M, S, O, R, T, P, X]

a[0...i] ordenado

```java
public class Bubble extends Sorting {

    /**
     * @post Creates an object that encapsulates the bubble
     *   sort algorithm.
     */
    public Bubble() { }

    /**
     * @post Rearranges the array a in ascending order
     *   using the bubble sort algorithm.
     */
    public void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
        // Invariant: a[0...i-1] is sorted
            int exchanges = 0;
            // Swap unsorted adjacent elements a[i...n-1]
            // This puts the smallest element from a[i...n-1] in a[i]
            for (int j = n-1; j > i; j--) {
                if (less(a[j], a[j-1])) {
                    exch(a, j, j-1);
                    exchanges++;
                }
            }
            // a[0...i] is now sorted
            assert isSorted(a, 0, i);
            if (exchanges == 0) break;
        }
        assert isSorted(a);
    }
}
```

Bubble sort: Tests

- Como están programados a partir de la misma clase abstracta Sorting, los tests son muy similares para todos los algoritmos de Sorting
- Esto indica que luego podremos cambiar fácilmente la implementación de los algoritmos de Sorting en los clientes (si están programados respecto de la clase abstracta Sorting)

```java
@Test
public void test0()
{
    Integer [] arr = new Integer [] {3,2,1};
    Integer [] expected = new Integer [] {1,2,3};
    Sorting s = new Selection();
    s.sort(arr);
    assertTrue(Arrays.equals(arr, expected));
    assertTrue(s.isSorted(arr));
}
```

```java
@Test
public void test4()
{
    Integer [] arr = new Integer [] {3,2,1};
    Integer [] expected = new Integer [] {1,2,3};
    Sorting s = new Bubble();
    s.sort(arr);
    assertTrue(Arrays.equals(arr, expected));
    assertTrue(s.isSorted(arr));
}
```

Bubble sort: Características

- El peor caso es cuando el arreglo está ordenado en forma decreciente
- El tiempo de ejecución en el peor caso es:
  - $T(n) = \sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} c \quad \in \theta(n^2)$
- La cantidad de comparaciones que realiza en el peor caso es:
  - $\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 \quad \in \theta(n^2)$
- La cantidad de intercambios que realiza en el peor caso es:
  - $\sum_{i=0}^{n-1} \sum_{j=i+1}^{n-1} 1 \quad \in \theta(n^2)$
- Esta implementación es estable

```java
public void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 0; i < n; i++) {
    // Invariant: a[0...i-1] is sorted
        int exchanges = 0;
        // Swap unsorted adjacent elements a[i...n-1]
        // This puts the smallest element from a[i...n-1] in a[i]
        for (int j = n-1; j > i; j--) {
            if (less(a[j], a[j-1])) {
                exch(a, j, j-1);
                exchanges++;
            }
        }
        // a[0...i] is now sorted
        assert isSorted(a, 0, i);
        if (exchanges == 0) break;
    }
    assert isSorted(a);
}
```

Insertion sort

- Insertar el elemento en la posición 1 ordenado en a[0...1]
- Insertar el elemento en la posición 2 ordenado en a[0...2]
- Insertar el elemento en la posición 3 ordenado en a[0...3]
- y así sucesivamente

[S, O, R, T, E, X, A, M, P, L, E]
[O, S, R, T, E, X, A, M, P, L, E]
[O, R, S, T, E, X, A, M, P, L, E]
[O, R, S, T, E, X, A, M, P, L, E]

...

Insertion sort

- Invariante: Insertion sort mantiene ordenado el arreglo a[0...i-1]
- En cada iteración, inserta el elemento en i de manera ordenada en a[0...i]
  - De esta manera, se obtiene que el arreglo a[0...i] está ordenado

Iteración i:

[O, R, S, T, E, X, A, M, P, L, E]

a[0...i-1] ordenado

Iteración i+1:

[E, O, R, S, T, X, A, M, P, L, E]

a[0...i] ordenado

- Encapsular los algoritmos en objetos nos va a permitir intercambiar las implementaciones de los algoritmos de Sorting sin modificar los clientes
  - Esto se corresponde con el patrón de diseño llamado Strategy
- Para ordenar un arreglo con selection sort crearemos un objeto de la clase Selection:
  - Sorting s = new Selection(); s.sort(arr);
- Para ordenar con insertion sort ejecutaremos:
  - Sorting s = new Insertion(); s.sort(arr);

```java
public class Selection extends Sorting {

    /**
     * @post Creates an object that encapsulates the selection
     *   sort algorithm.
     */
    public Selection() { }

    /**
     * @post Rearranges the array a in ascending order
     *   using the selection sort algorithm.
     */
    public void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            // Invariant: a[0...i-1] is sorted
            assert isSorted(a, 0, i-1);
            // Exchange a[i] with smallest entry in a[i...n-1]
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(a[j], a[min]))
                    min = j;
            }
            exch(a, i, min);
            // Now a[0...i] is sorted
            assert isSorted(a, 0, i);
        }
        assert isSorted(a);
    }
}
```

- Encapsular los algoritmos en objetos nos va a permitir intercambiar las implementaciones de los algoritmos de Sorting sin modificar los clientes
  - Esto se corresponde con el patrón de diseño llamado Strategy
- Para ordenar un arreglo con selection sort crearemos un objeto de la clase Selection:
  - Sorting s = new Selection(); s.sort(arr);

```
public class Selection extends Sorting {

    /**
     * @post Creates an object that encapsulates the selection
     *   sort algorithm.
     */
    public Selection() { }

    /**
     * @post Rearranges the array a in ascending order
     *   using the selection sort algorithm.
     */
    public void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            // Invariant: a[0...i-1] is sorted
            assert isSorted(a, 0, i-1);
            // Exchange a[i] with smallest entry in a[i...n-1]
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(a[j], a[min]))
                    min = j;
            }
            h(a, i, min);
            Now a[0...i] is sorted
            ert isSorted(a, 0, i);
        }
        isSorted(a);
    }
```

```
@Test
public void test0()
{
    Integer [] arr = new Integer [] {3,2,1};
    Integer [] expected = new Integer [] {1,2,3};
    Sorting s = new Selection();
    s.sort(arr);
    assertTrue(Arrays.equals(arr, expected));
    assertTrue(s.isSorted(arr));
}
```

- Encapsular los algoritmos en objetos nos va a permitir intercambiar las implementaciones de los algoritmos de Sorting sin modificar los clientes
  - Esto se corresponde con el patrón de diseño llamado Strategy
- Para ordenar un arreglo con selection sort crearemos un objeto de la clase Selection:
  - Sorting s = new Selection(); s.sort(arr);

```java
public class Selection extends Sorting {

    /**
     * @post Creates an object that encapsulates the selection
     *   sort algorithm.
     */
    public Selection() { }

    /**
     * @post Rearranges the array a in ascending order
     *   using the selection sort algorithm.
     */
    public void sort(Comparable[] a) {
        int n = a.length;
        for (int i = 0; i < n; i++) {
            // Invariant: a[0...i-1] is sorted
            assert isSorted(a, 0, i-1);
            // Exchange a[i] with smallest entry in a[i...n-1]
            int min = i;
            for (int j = i+1; j < n; j++) {
                if (less(a[j], a[min]))
```

```java
@Test
public void test0()
{
    Integer [] arr = new Integer [] {3,2,1};
    Integer [] expected = new Integer [] {1,2,3
    Sorting s = new Selection();
    s.sort(arr);
    assertTrue(Arrays.equals(arr, expected));
    assertTrue(s.isSorted(arr));
}
```

```java
@Test
public void test4()
{
    Integer [] arr = new Integer [] {3,2,1};
    Integer [] expected = new Integer [] {1,2,3};
    Sorting s = new Bubble();
    s.sort(arr);
    assertTrue(Arrays.equals(arr, expected));
    assertTrue(s.isSorted(arr));
}
```

