Fundamentos de programación

Algoritmos de búsqueda y ordenamiento

Cátedra de Fundamentos de Programación – Plan 2023

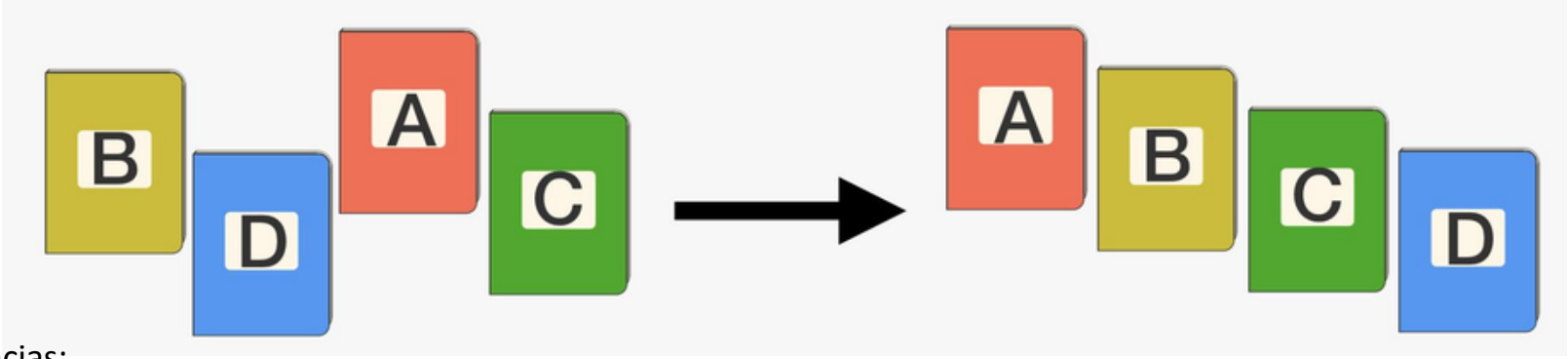
¿Qué veremos hoy?

- Slides del Prof. José Colbes

- Ordenamiento de datos

- Métodos directos
  - Selección (Selection Sort)
  - Inserción (Insertion Sort)
  - Burbuja (Bubble Sort) – Burbuja mejorada

- Ejercicios



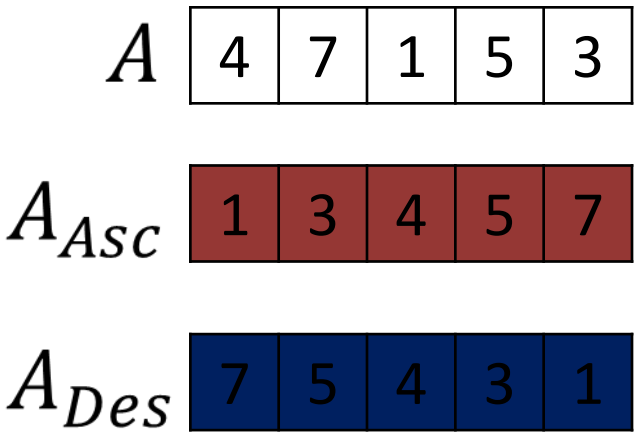
Referencias:
Algoritmos y estructuras de datos – Cap. 6 – Joyanes Aguilar
https://www.infor.uva.es/~mserrano/EDI/cap4.pdf

Ordenamiento de datos

Consiste en reorganizar un conjunto de datos u objetos en una secuencia específica, la cual puede ser de dos formas:

Ascendente $i < j \rightarrow A[i] \leq A[j]$

Descendente $i < j \rightarrow A[i] \geq A[j]$



Alumnos:

- Carlos Sauer
- Diego Stalder
- José Colbes
- Viviana Ortellado
- Juan Ovelar
- David Fretes

Por apellidos (Asc):

- José Colbes
- David Fretes
- Viviana Ortellado
- Juan Ovelar
- Carlos Sauer
- Diego Stalder

Ordenamiento de datos

El proceso de ordenamiento es uno de los ejemplos más interesantes para mostrar que existen múltiples soluciones para un mismo problema, y que cada algoritmo tiene sus propias ventajas y desventajas.

Según la cantidad de datos que se manejan, los algoritmos de ordenamiento se dividen en grupos:

Directos

- Selección
- Burbuja
- Inserción

Indirectos

- Mezcla
- Quicksort (Rápida)
- Shell
- Radix

¿Cuál elegimos?

La ***eficiencia*** es el factor que mide la calidad y rendimiento de un algoritmo. Área de estudio: Análisis de Algoritmos.

Criterios:

a) El de menor tiempo de ejecución.

b) El de menor número de instrucciones.

Sin embargo, no siempre es fácil efectuar estas medidas. Por ello, el mejor criterio es aislar una operación específica del algoritmo y contar el número de veces que se realiza.

Eficiencia
(algoritmos de ordenamiento)

Número de comparaciones entre elementos

Métodos directos de ordenamiento

Por Selección

Por Inserción

Por Burbuja

Métodos en Listas .sorted()

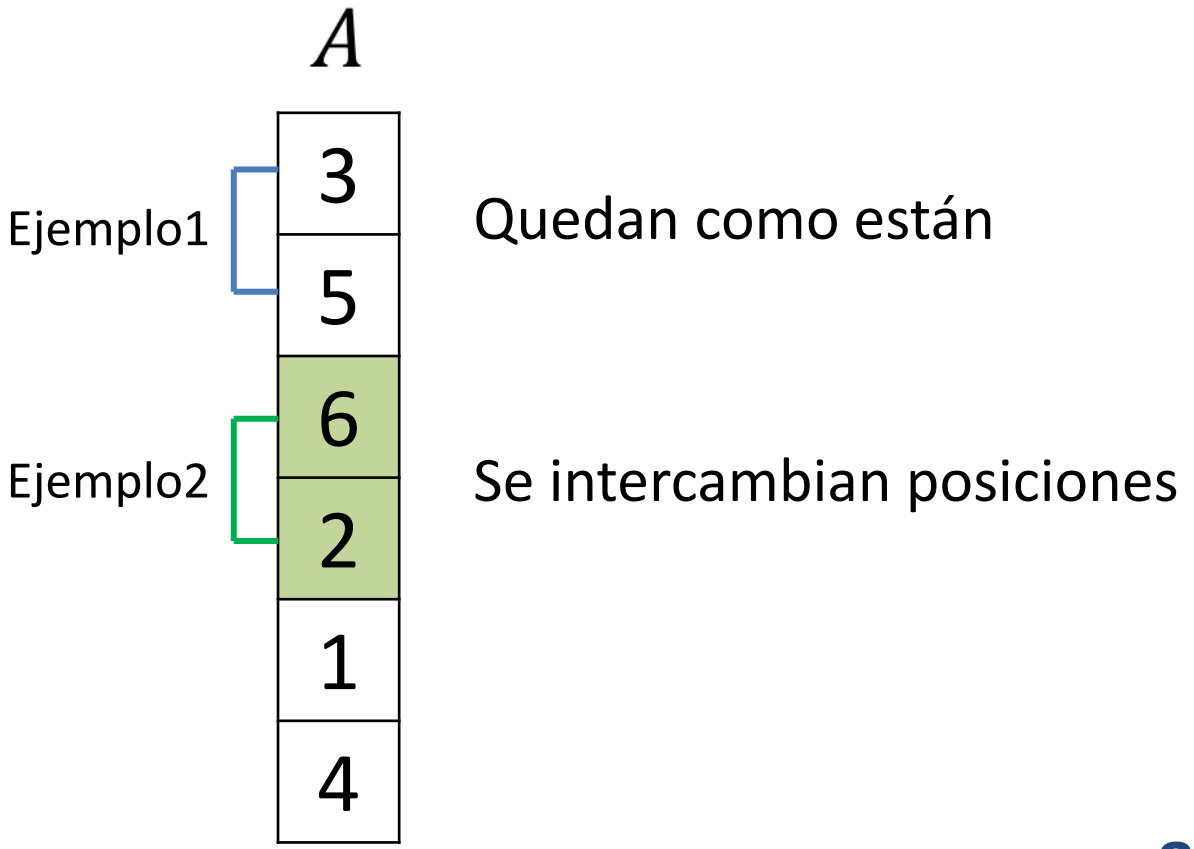
Algoritmos de Ordenamiento - Burbuja

Consiste en realizar ciclos repetidamente a través de la lista, comparando elementos adyacentes de dos en dos.

