

Algoritmos de Ordenamiento y Búsqueda

Módulo 2

Capítulo 7

Arreglos Parciales

El tamaño de un arreglo se especifica desde la definición.

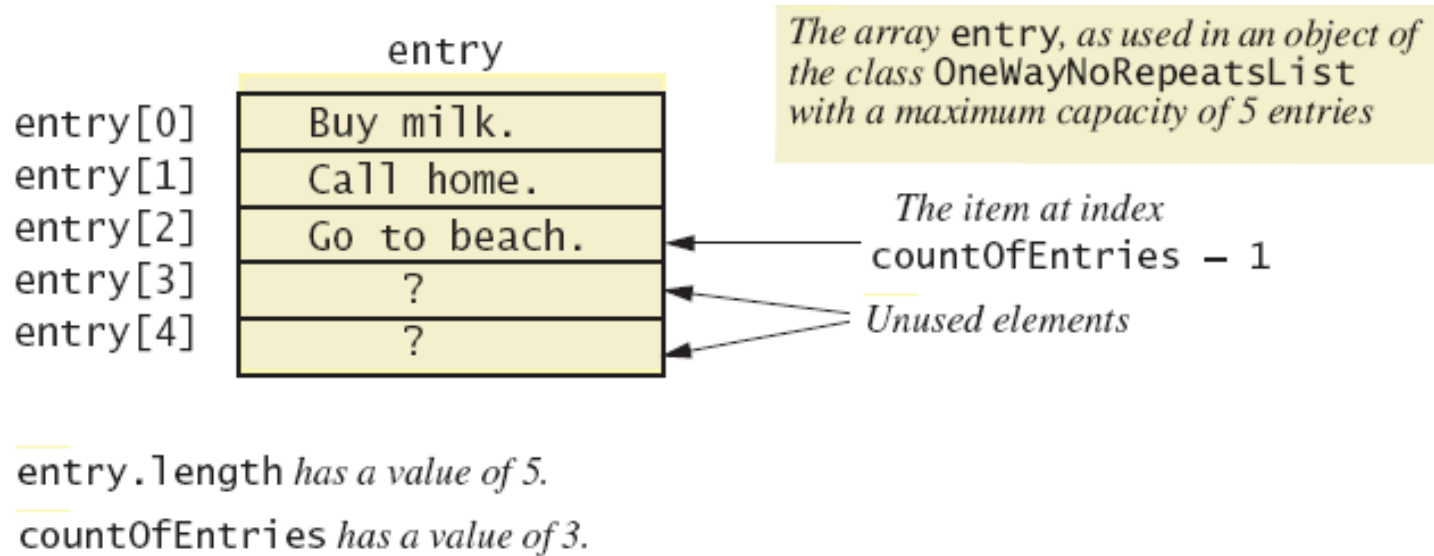
No todos los elementos del arreglo almacenan algún contenido.

- Los arreglos que no tienen todos los elementos ocupados son llamados “partially filled arrays”, o “arreglos parciales”.

Es tarea del programador llevar un registro de cuáles están llenos y cuáles están ocupados.