Insertion sort: Características

- El peor caso es cuando el arreglo está ordenado en forma decreciente
- El tiempo de ejecución en el peor caso es:
  - $T(n) = \sum_{i=0}^{n-1} \sum_{j=1}^{i} c \quad \in \theta(n^2)$
- La cantidad de comparaciones que realiza en el peor caso es:
  - $\sum_{i=0}^{n-1} \sum_{j=1}^{i} 1 \quad \in \theta(n^2)$
- La cantidad de intercambios que realiza en el peor caso es:
  - $\sum_{i=0}^{n-1} \sum_{j=1}^{i} 1 \quad \in \theta(n^2)$

```java
public void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 1; i < n; i++) {
        // Invariant: a[0...i-1] is sorted
        assert isSorted(a, 0, i-1);
        // Insert a[i] in sorted order in a[0...i]
        for (int j = i; j > 0 && less(a[j], a[j-1]); j--) {
            exch(a, j, j-1);
        }
        // a[0...i] is now sorted
        assert isSorted(a, 0, i);
    }
    assert isSorted(a);
}
```

Insertion sort: Características

- Esta implementación es estable
- En el mejor caso es:
  - $T(n) = \sum_{i=0}^{n-1} c \quad \in \theta(n)$
- En el caso promedio es cuadrático, y se lo considera ineficiente para ser usado en la práctica

```
public void sort(Comparable[] a) {
    int n = a.length;
    for (int i = 1; i < n; i++) {
    // Invariant: a[0...i-1] is sorted
        assert isSorted(a, 0, i-1);
        // Insert a[i] in sorted order in a[0...i]
        for (int j = i; j > 0 && less(a[j], a[j-1]); j--) {
            exch(a, j, j-1);
        }
        // a[0...i] is now sorted
        assert isSorted(a, 0, i);
    }
    assert isSorted(a);
}
```

Mergesort

- Mergesort es un algoritmo divide and conquer
  - La idea dividir el arreglo en dos mitades, y ordenar recursivamente cada una de las mitades
  - Y luego, asumiendo las dos mitades ordenadas, mezclarlas para obtener un arreglo ordenado

Mergesort

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the mergesort algorithm.
 */
public void sort(Comparable[] a) {
    // We need an auxiliary array of size a.length
    // That is, we need a.length extra space
    Comparable[] aux = new Comparable[a.length];
    sort(a, aux, 0, a.length-1);
    assert isSorted(a);
}
```

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the mergesort algorithm.
 */
protected void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Mergesort

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the mergesort algorithm.
 */
public void sort(Comparable[] a) {
    // We need an auxiliary array of size a.length
    // That is, we need a.length extra space
    Comparable[] aux = new Comparable[a.length];
    sort(a, aux, 0, a.length-1);
    assert isSorted(a);
}
```

ordenar la primera mitad

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the mergesort algorithm.
 */
protected void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}
```

Mergesort

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the mergesort algorithm.
 */
public void sort(Comparable[] a) {
    // We need an auxiliary array of size a.length
    // That is, we need a.length extra space
    Comparable[] aux = new Comparable[a.length];
    sort(a, aux, 0, a.length-1);
    assert isSorted(a);
}
```

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the mergesort algorithm.
 */
protected void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}
```

ordenar la primera mitad

ordenar la segunda mitad

Mergesort

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the mergesort algorithm.
 */
public void sort(Comparable[] a) {
    // We need an auxiliary array of size a.length
    // That is, we need a.length extra space
    Comparable[] aux = new Comparable[a.length];
    sort(a, aux, 0, a.length-1);
    assert isSorted(a);
}
```

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the mergesort algorithm.
 */
protected void sort(Comparable[] a, Comparable[] aux, int lo, int hi) {
    if (hi <= lo) return;
    int mid = lo + (hi - lo) / 2;
    sort(a, aux, lo, mid);
    sort(a, aux, mid + 1, hi);
    merge(a, aux, lo, mid, hi);
}
```

ordenar la primera mitad

ordenar la segunda mitad

mezclar las mitades ordenadas

Merge