Si un elemento es mayor que el que está en la siguiente posición se intercambian.

Ordenamiento por burbuja

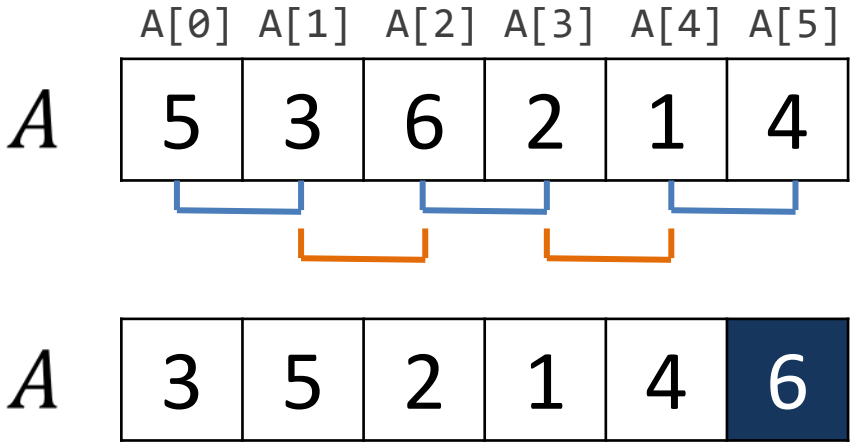
En cada iteración del algoritmo, se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente (o los valores son idénticos), se dejan como están. Si una pareja está en orden decreciente, sus valores se intercambian en el arreglo.



Ordenamiento por burbuja

El ordenamiento por burbuja requiere hasta `N-1` iteraciones. En cada iteración se comparan elementos adyacentes y se intercambian sus valores cuando el primer elemento es mayor que el segundo elemento. Al final de cada pasada, el elemento mayor se ha “hundido” hasta el fondo del sub-arreglo.

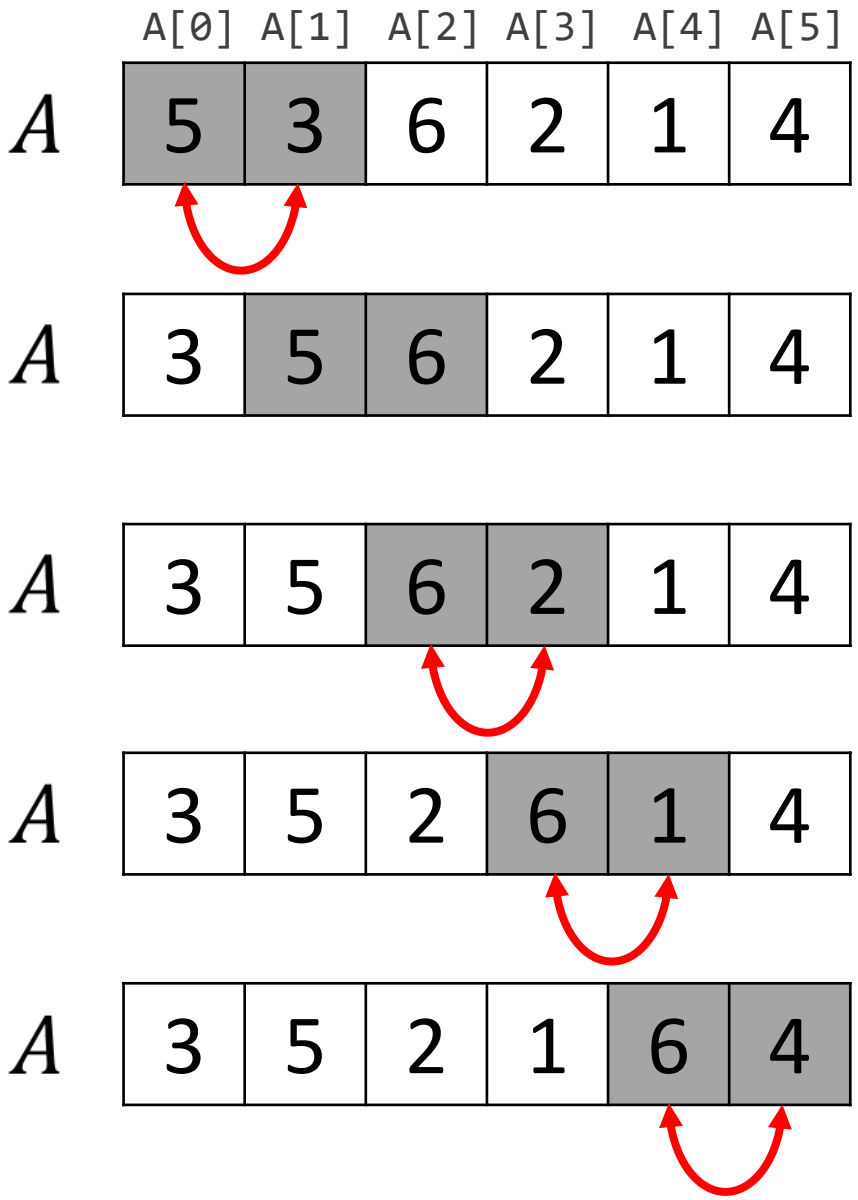
Por ejemplo, en la primera iteración se comparan los elementos adyacentes `(A[0],A[1]),(A[1],A[2]),(A[2],A[3]),...,(A[n-2],A[n-1])`:



Al final de esta iteración, el elemento máximo del arreglo se encuentra en el extremo derecho. Este proceso se repite con el sub-arreglo “desordenado”.