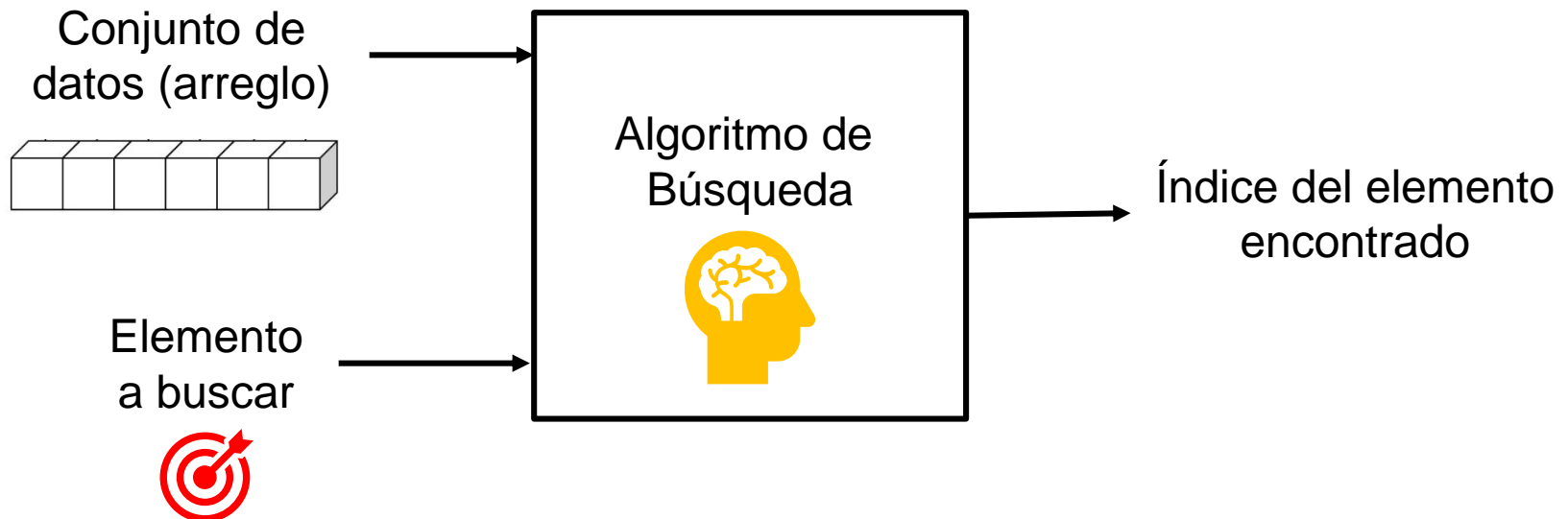
Arreglos Parciales

Figure 7.4 A partially filled array



Algoritmos de Búsqueda

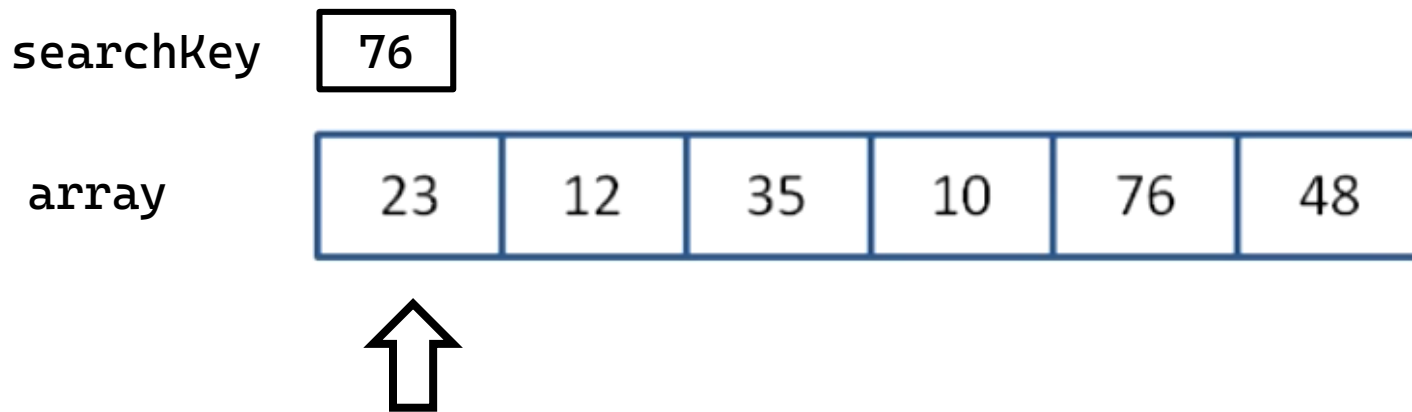
Un algoritmo de búsqueda nos sirve para encontrar algún elemento dentro de un conjunto de elementos.



Búsqueda secuencial

Búsqueda lineal

La búsqueda secuencial consiste revisar cada índice del arreglo hasta que encontremos el elemento deseado.