```
/**
 * @pre isSorted(a, lo, mid) && isSorted(a, mid+1, hi)
 * @post Stably merge a[lo...mid] with a[mid+1...hi]
 *   using aux[lo...hi]
 */
protected void merge(Comparable[] a, Comparable[] aux,
                     int lo, int mid, int hi) {
    // check precondition
    assert isSorted(a, lo, mid);
    assert isSorted(a, mid+1, hi);
    // copy to aux[]
    for (int k = lo; k <= hi; k++) {
        aux[k] = a[k];
    }
    // merge back to a[]
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++) {
        // left half exhausted; take from the right
        if      (i > mid)                a[k] = aux[j++];
        // right half exhausted; take from the left
        else if (j > hi)                 a[k] = aux[i++];
        // key on right < key on left; take from the right
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        // key on right >= key on left; take from the left
        else                             a[k] = aux[i++];
    }
    // check postcondition: a[lo...hi] is sorted
    assert isSorted(a, lo, hi);
}
```

Tiempo de Ejecución

Analicemos su tiempo de ejecución en el peor caso:

- Merge se puede implementar en $\Theta(n)$
- Su ecuación de recurrencia viene dada por:

$$T(0) = c_1$$

$$T(1) = c_2$$

$$T(n) = 2 * T(\frac{n}{2}) + n$$

llamadas recursivas

Merge

Tiempo del MergeSort

Hagamos sustituciones:

$$2 * T(\frac{n}{2}) + n$$
$$= 2 * [2 * T(\frac{n}{4}) + \frac{n}{2}] + n$$
$$= 2 * [2 * [2 * T(\frac{n}{8}) + \frac{n}{4}] + \frac{n}{2}] + n$$
$$= \dots$$

En i sustituciones nos da: $2^i T(\frac{n}{2^i}) + i * n$

Obtenemos que: $T(\frac{n}{2^i}) = 1$ cuando: $\frac{n}{2^i} = 1$ Es decir: $i = log_2 n$

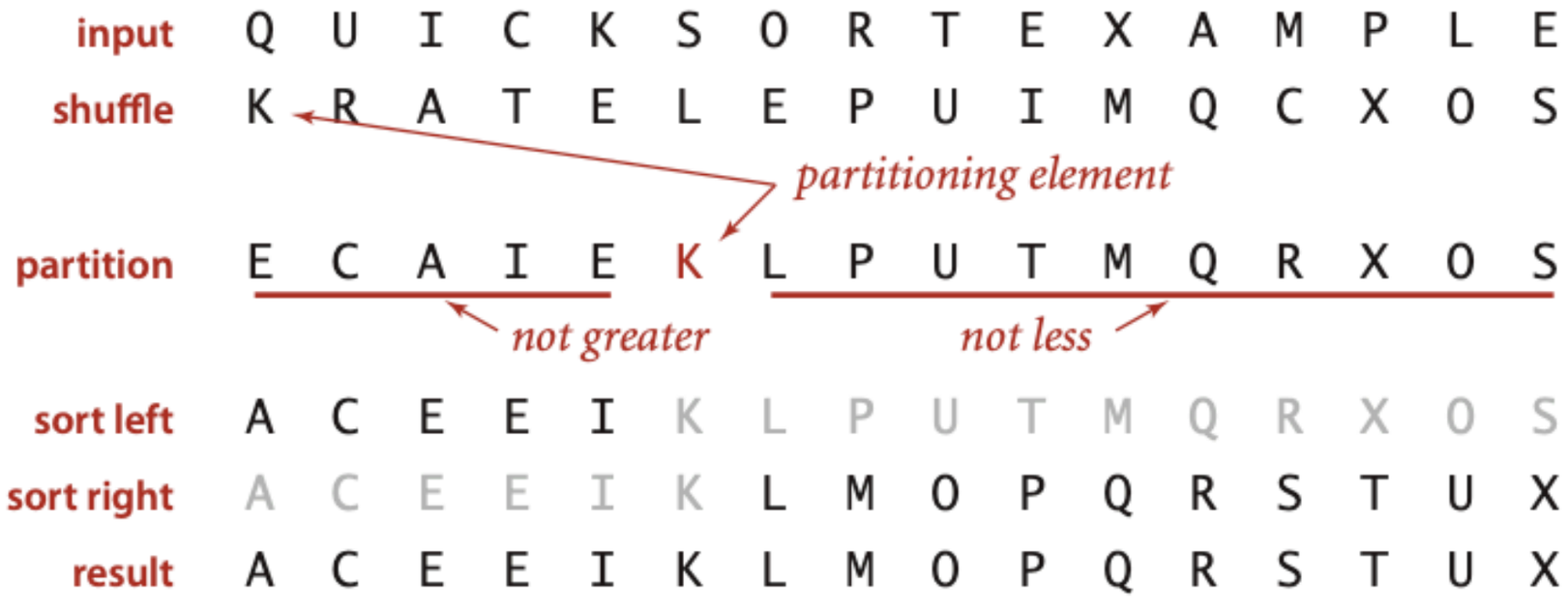
Reemplazando: $2^{log_2 n} + c_2 + n * log_2 n = n + n * log_2 n \in \Theta(n * log_2 n)$

Mergesort

- Es un algoritmo estable (bien implementado)
- La cantidad de comparaciones es: $O(n * log\ n)$
- La cantidad de intercambios es: $O(n * log\ n)$
- La implementación que vimos no ordena in-place, es decir, no ordena sobre el mismo arreglo
- Requiere $O(n)$ espacio adicional

Quicksort

- Elegir un pivot (elemento del arreglo),
- Particionar el arreglo en dos partes:
  - Una parte izquierda conteniendo todos elementos menores o iguales al pivot
  - Una parte derecha conteniendo todos elementos mayores o iguales al pivot
  - Poner al pivot en la posición que le corresponde
- Ordenar recursivamente las partes izquierda y derecha



Quicksort

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the quick sort algorithm.
 */
public void sort(Comparable[] a) {
    // Shuffle the array to eliminate dependence on input.
    StdRandom.shuffle(a);
    sort(a, 0, a.length - 1);
    assert isSorted(a);
}

/**
 * @post quicksort the subarray from a[lo] to a[hi]
 */
protected void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    // Sort left part a[lo .. j-1]
    sort(a, lo, j-1);
    // Sort right part a[j+1 .. hi].
    sort(a, j+1, hi);
    assert isSorted(a, lo, hi);
}
```

Quicksort

reordenamos el arreglo aleatoriamente por razones de eficiencia

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the quick sort algorithm.
 */
public void sort(Comparable[] a) {
    // Shuffle the array to eliminate dependence on input.
    StdRandom.shuffle(a);
    sort(a, 0, a.length - 1);
    assert isSorted(a);
}

/**
 * @post quicksort the subarray from a[lo] to a[hi]
 */
protected void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    // Sort left part a[lo .. j-1]
    sort(a, lo, j-1);
    // Sort right part a[j+1 .. hi].
    sort(a, j+1, hi);
    assert isSorted(a, lo, hi);
}
```

Quicksort

reordenamos el arreglo aleatoriamente por razones de eficiencia

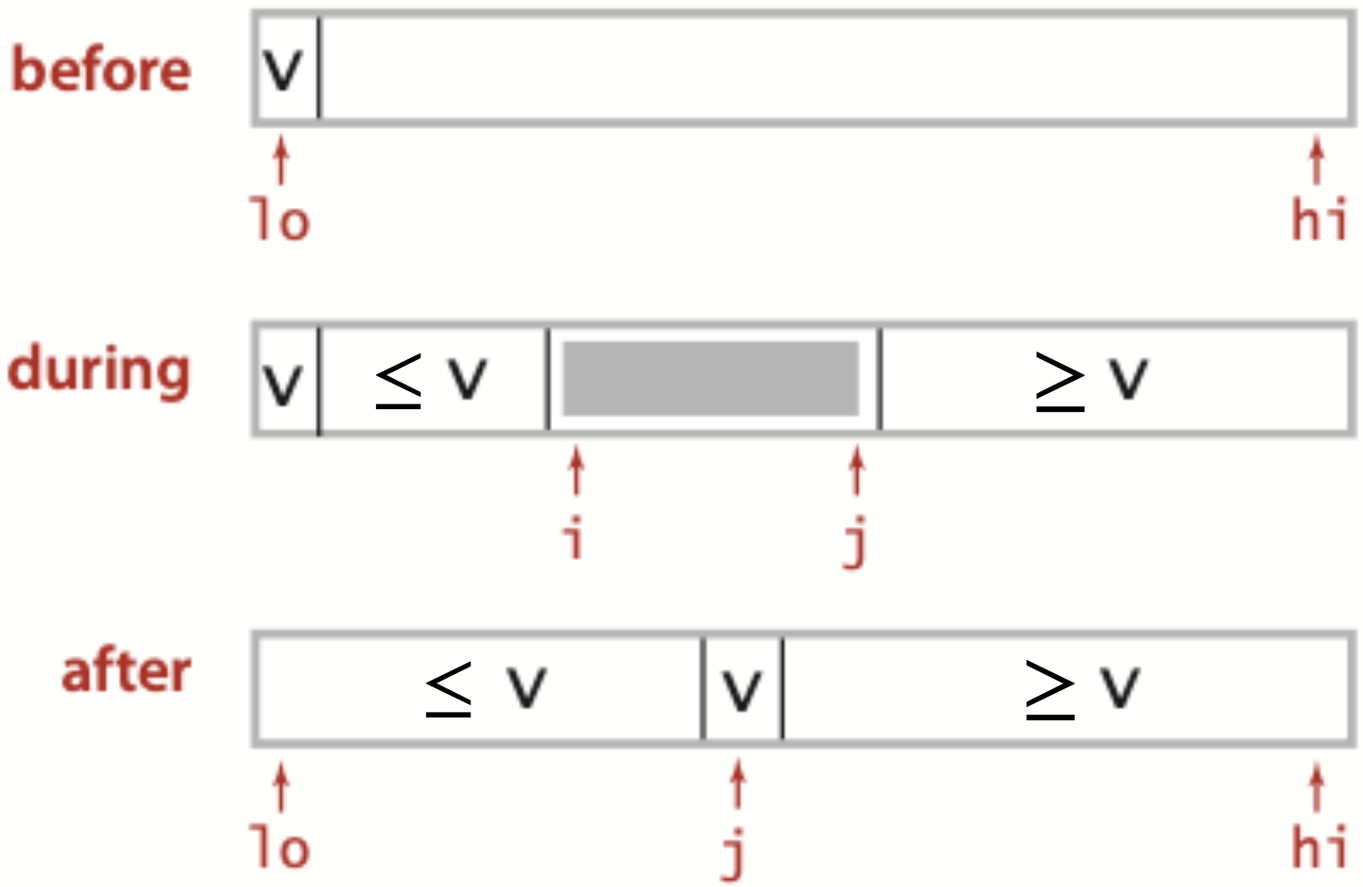
otras implementaciones eligen como pivot un elemento aleatorio del arreglo