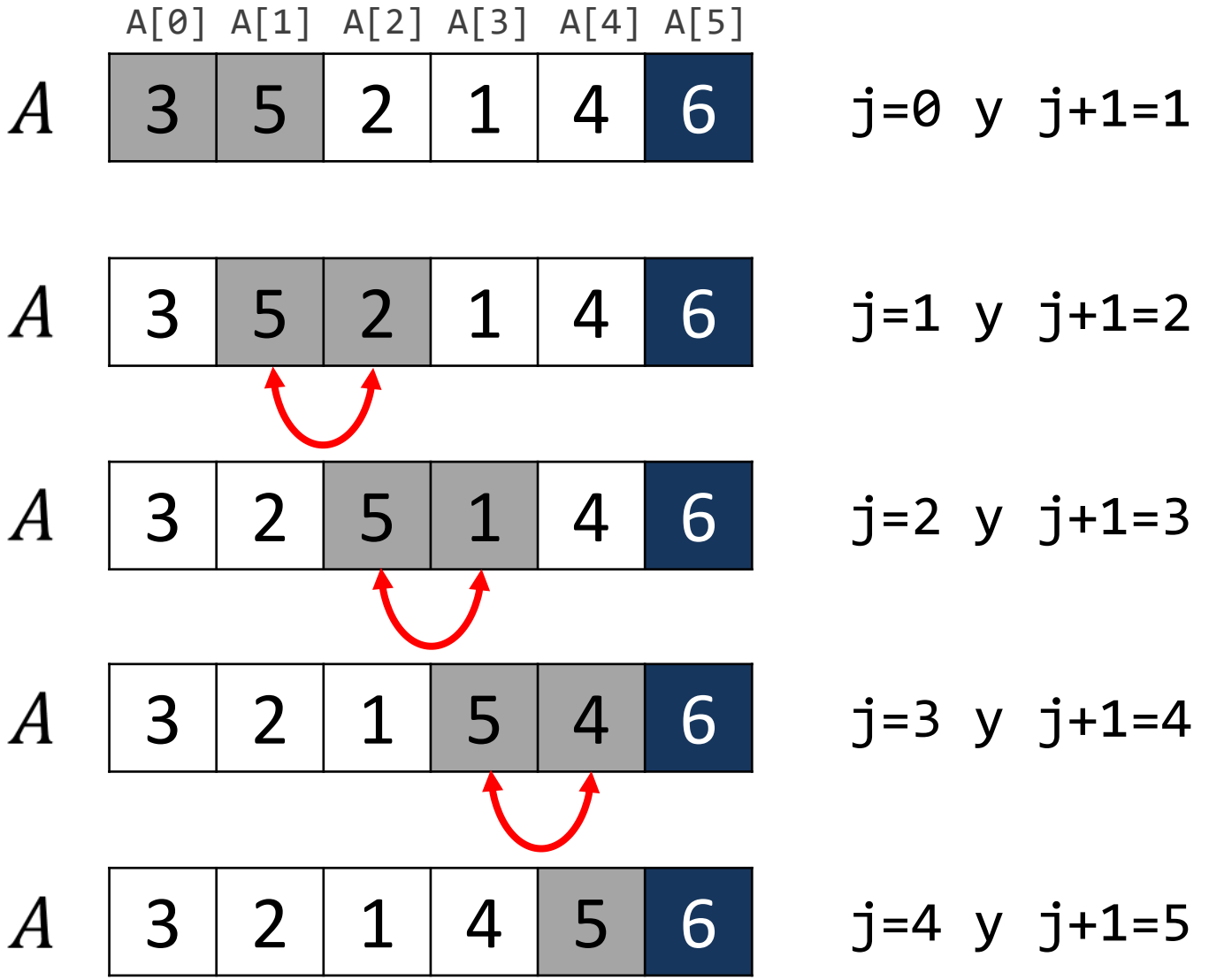
Ejemplo – Burbuja

`i=0`



Ejemplo – Burbuja

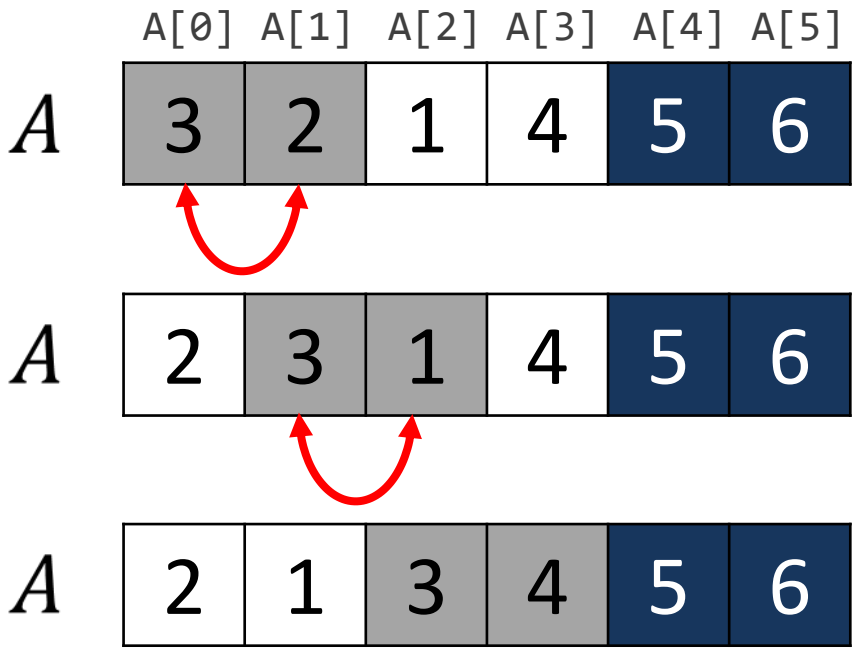
`i=1`



Fue necesaria esta última comparación?

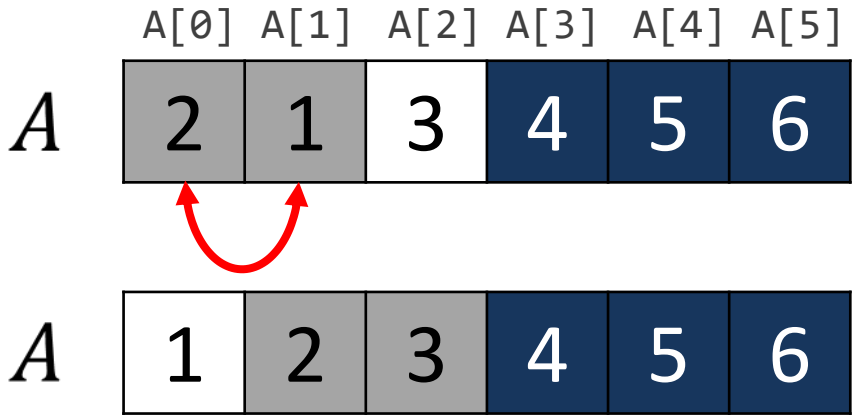
Ejemplo – Burbuja

`i=2`



Ejemplo – Burbuja

`i=3`



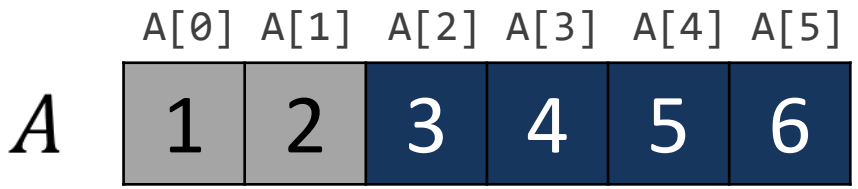
`j=0 y j+1=1`

`j=1 y j+1=2`

Acá ya está ordenado!

Ejemplo – Burbuja

`i=4`



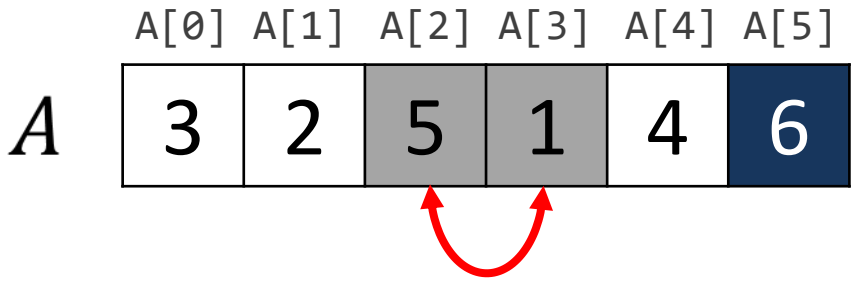
`j=0 y j+1=1`

Proceso finalizado!!