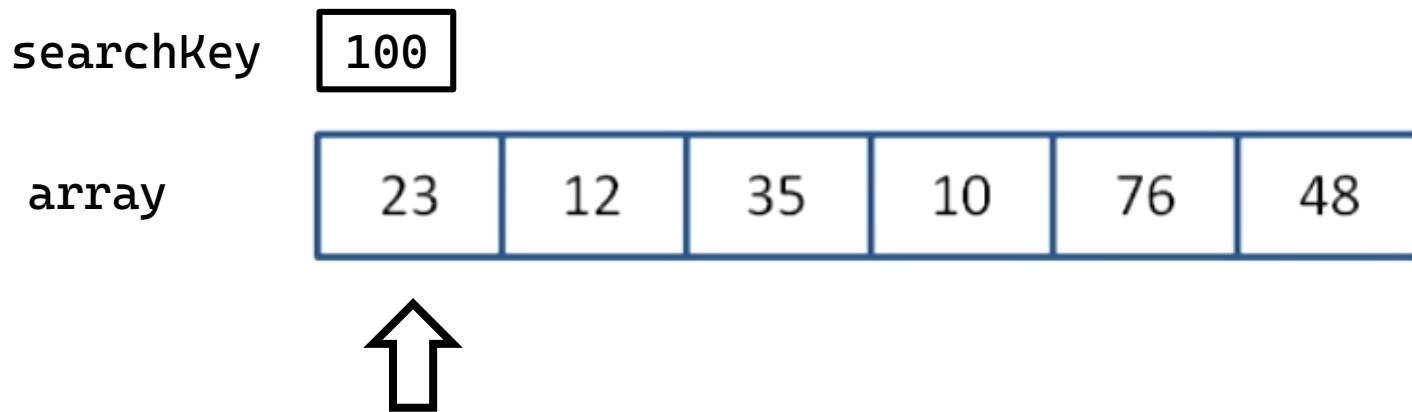


Found!
Return index 4!

Búsqueda secuencial

Búsqueda lineal

La búsqueda secuencial consiste revisar cada índice del arreglo hasta que encontremos el elemento deseado.



Not found!
Return -1!


Búsqueda secuencial

La búsqueda secuencial es puede llegar a ser muy tardada, pues el peor escenario es que el elemento no **existe en el arreglo**.

Sólo tiene sentido realizarla cuando el arreglo está desordenado.

Búsqueda secuencial

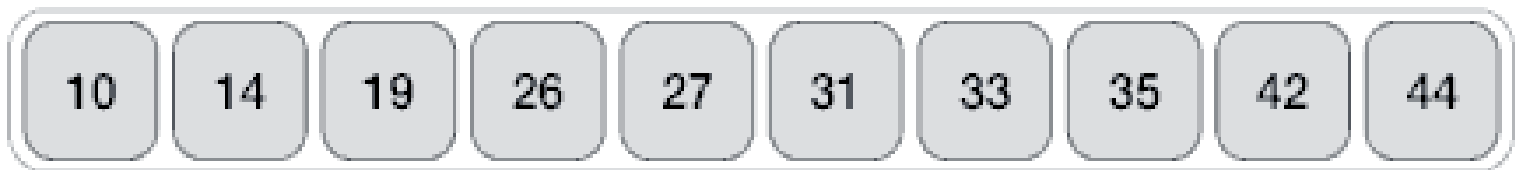
```
public static int findElement(int[] array, int target){  
    for(int i=0; i<array.length; i++){  
        if (array[i] == target)  
            return i;  
    }  
    return -1;  
}
```



Si no encuentra una
coincidencia, devuelve
-1, indicando error

Retorna el índice de la
primera coincidencia
encontrada

Linear Search



=
33

Algoritmos de Ordenamiento

Un algoritmo de ordenamiento se utiliza para **reacomodar un arreglo o lista** de acuerdo a una comparación interna de los elementos. Ejemplo:

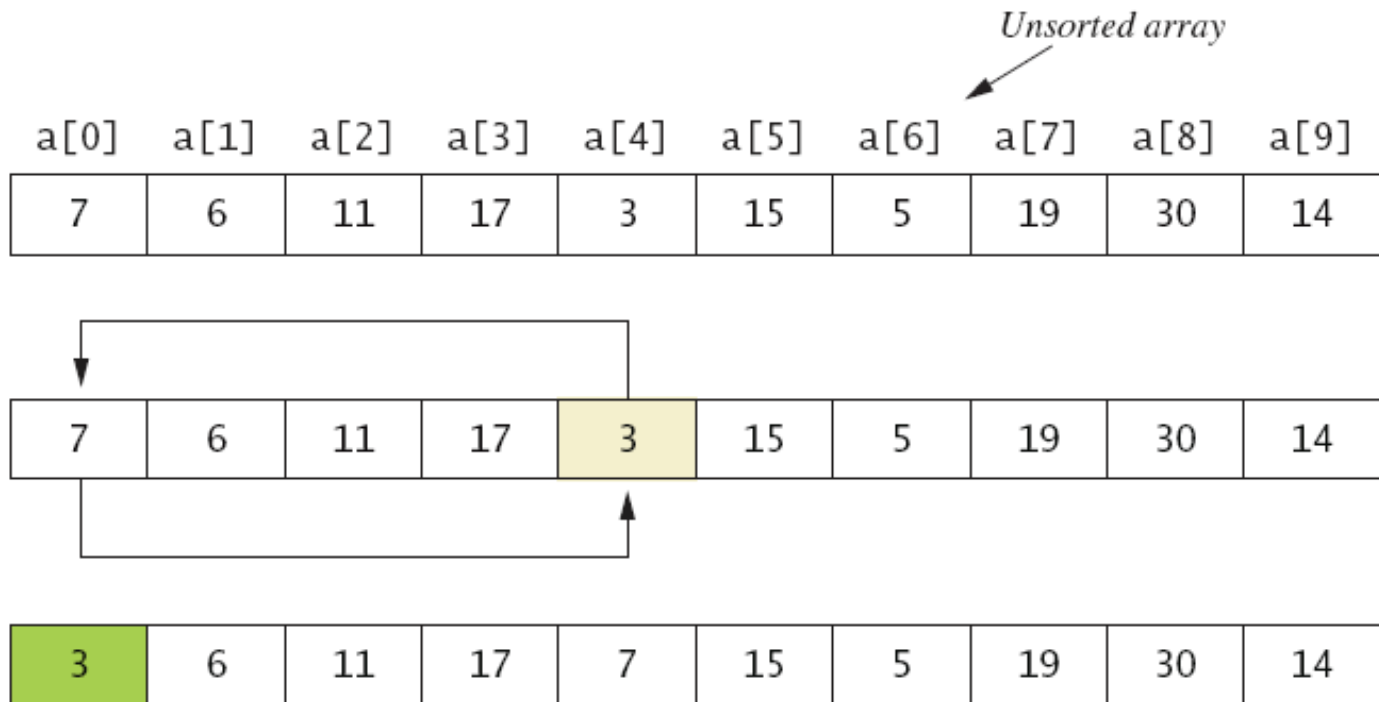
- Ordenar de mayor a menor
- Ordenar alfabéticamente
- Ordenar por fecha
- Ordenar por valor absoluto

Selection Sort

El Selection Sort es un algoritmo de ordenamiento que consiste en encontrar el elemento más pequeño de una lista y acomodarlo en la posición correcta.