```java
/**
 * @post Rearranges the array a in ascending order
 *   using the quick sort algorithm.
 */
public void sort(Comparable[] a) {
    // Shuffle the array to eliminate dependence on input.
    StdRandom.shuffle(a);
    sort(a, 0, a.length - 1);
    assert isSorted(a);
}

/**
 * @post quicksort the subarray from a[lo] to a[hi]
 */
protected void sort(Comparable[] a, int lo, int hi) {
    if (hi <= lo) return;
    int j = partition(a, lo, hi);
    // Sort left part a[lo .. j-1]
    sort(a, lo, j-1);
    // Sort right part a[j+1 .. hi].
    sort(a, j+1, hi);
    assert isSorted(a, lo, hi);
}
```

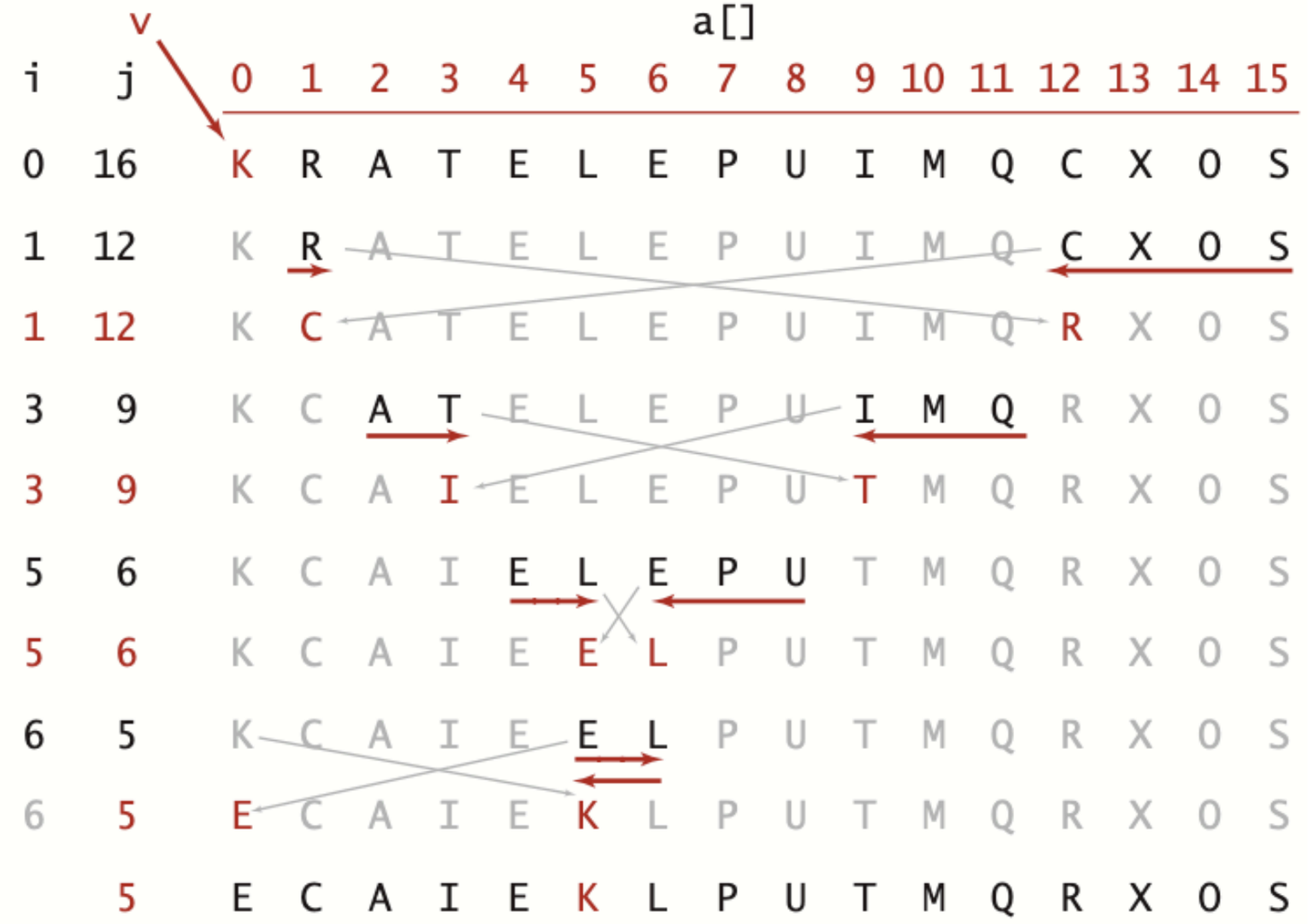
Partition

- Elegir el primer elemento como pivot
- Avanzar un índice i desde el inicio mientras los elementos sean menores al pivot
- Avanzar un índice j desde el final mientras los elementos sean mayores al pivot
- Intercambiar los elementos en i y j cuando ya no se pueda avanzar por las condiciones anteriores
- Al terminar, ubicar el pivot en su posición correspondiente: a[j]



```java
/**
 * @post Partition the subarray a[lo..hi] so that
 *   a[lo..j-1] <= a[j] <= a[j+1..hi] and return the index j.
 */
protected int partition(Comparable[] a, int lo, int hi) {
    int i = lo;
    int j = hi + 1;
    Comparable v = a[lo];
    while (true) {
        // Scan right
        while (less(a[++i], v)) if (i == hi) break;
        // Scan left
        while (less(v, a[--j])) if (j == lo) break;
        // Check if scan is complete
        if (i >= j) break;
        // Exchange
        exch(a, i, j);
    }
    // Put v into position (a[j])
    exch(a, lo, j);
    // Now, a[lo .. j-1] <= a[j]<= a[j+1 .. hi]
    return j;
}
```