Algoritmo – Burbuja

Seudocódigo

```
Desde i=0 hasta N-2:
        Desde j=0 hasta N-i-2:
                Si A[j]>A[j+1]:
                        Intercambiar A[j] y A[j+1]
```



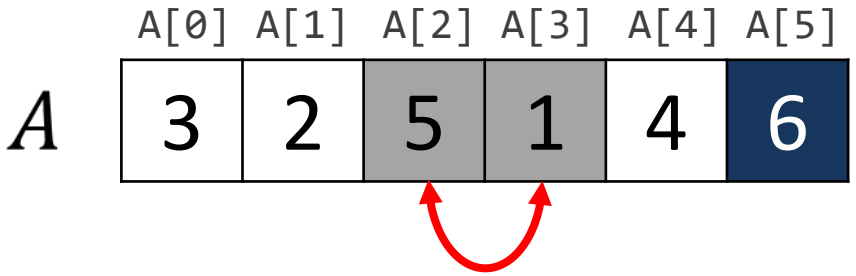
i=1

j=2 y j+1=3

Programa – Burbuja

```
def ordenar_vec_burbuja(A):
    n = len(A)
    for i in range(n - 1):
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                # Intercambiar A[j] y A[j+1]
                A[j], A[j + 1] = A[j + 1], A[j]

# Ejemplo de uso
A = [64, 34, 25, 12, 22, 11, 90]
ordenar_vec_burbuja(A)
print("Arreglo ordenado:", A)
```



i=1

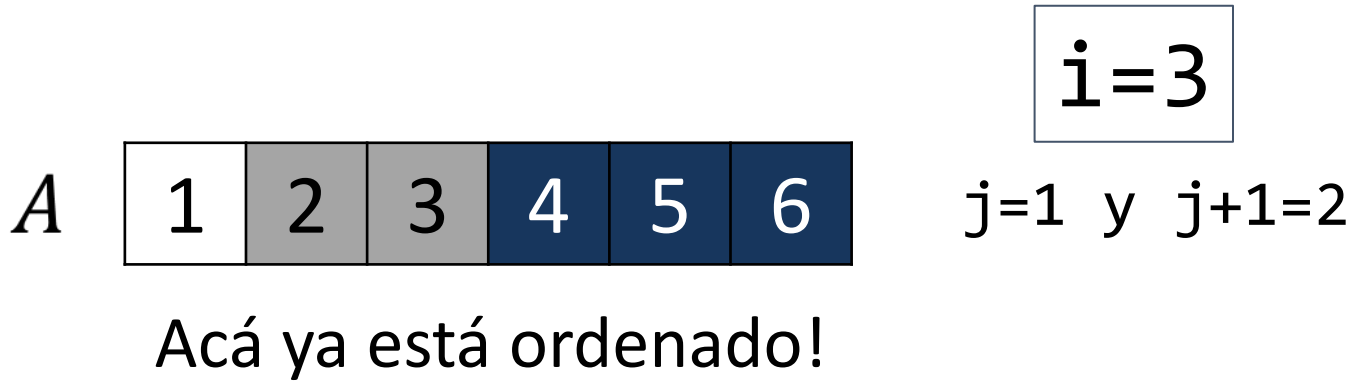
j=2 y j+1=3

Mejora del método de burbuja

El algoritmo tiene una mejora inmediata, el proceso de ordenación puede terminar en `N-1` iteraciones, o bien antes. Si en una iteración **no se produce intercambio alguno** entre elementos del vector es porque el vector **ya está ordenado**.

Así, se introduce una variable `continuar` (funciona como interruptor) para detectar si se ha producido intercambio en la pasada.

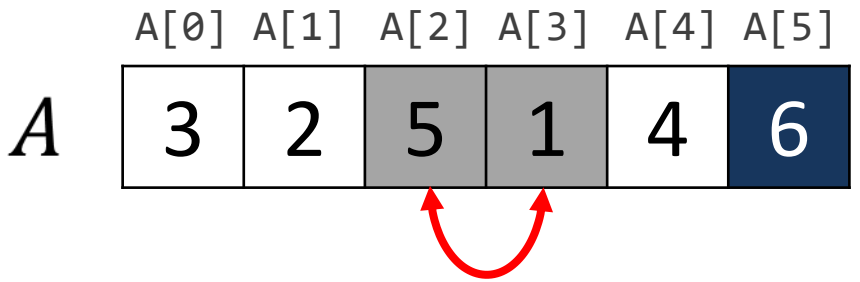
En nuestro ejemplo, para `i=3` ya se tenía el vector ordenado; por lo que el proceso podría finalizar allí.



Algoritmo – Burbuja Mejorada

Seudocódigo

```
i=0
Repetir:
        continuar = Falso
        Desde j=0 hasta N-i-2:
                Si A[j]>A[j+1]:
                        Intercambiar A[j] y A[j+1]
                        continuar = Verdadero
        i=i+1
Hasta continuar==Falso
```



i=1

j=2 y j+1=3

Programa – Burbuja mejorada

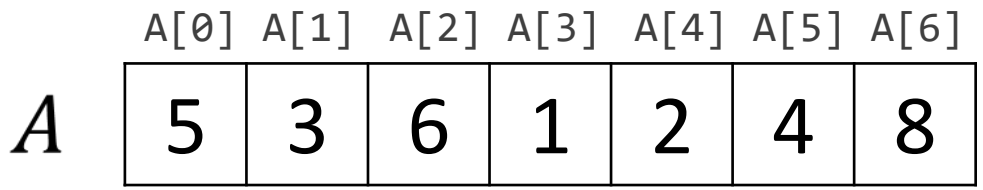
```
def ordenar_vec_burbuja_mejora(A):
    n = len(A)
    i = 0
    continuar = True
    while continuar:
        continuar = False
        for j in range(n - i - 1):
            if A[j] > A[j + 1]:
                # Intercambiar A[j] y A[j+1]
                A[j], A[j + 1] = A[j + 1], A[j]
                continuar = True
        i += 1

# Ejemplo de uso
A = [64, 34, 25, 12, 22, 11, 90]
ordenar_vec_burbuja_mejora(A)
print("Arreglo ordenado:", A)
```

Algoritmos de búsqueda

Supongamos que queremos saber si un elemento dado se encuentra en cierto vector de tamaño N. Si está, queremos conocer también su posición.

La primera forma sería a través de un método secuencial, en el cual se recorren los elementos del vector de inicio a fin. En el peor caso, deberíamos hacer N consultas.



Está el número 7 aquí?

Algoritmos de Ordenamiento - Inserción

Por ejemplo, cuando se tiene un mazo de cartas y se lo quiere ordenar: Hay que tomar la primera carta y colocarla en la mano.

Luego se toma la segunda y la compara con la que se tiene: si es mayor, se la coloca a la derecha, y si es menor a la izquierda.

Después se toma la tercera y se la compara con las que se tiene en la mano, desplazándola hasta que quede en su posición final.

Si se continúa haciendo esto, se estará insertando cada carta en la posición que le corresponde, hasta quedar todas en orden.

Ordenamiento por inserción

El método de ordenación por inserción es similar al proceso típico de ordenar tarjetas de nombres (cartas de una baraja) por orden alfabético, que consiste en insertar un nombre en su posición correcta dentro de una lista o archivo que ya está ordenado.