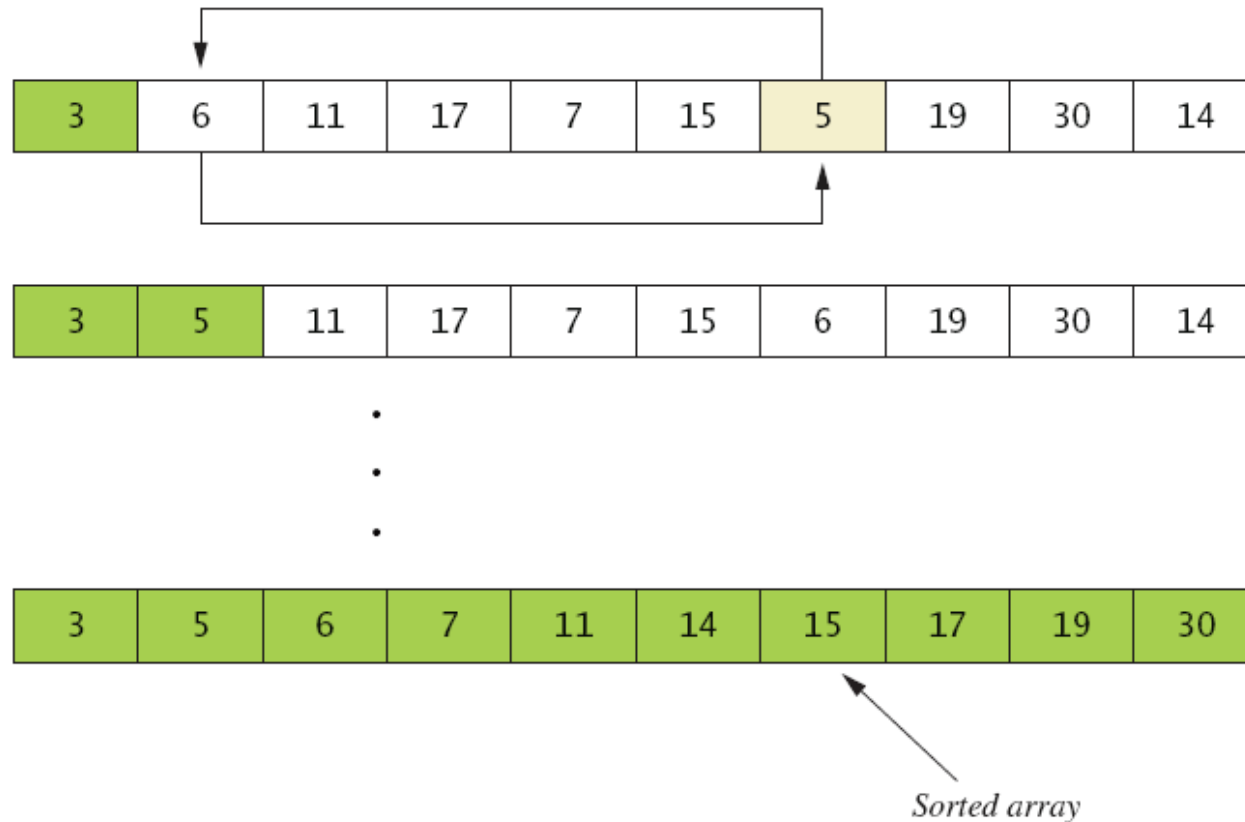
Selection Sort

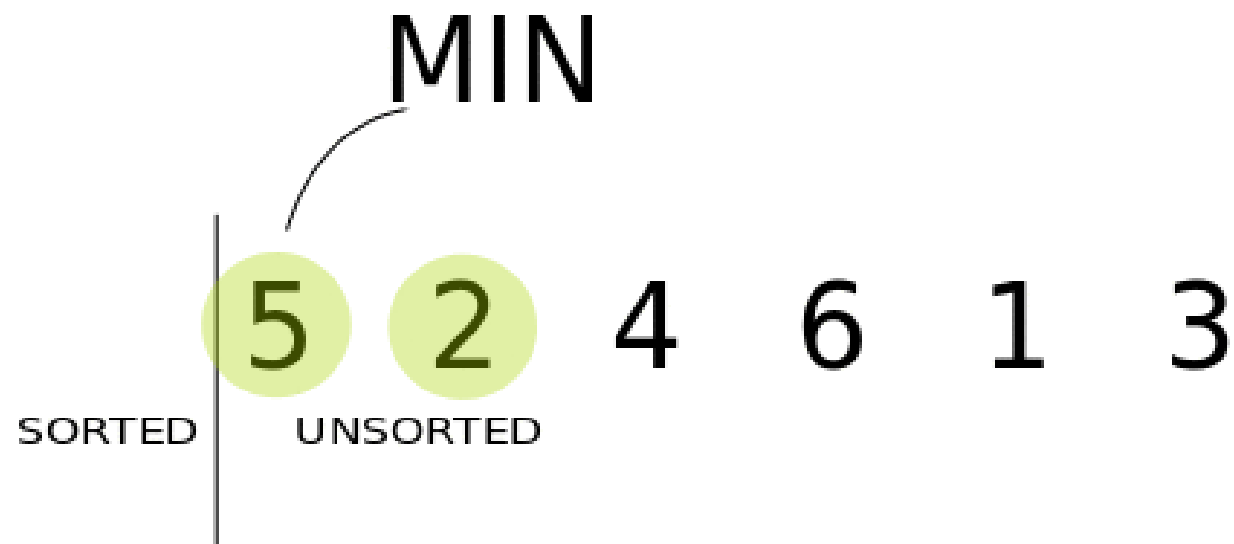
Figure 7.5a



Selection Sort

Figure 7.5b





```
public static void selectionSort(int[] array){

    //check valid input array
    if (array == null)
        return;

    //Iterate over every position, trying to find the smallest element and
    // place it on index i
    for(int i=0; i<array.length-1; i++){

        //Starting from i (since all elements before i are already sorted),
        // look for the smallest array element, and store its index in min.
        // Assuming initially the smallest element is i, we will store it on min.
        int min = i;

        //Well begin our internal loop on i+1, since min was already initiated with i
        for(int j=i+1; j<array.length; j++){
            //If we encounter a smaller element than array[min], we store its index on min
            if (array[j] < array[min]){
                min = j;
            }
        }

        //Swap the contents of array[i] with array[min]
        int temp = array[min];
        array[min] = array[i];
        array[i] = temp;
    }
}
```