```java
/**
 * @post Partition the subarray a[lo..hi] so that
 *   a[lo..j-1] <= a[j] <= a[j+1..hi] and return the index j.
 */
protected int partition(Comparable[] a, int lo, int hi) {
    int i = lo;
    int j = hi + 1;
    Comparable v = a[lo];
    while (true) {
        // Scan right
        while (less(a[++i], v)) if (i == hi) break;
        // Scan left
        while (less(v, a[--j])) if (j == lo) break;
        // Check if scan is complete
        if (i >= j) break;
        // Exchange
        exch(a, i, j);
    }
    // Put v into position
    exch(a, lo, j);
    // Now, a[lo .. j-1] <=
    return j;
}
```



Tiempo de Ejecución

En el peor caso de elección del pivot, la cantidad de elementos se decrementa en uno, es decir:

$$T(0) = c_1$$
$$T(1) = c_2$$
$$T(n) = c + T(n-1) + n$$

Siempre se elige el elemento más chico o más grande como pívot.

tiempo del partition

Llamada recursiva

Es decir, tenemos: $T(n) \in O(n^2)$

Por ejemplo: [4,1,2,3]

Ej:Correr partition!

Tiempo de Ejecución

En un buen caso, partition dividirá el arreglo en aproximadamente dos mitades, y obtenemos la misma ecuación de recurrencia que en el mergesort

$$T(0) = c_1$$

$$T(1) = c_2$$

$$T(n) = 2 * T(\frac{n}{2}) + n$$

tiempo del partition

Es decir, tenemos: $T(n) \in O(n * log_2 n)$

En el caso promedio, para inputs ordenados aleatoriamente, partition va a generar algunas divisiones buenas y otras malas

Se puede demostrar que en el caso promedio es $O(n * log_2 n)$

Observaciones

- En la practica el Quicksort se comporta mejor que otros algoritmos de sorting
  - En el caso promedio es $O(n * log_2 n)$
    - El peor caso tiene muy pocas probabilidades de ocurrir
  - Y la constante de proporcionalidad es más pequeña que para Mergesort

- El partition no necesita espacio extra

- No es un algoritmo estable (depende de la implementación del partition)

Cota inferior para Algoritmos de Sorting

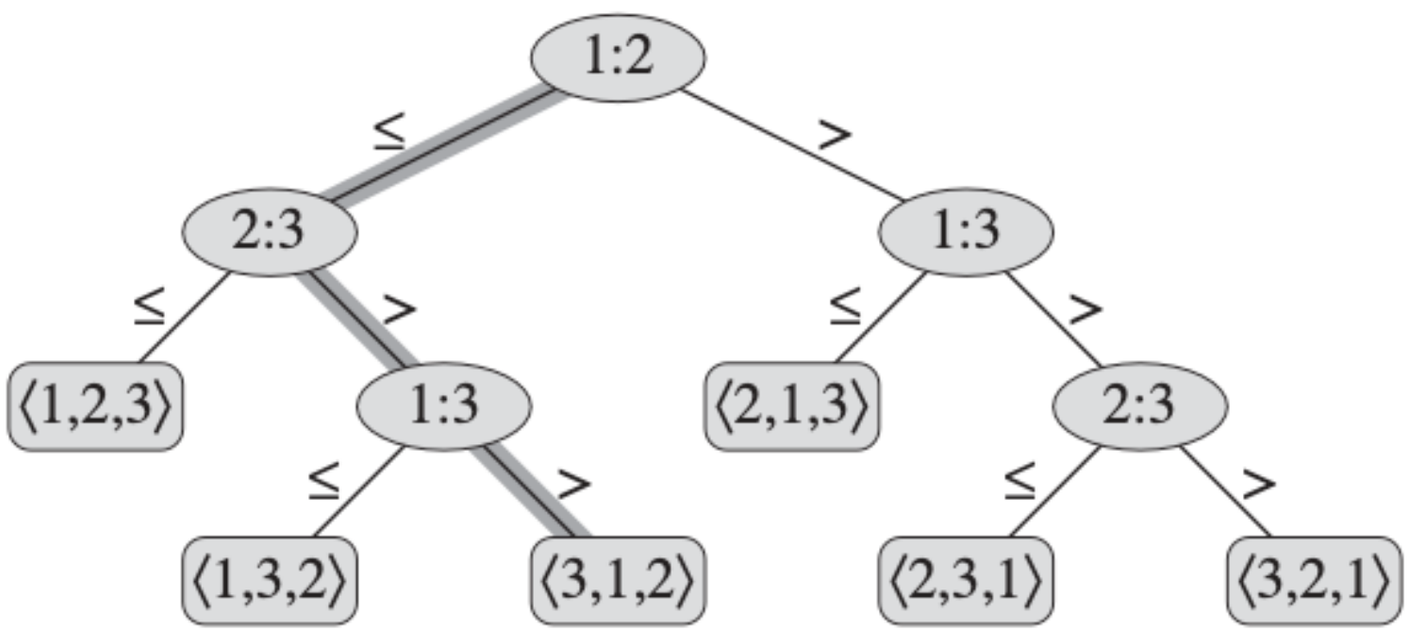
Tenemos el siguiente resultado para algoritmos de sorting:

Teorema: Cualquier algoritmo de sorting que utilice comparaciones para ordenar es $\Omega(n * log\ n)$