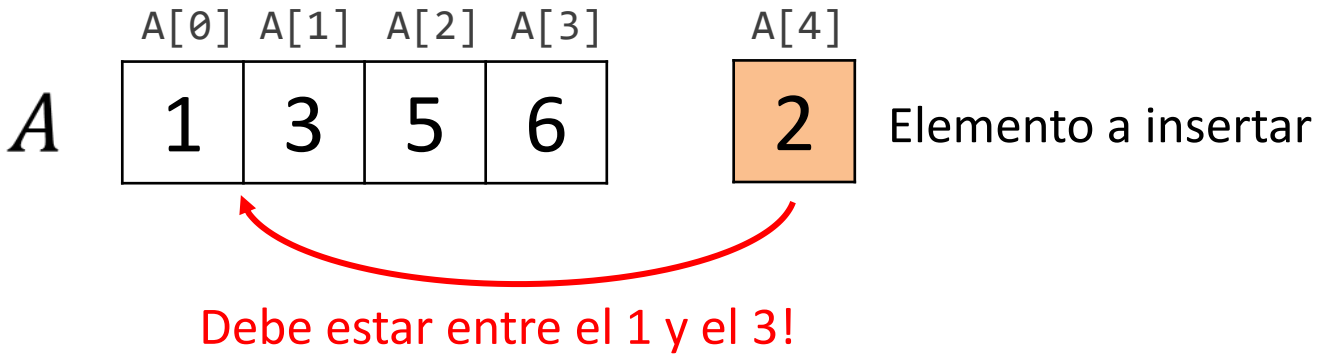
Supongamos de vuelta que deseamos ordenar de manera ascendente el siguiente vector A (donde N=6):

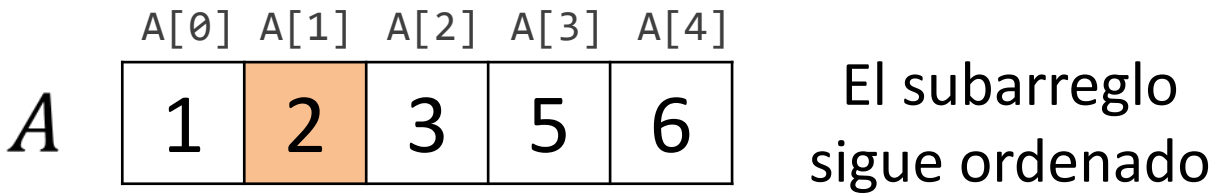
	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
A	5	3	6	1	2	4

Ordenamiento por inserción

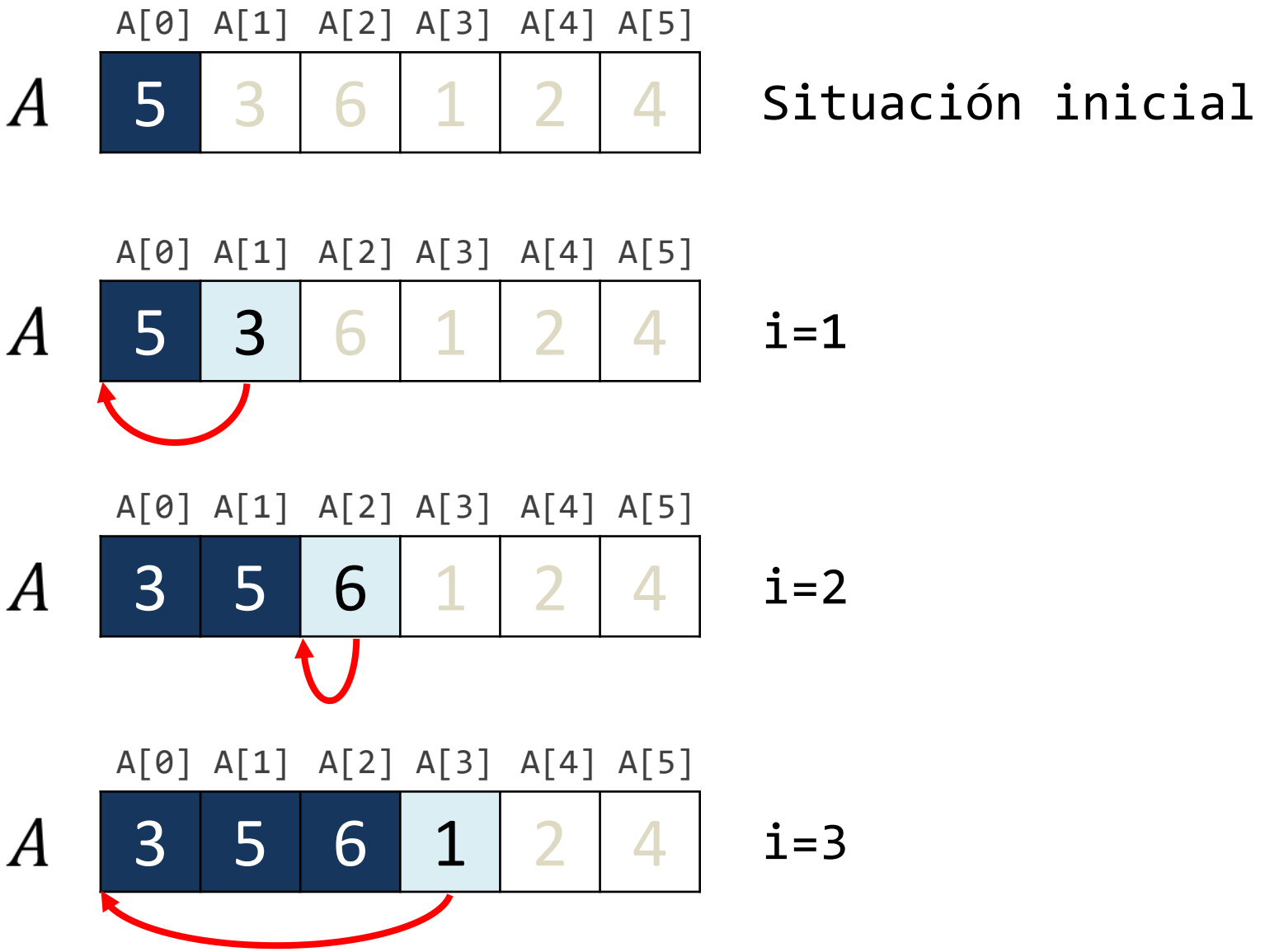
El método se basa en considerar una parte de la lista (subarreglo) ya ordenada y situar cada uno de los elementos restantes insertándolo en el lugar que le corresponde por su valor. Todos los valores a la derecha se desplazan una posición para “dejar espacio”.