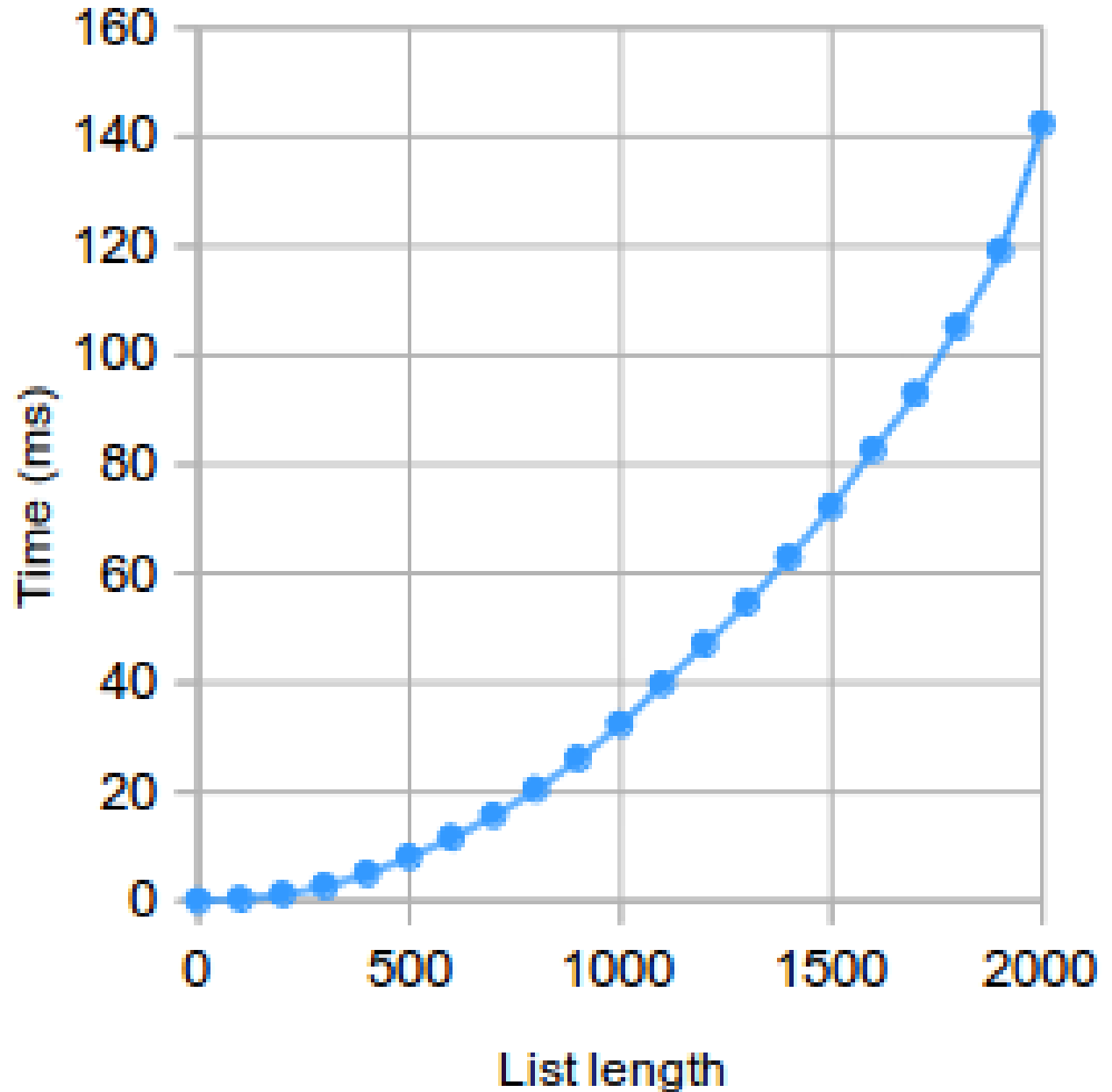
Selection Sort

Selection sort es el algoritmo más simple, **¡pero es muy ineficiente para arreglos muy grandes!**

- Para un arreglo de **10 elementos**, se tienen que realizar **45 comparaciones**.
- Para un arreglo de **20 elementos**, se tienen que realizar **210 comparaciones**.
- Para un arreglo de **100 elementos**, se realizarán **5050 comparaciones**

Conforme incrementa la cantidad de elementos, la cantidad de comparaciones crecerá **cuadráticamente**.

Selection Sort



Bubble Sort

El **bubbleSort** es un algoritmo de ordenamiento basado en comparar **elementos adyacentes**.

Cuando los dos elementos no están en el orden correcto, los intercambia.

El algoritmo termina cuando recorre la lista completa sin realizar intercambios.

Bubble Sort

Primera Corrida

(**5** **1** 4 2 8) \rightarrow (**1** **5** 4 2 8) Comparamos los primeros dos elementos, como $5 > 1$ intercambiamos.

(1 **5** **4** 2 8) \rightarrow (1 **4** **5** 2 8) Como $5 > 4$, intercambiamos

(1 4 **5** **2** 8) \rightarrow (1 4 **2** **5** 8) Como $5 > 2$, intercambiamos

(1 4 2 **5** **8**) \rightarrow (1 4 2 **5** **8**) Como $5 < 8$, no intercambiamos.

Bubble Sort

Segunda Corrida

(**1** **4** 2 5 8) \rightarrow (**1** **4** 2 5 8)

(1 **4** **2** 5 8) \rightarrow (1 **2** **4** 5 8) Como $4 > 2$, intercambiamos

(1 2 **4** **5** 8) \rightarrow (1 2 **4** **5** 8)

(1 2 4 **5** **8**) \rightarrow (1 2 4 **5** **8**)

Bubble Sort

Tercera Corrida

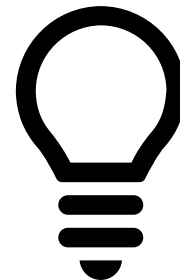
(**1** **2** 4 5 8) \rightarrow (**1** **2** 4 5 8)