Corolario: Mergesort es un algoritmo basado en comparaciones asintóticamente óptimo (es $\Theta(n * log_2 n)$)

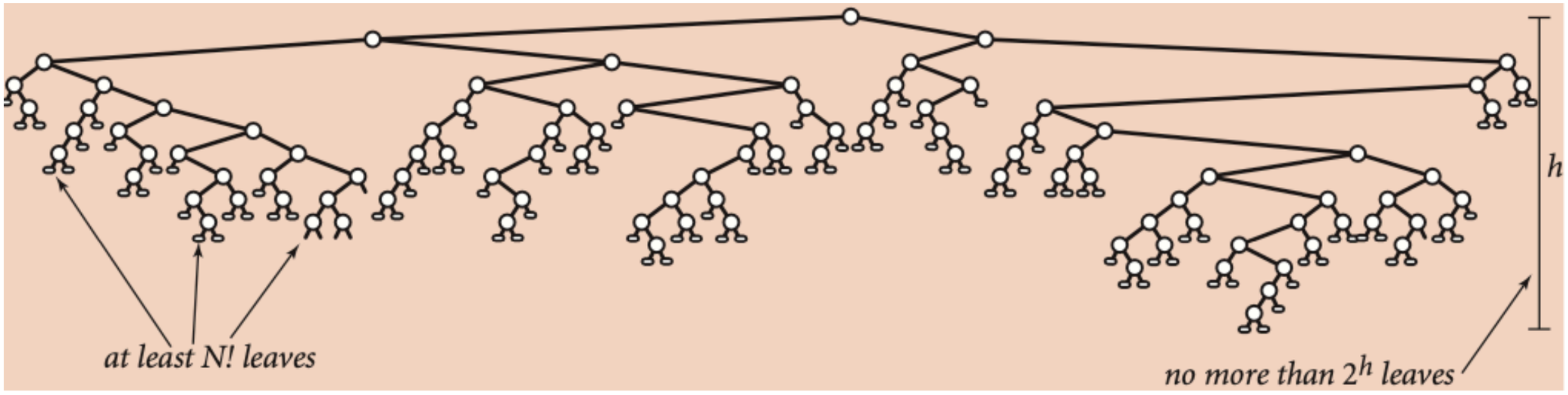
Cota inferior para sorting: Prueba

- Podemos modelar el funcionamiento de los algoritmos de sorting que realizan comparaciones con un árbol de decisión
- Nos abstraemos del valor particular de cada elemento y consideramos sólo sus posiciones
- Los nodos internos modelan las comparaciones realizadas por el algoritmo
  - Un nodo anotado con i:j modela que el algoritmo realiza una comparación entre los elementos a[i] y a[j]
- Las hojas denotan que el algoritmo llegó a una permutación ordenada del arreglo original
  - Ej: <3, 1, 2> denota que a[3] <= a[1] <= a[2]
- Un camino desde la raíz a una hoja representa una ejecución del algoritmo
- Cualquier algoritmo correcto debe producir las $n!$ permutaciones posibles de su entrada
  - O fallaría para las permutaciones que no puede generar



Cota inferior para sorting: Prueba

- Sea h la altura del árbol con l hojas que modela el comportamiento del algoritmo de comparación para ordenar n elementos
  - La máxima cantidad de comparaciones que hace el algoritmo es h



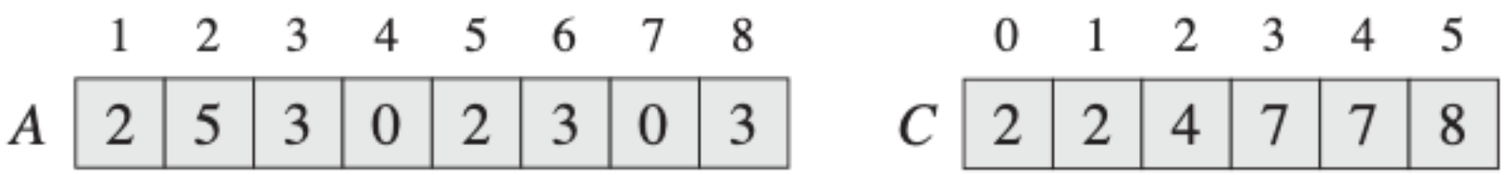
- Para que el algoritmo sea correcto debe valer: $n! \leq l$
- Propiedad: Un árbol de altura h tiene a lo sumo 2^h hojas: $l \leq 2^h$
- Luego: $n! \leq 2^h$ y
  - $log_2 2^h \geq log_2(n!)$ (tomando logaritmos a ambos miembros)
  - $h \geq log_2(n!)$ (propiedad: $log_2(n!) \in \Theta(n * log_2 n)$)
  - $h \geq n * log_2 n$

Concluimos que: $h \in \Omega(n * log_2 n)$

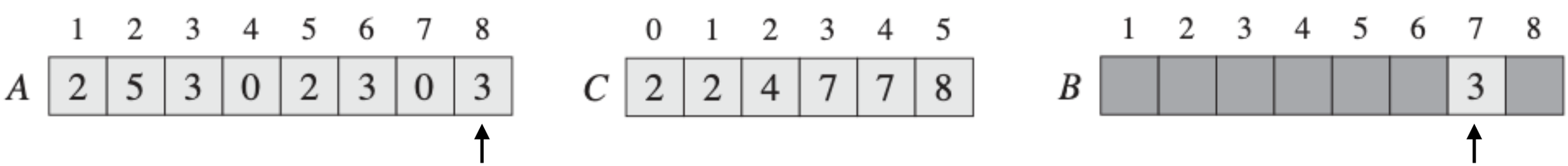
Counting sort

- Counting sort asume que los n elementos de entrada están en un rango pequeño $[0...k]$
  - Counting sort no hace comparaciones entre los elementos
  - Si $k \in O(n)$ el algoritmo es $\Theta(n)$
- Idea:
  - Contar la cantidad de repeticiones de los elementos en un arreglo auxiliar C

  - Contar la cantidad de elementos menores o iguales en el arreglo C



  - Iterar comenzando desde el final: el elemento A[i] va en la posición C[i] del resultado B, se resta 1 a C[i], y se decrementa i en 1