Supongamos que ya tenemos ordenados A[0], A[1], A[2], y A[3]:

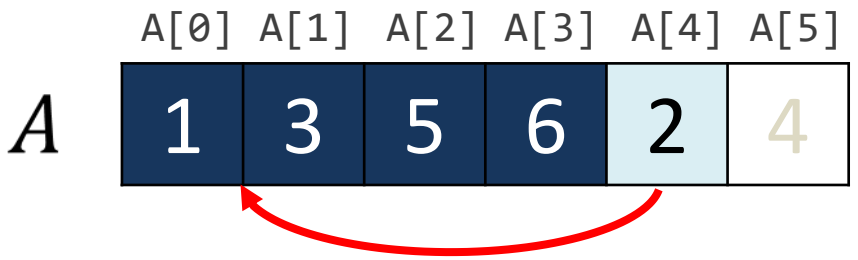




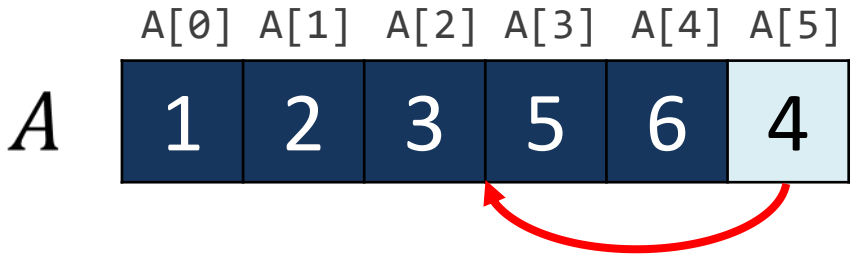
Ejemplo – Inserción



Ejemplo – Inserción



i=4



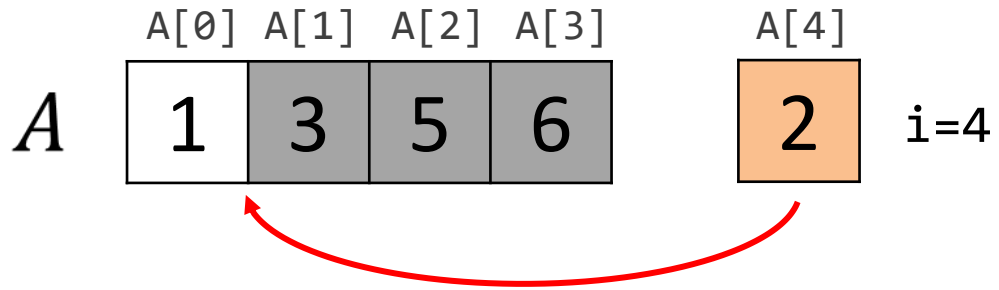
i=5

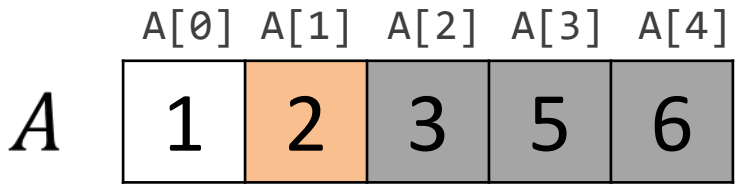
Arreglo ordenado!

Algoritmo – Inserción

Seudocódigo

```
Desde i=1 hasta N-1:
    • Guardar el valor de A[i] en una variable auxiliar.
    • Hacer espacio para este valor desplazando todos los
      valores mayores que aux una posición a la derecha.
    • Insertar el valor de aux en el lugar del último
      valor desplazado
```



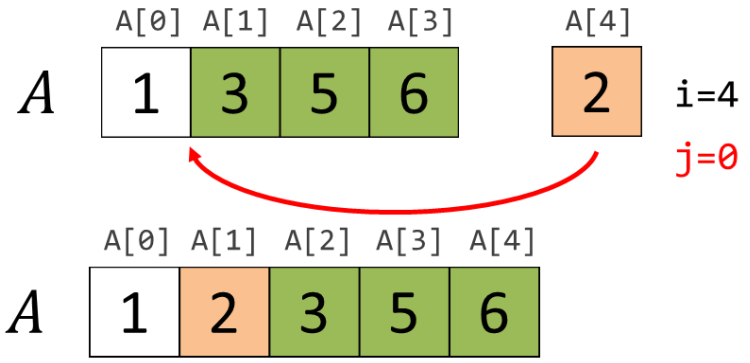


Fueron desplazados un lugar a la derecha

Programa – Inserción

```
def ordenar_vec_insercion(A):
    n = len(A)
    for i in range(1, n):
        aux = A[i]
        pos_encontrada = False  # indica si ya se encontró la posición del
nuevo elemento
        j = i - 1
        while j >= 0 and not pos_encontrada:
            if A[j] > aux:
                # Se desplaza a la derecha
                A[j + 1] = A[j]
                j -= 1
            else:
                pos_encontrada = True
        # Se coloca el nuevo elemento
        A[j + 1] = aux

# Ejemplo de uso
A = [64, 34, 25, 12, 22, 11, 90]
ordenar_vec_insercion(A)
print("Arreglo ordenado:", A)
```



Algoritmos de Ordenamiento - Selección

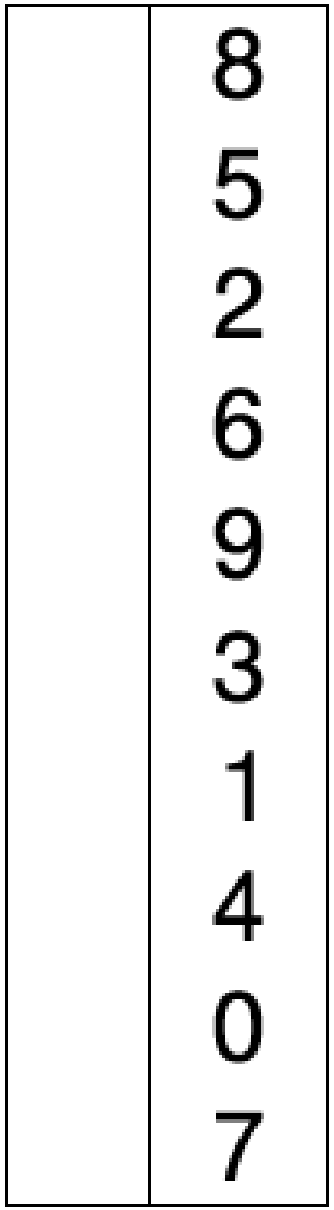
Buscar el elemento más pequeño de la lista.

Intercambiar con el elemento ubicado en la primera posición de la lista.

Buscar el segundo elemento más pequeño de la lista.

Intercambiar con el elemento que ocupa la segunda posición en la lista.

Repetir este proceso hasta que esté ordenada toda la lista.



Ordenamiento por selección

Consiste en escoger el elemento del arreglo (mayor/menor) y situarlo en uno de los extremos del arreglo de N elementos, y el elemento que estaba en esa posición intercambiarle por la posición del elemento (mayor/menor).

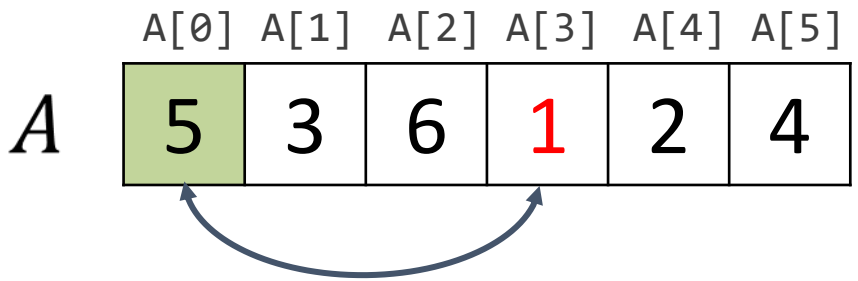
Supongamos que deseamos ordenar de manera ascendente el siguiente vector A (donde N=6):

	A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
A	5	3	6	1	2	4

En `A[0]` debe estar el elemento más pequeño, en `A[1]` el siguiente más pequeño, y así seguimos hasta llegar a `A[n-1]` (el elemento más grande). Por lo tanto, el primer paso en el ejemplo sería intercambiar los valores de `A[0]` y `A[3]`.