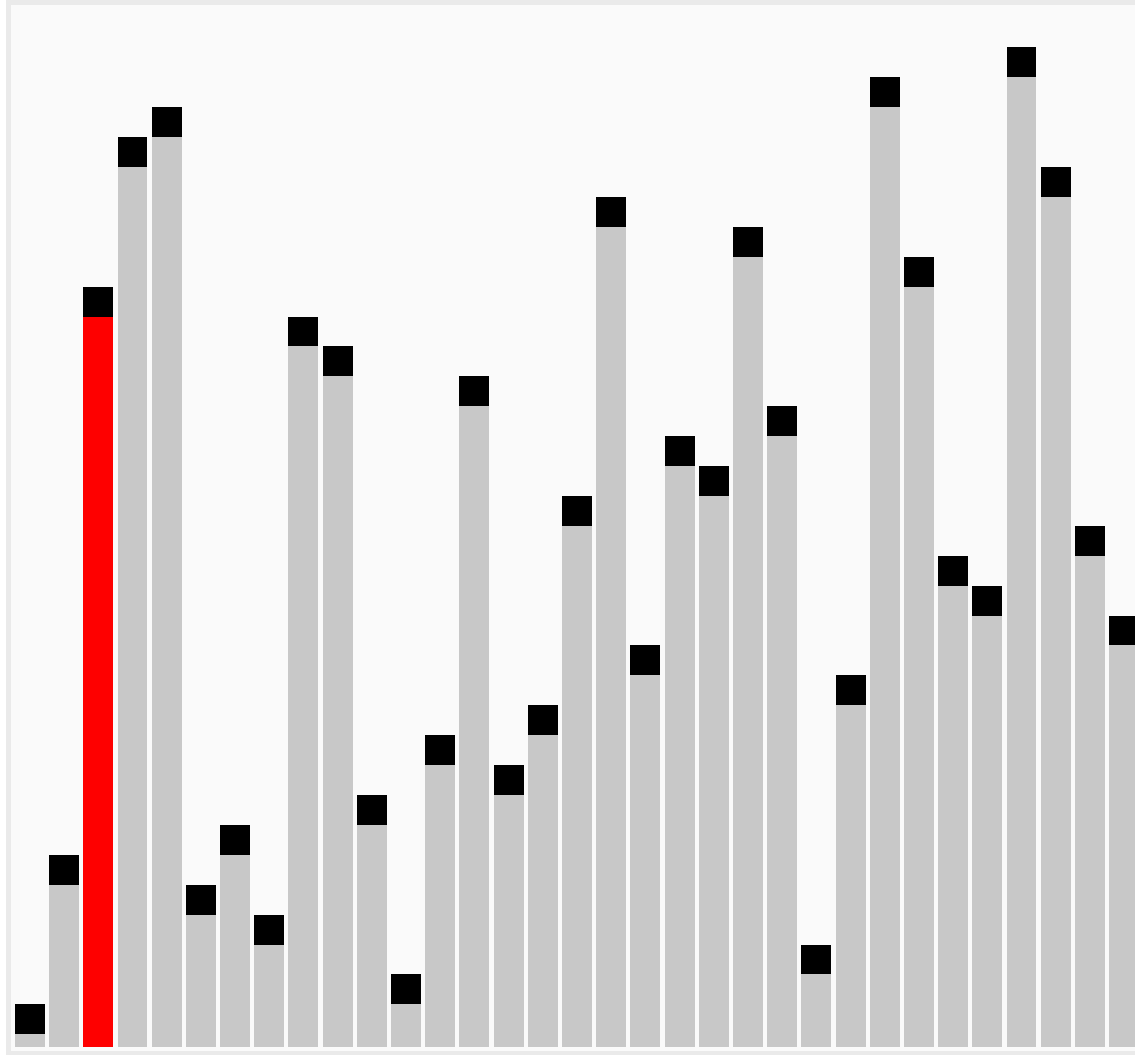
(1 **2** **4** 5 8) \rightarrow (1 **2** **4** 5 8)

(1 2 **4** **5** 8) \rightarrow (1 2 **4** **5** 8)

(1 2 4 **5** **8**) \rightarrow (1 2 4 **5** **8**)

Como completamos una
corrida completa sin
intercambiar valores,
sabemos que el arreglo
está ordenado!





```
//Bubble Sort
public static void bubbleSort(int[] array){

    //check valid input array
    if (array == null)
        return;

    //This flag will turn "true" every time a swap has been performed
    // Its initial value is true
    boolean flagSwap = true;

    //This loop will control every pass we do through the array.
    // If no swaps are performed on a pass, then flagSwap will be false
    // and finish the loop
    for(int i = 0; i < array.length - 1 && flagSwap; i++){
        flagSwap = false;
        //Loop through the array up to array.length-1-i.
        // Everything after array.length-i is already sorted.
        // We subtract 1 to avoid overflowing array[j+1]
        for(int j = 0; j < array.length - 1 - i; j++){
            //Swap the contents of array[j] with array[j+1]
            if (array[j+1] < array[j]){
                int temp = array[j+1];
                array[j+1] = array[j];
                array[j] = temp;
                flagSwap = true;
            }
        }
    }
}
```

Bubble Sort

Selection sort puede ser un poco más eficiente, pues puede terminar su ejecución sin comparar todos los elementos.

- En el mejor escenario **(un arreglo ya ordenado)**, se realiza sólo una pasada por el arreglo.
- En el peor escenario **(el arreglo está ordenado descendientemente)**, cada pasada realiza $n-1$ intercambios y $n-1$ comparaciones.

En promedio, podemos decir que este algoritmo tiende a crecer **cuadráticamente**, al igual que el Selection Sort.

Algoritmos de Ordenamiento Comparados