```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

	0	1	2	3	4	5
C	2	0	2	3	0	1

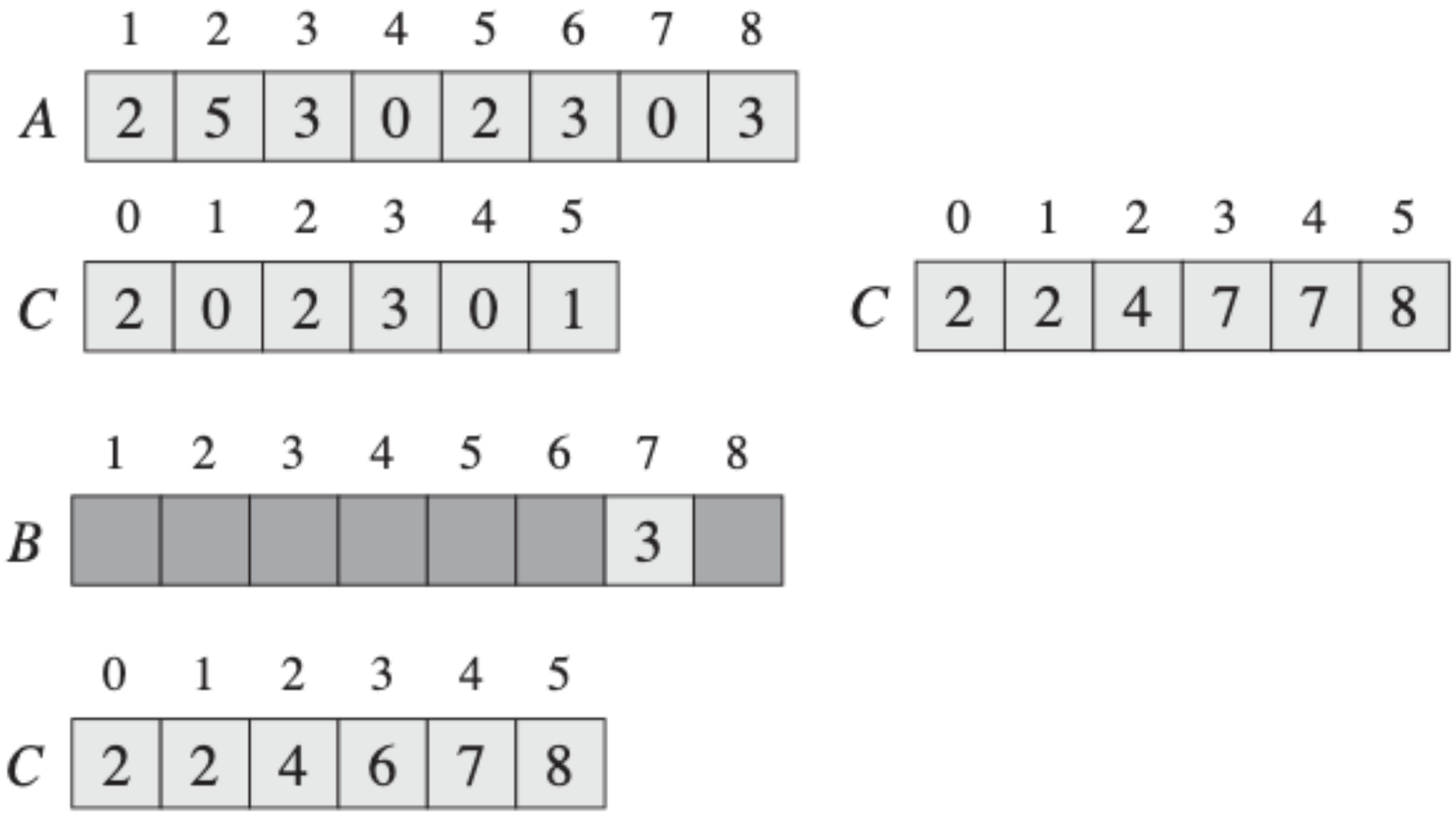
```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

	1	2	3	4	5	6	7	8
A	2	5	3	0	2	3	0	3

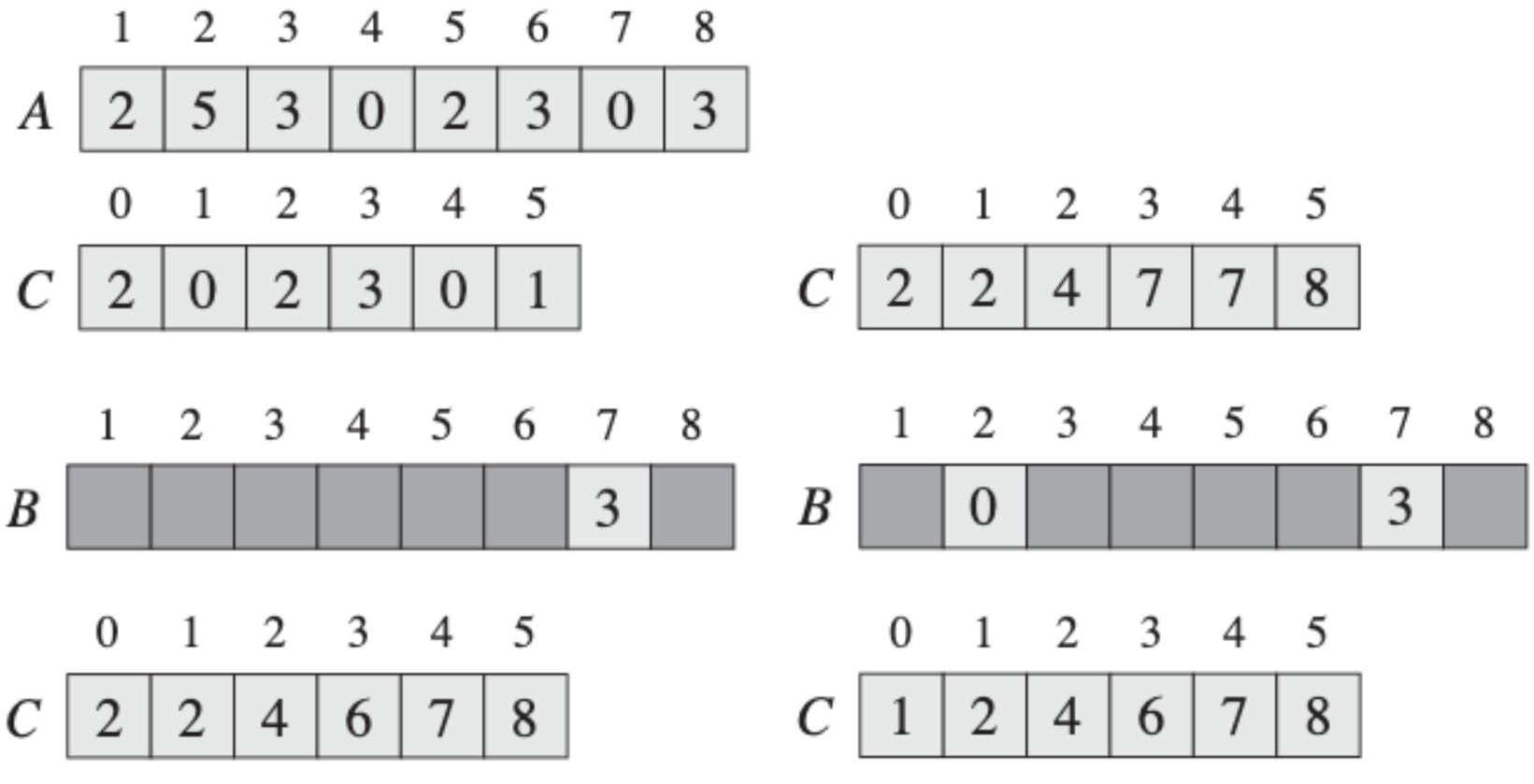
	0	1	2	3	4	5
C	2	0	2	3	0	1

	0	1	2	3	4	5
C	2	2	4	7	7	8

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```

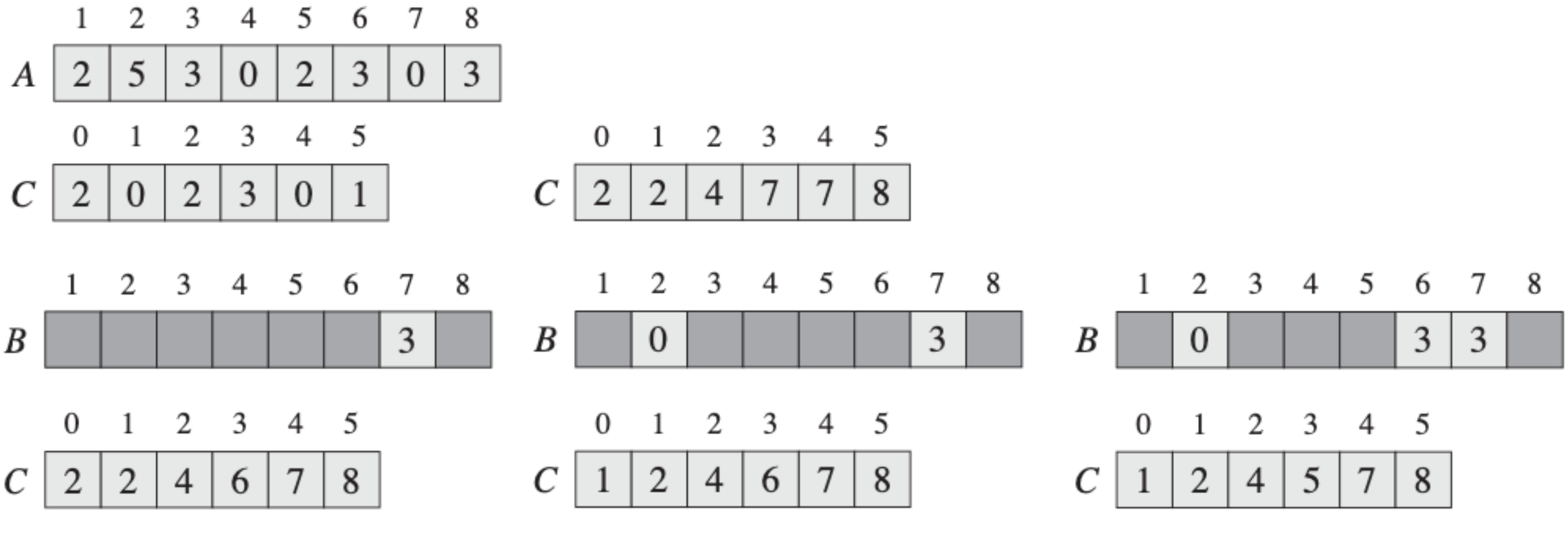


```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```



COUNTING-SORT(A, B, k)

```
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i − 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] − 1
```