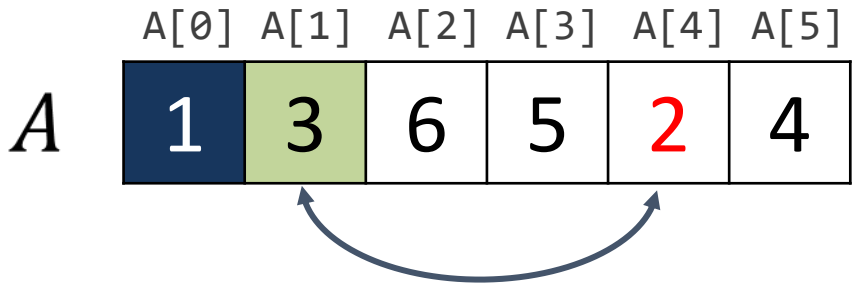
Así, en `A[i]` debe estar el elemento más pequeño del sub-arreglo aún desordenado `A[i],A[i+1]…,A[n-1].`

Ejemplo – Selección



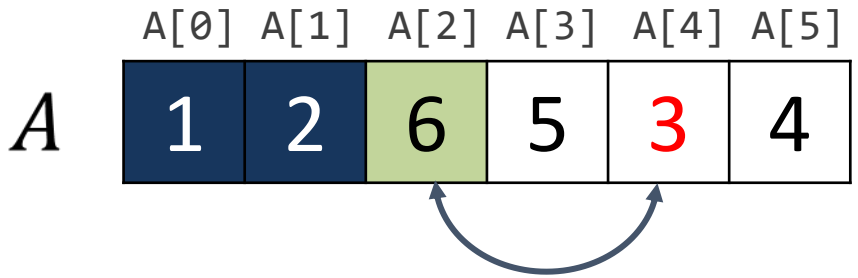
i=0

Intercambiar A[0] y A[3]



i=1

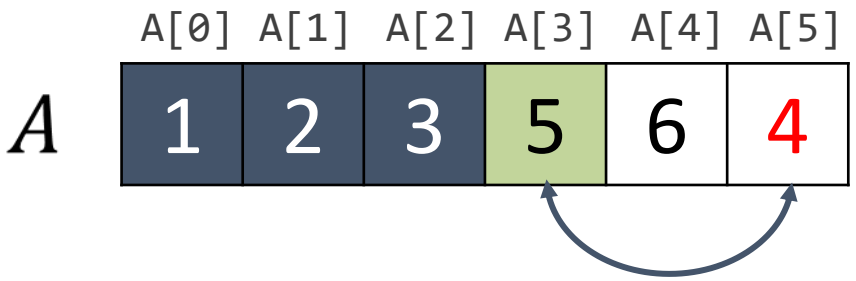
Intercambiar A[1] y A[4]



i=2

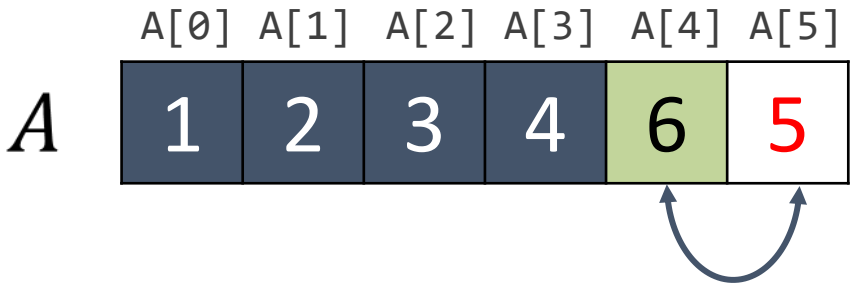
Intercambiar A[2] y A[4]

Ejemplo – Selección



i=3

Intercambiar A[3] y A[5]



i=4

Intercambiar A[4] y A[5]

i=5

Lista ordenada!!

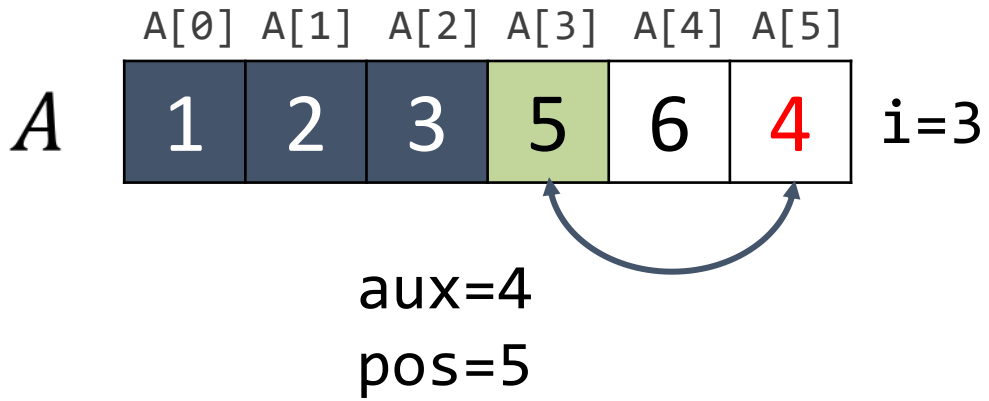
Algoritmo – Selección

Seudocódigo

```
Desde i=0 hasta N-1:
    • Encontrar el menor elemento del sub-arreglo
      A[i],A[i+1],…,A[n-1]
    • Si el menor elemento no está en la posición i,
      entonces intercambiar los valores de las posiciones
      i y la del menor elemento del sub-arreglo.
```

Programa – Selección

```
def ordenar_vec_seleccion(A):
    n = len(A)
    for i in range(n):
        aux = A[i]
        pos = i
        for j in range(i + 1, n):
            if A[j] < aux:
                aux = A[j]
                pos = j
        # Intercambio
        A[pos], A[i] = A[i], aux
# Ejemplo de uso
A = [64, 34, 25, 12, 22, 11, 90]
ordenar_vec_seleccion(A)
print("Arreglo ordenado:", A)
```



Ordenamiento en listas de python

El método sort() se utiliza para ordenar los elementos de una lista determinada

list.sort(*, key=None, reverse=False)

Parámetros:

key: Opcional. Una función que especifica los criterios de clasificación.

reverse: el valor predeterminado de inverso es Falso.
False ordena la lista en orden ascendente y
True ordena la lista en orden descendente.

Valor de retorno: El método sort() no devuelve ningún valor. Más bien, modifica la lista original

Ejemplo 1: ordenar una lista en orden ascendente/descendente

```
A = [64, 34, 25, 12, 22, 11, 90]
print("Arreglo :", A)
A.sort()
print("Arreglo ordenado:", A)
A.sort(reverse=True)
print("Arreglo ordenado:", A)
```

Ordenamiento en listas de python

```
Tipo de lista: Ordenada, Tamaño: 1000
ordenar_vec_burbuja: 0.038999 segundos
ordenar_vec_seleccion: 0.037003 segundos
ordenar_vec_insercion: 0.000000 segundos
sorted: 0.000000 segundos

Tipo de lista: Inversa, Tamaño: 1000
ordenar_vec_burbuja: 0.110002 segundos
ordenar_vec_seleccion: 0.039001 segundos
ordenar_vec_insercion: 0.070000 segundos
sorted: 0.000000 segundos

Tipo de lista: Aleatoria, Tamaño: 1000
ordenar_vec_burbuja: 0.071001 segundos
ordenar_vec_seleccion: 0.037000 segundos
ordenar_vec_insercion: 0.036000 segundos
sorted: 0.000000 segundos
```

Ordenamiento en listas de python

El método sorted() en Python utiliza un algoritmo de ordenamiento llamado Timsort. Timsort es una combinación de los algoritmos de ordenamiento por inserción y ordenamiento por fusión.