Tiempo de ejecución

Contar menores o iguales

Inicialización

$$T(n,k) = n + n + k + n \in \Theta(k+n)$$

Contar repeticiones

Poner los elementos en su lugar

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k
 8      C[i] = C[i] + C[i - 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] - 1
```

se ejecuta n veces

se ejecuta n veces

se ejecuta k veces

se ejecuta n veces

Tiempo de ejecución

Contar menores o iguales

Inicialización

$$T(n,k) = n + n + k + n \in \Theta(k+n)$$

Contar repeticiones

Poner los elementos en su lugar

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length                          | se ejecuta n veces
 5      C[A[j]] = C[A[j]] + 1                      |
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k                                 | se ejecuta k veces
 8      C[i] = C[i] + C[i - 1]                     |
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1                      | se ejecuta n veces
11      B[C[A[j]]] = A[j]                          |
12      C[A[j]] = C[A[j]] - 1                      |
```

Si $k \in O(n)$ es $\Theta(n)$

Tiempo de ejecución

Contar menores o iguales

Inicialización

$$T(n,k) = n + n + k + n \in \Theta(k+n)$$

Contar repeticiones

Poner los elementos en su lugar

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k
 3      C[i] = 0
 4  for j = 1 to A.length                          | se ejecuta n veces
 5      C[A[j]] = C[A[j]] + 1                      |
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k                                 | se ejecuta k veces
 8      C[i] = C[i] + C[i - 1]                     |
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1                      | se ejecuta n veces
11      B[C[A[j]]] = A[j]                          |
12      C[A[j]] = C[A[j]] - 1                      |
```

Si $k \in O(n)$ es $\Theta(n)$

Si k es muy grande, el algoritmo es ineficiente!

Tiempo de ejecución

Contar menores o iguales

Inicialización

$$T(n,k) = n + n + k + n \in \Theta(k+n)$$

Contar repeticiones

Poner los elementos en su lugar

```
COUNTING-SORT(A, B, k)
 1  let C[0..k] be a new array
 2  for i = 0 to k                          | se ejecuta n veces
 3      C[i] = 0
 4  for j = 1 to A.length                   | se ejecuta n veces
 5      C[A[j]] = C[A[j]] + 1
 6  // C[i] now contains the number of elements equal to i.
 7  for i = 1 to k                          | se ejecuta k veces
 8      C[i] = C[i] + C[i - 1]
 9  // C[i] now contains the number of elements less than or equal to i.
10  for j = A.length downto 1               | se ejecuta n veces
11      B[C[A[j]]] = A[j]
12      C[A[j]] = C[A[j]] - 1
```

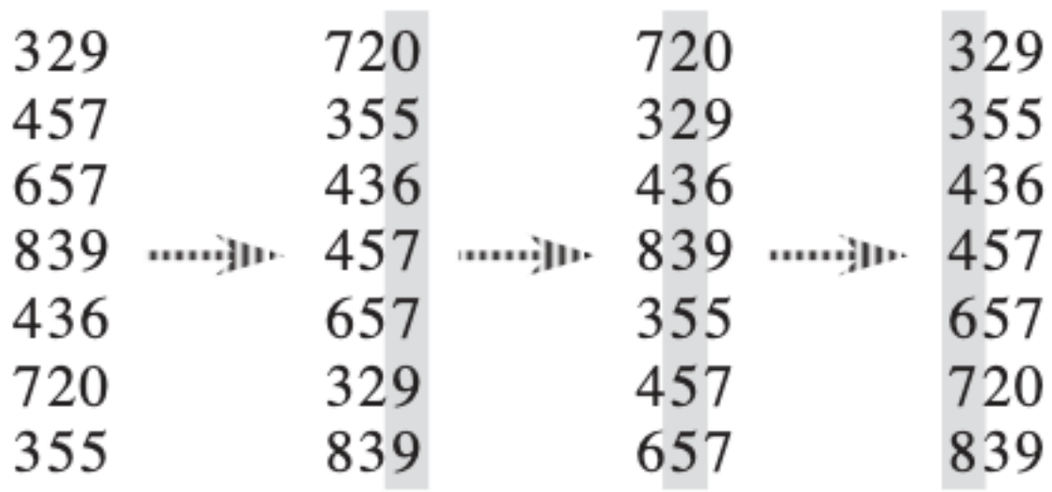
Si $k \in O(n)$ es $\Theta(n)$

Si k es muy grande, el algoritmo es ineficiente!

Esta implementación de counting sort es estable

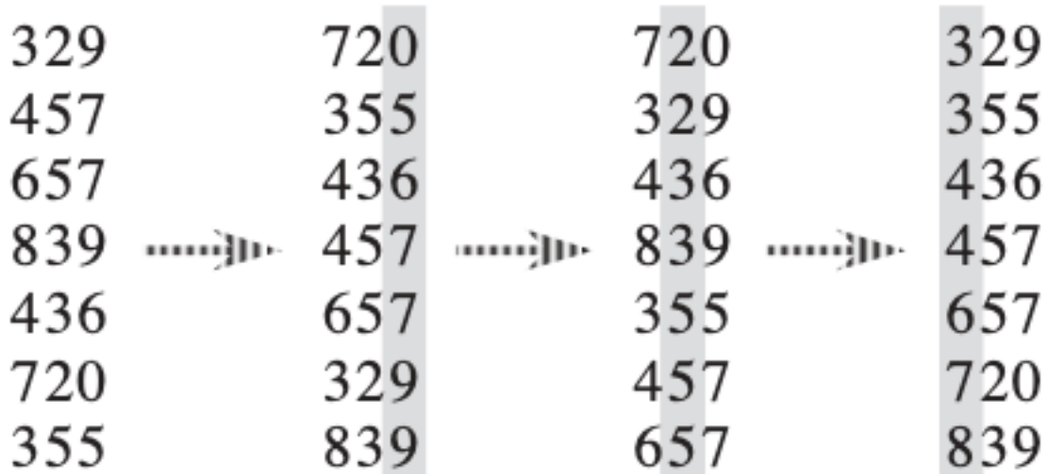
Radix sort

- Ordenar los números por el primer dígito menos significativo
- Ordenar los números por el segundo dígito menos significativo
- Ordenar los números por el tercer dígito menos significativo
- Y así sucesivamente...



- Para que funcione correctamente requiere usar un algoritmo de sorting estable
- También se puede usar para ordenar cartas, primero ordenando por número y después por palo

Radix sort



RADIX-SORT(A, d)

1 **for** $i = 1$ **to** d

2 use a stable sort to sort array A on digit i

- Podemos usar cualquier algoritmo de ordenamiento estable para implementar Radix sort
  - por ejemplo, Counting sort
- Si el algoritmo usado es $\Theta(n + k)$ (ej. counting sort) radix sort es $\Theta(d * (n + k))$

Actividades

- Leer los capítulos 2, 3, 7, 8 del libro "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009

Bibliografía

- "Algorithms (4th edition)". R. Sedgewick, K. Wayne. Addison-Wesley. 2016
- "Introduction to Algorithms, 3rd Edition". T. Cormen, C. Leiserson, R. Rivest, C. Stein. MIT Press. 2009
- "Data Structures and Algorithms". A. Aho, J. Hopcroft, J. Ullman. Addison-Wesley. 1983