Algorithm	Time Complexity			Space Complexity
	Best	Average	Worst	Worst
Quicksort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(log(n))
Mergesort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(n)
Timsort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)
Heapsort	Ω(n log(n))	θ(n log(n))	O(n log(n))	O(1)
Bubble Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Insertion Sort	Ω(n)	θ(n^2)	O(n^2)	O(1)
Selection Sort	Ω(n^2)	θ(n^2)	O(n^2)	O(1)
Tree Sort	Ω(n log(n))	θ(n log(n))	O(n^2)	O(n)
Shell Sort	Ω(n log(n))	θ(n(log(n))^2)	O(n(log(n))^2)	O(1)
Bucket Sort	Ω(n+k)	θ(n+k)	O(n^2)	O(n)
Radix Sort	Ω(nk)	θ(nk)	O(nk)	O(n+k)
Counting Sort	Ω(n+k)	θ(n+k)	O(n+k)	O(k)
Cubesort	Ω(n)	θ(n log(n))	O(n log(n))	O(n)

https://hackernoon.com/timsort-the-fastest-sorting-algorithm-youve-never-heard-of-36b28417f399

Algoritmos de Búsqueda

Algoritmos de Búsqueda

1. Secuencial:

Es un algoritmo bastante sencillo e intuitivo: consiste en buscar el elemento comparándolo secuencialmente con cada elemento del vector hasta encontrarlo, o hasta que se llegue al final.

Algoritmos de Búsqueda

1. Secuencial:

```
def busqueda_secuencial(arr, x):
    for i in range(len(arr)):
        if arr[i] == x:
            return i
    return -1

# Ejemplo de uso
lista = [3, 6, 2, 8, 1, 9, 5]
elemento_a_buscar = 8
resultado = busqueda_secuencial(lista,
elemento_a_buscar)
if resultado != -1:
    print(f"El elemento {elemento_a_buscar}
está presente en el índice {resultado}.")
else:
    print(f"El elemento {elemento_a_buscar} no
está presente en la lista.")
```

Algoritmos de Búsqueda

2. Binaria:

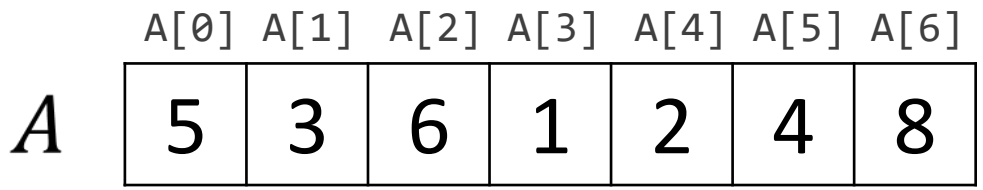
Requiere que la lista esté ordenada previamente.

1. Se compara el elemento a buscar con un elemento cualquiera de la lista (normalmente el elemento central):
2. Si el valor de éste es mayor que el del elemento buscado se repite el procedimiento en la parte de la lista que va desde el inicio y hasta el elemento tomado.
3. En caso contrario se toma la parte de la lista que va desde el elemento tomado hasta el final. De esta manera obtenemos intervalos cada vez más pequeños, hasta que se obtenga un intervalo indivisible.
4. Si el elemento no se encuentra dentro de este último entonces se deduce que el elemento buscado no se encuentra en la lista.

Algoritmos de búsqueda

Supongamos que queremos saber si un elemento dado se encuentra en cierto vector de tamaño N. Si está, queremos conocer también su posición.

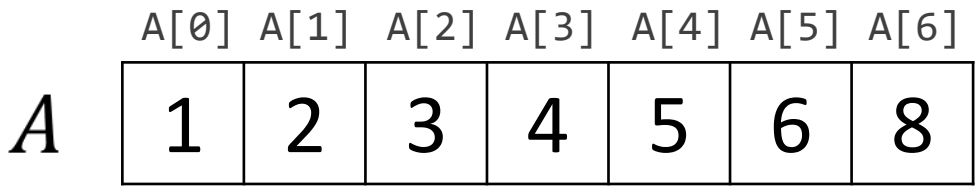
La primera forma sería a través de un método secuencial, en el cual se recorren los elementos del vector de inicio a fin. En el peor caso, deberíamos hacer N consultas.



Está el número 7 aquí?

Búsqueda binaria

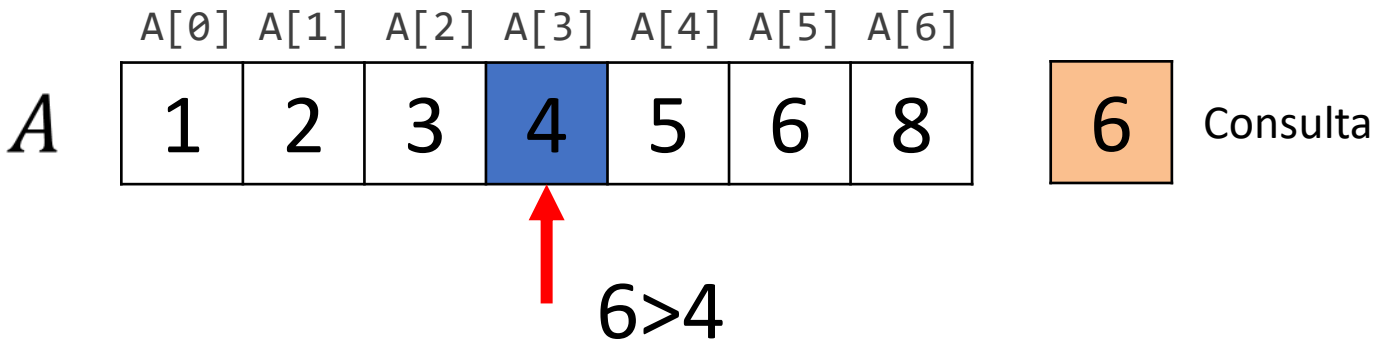
Otra forma es a través de un algoritmo de búsqueda binaria, el cual necesita que todos los elementos del vector estén ordenados.



De qué nos sirve que los números estén ordenados?

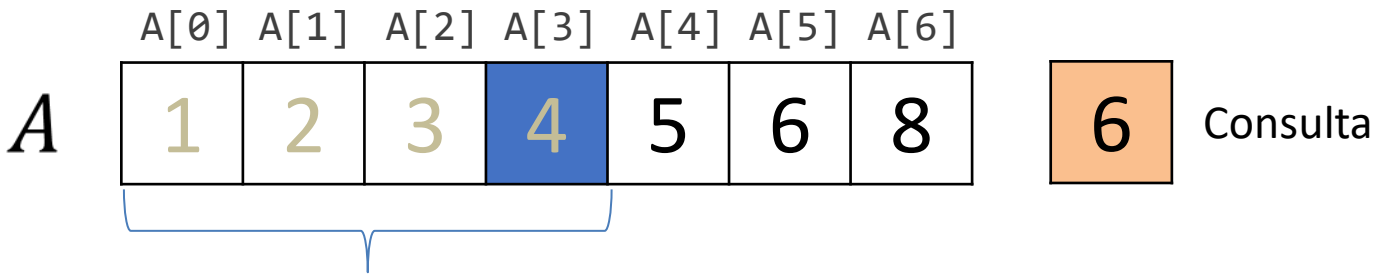
Búsqueda binaria

Empecemos posicionándonos en la mitad del arreglo.



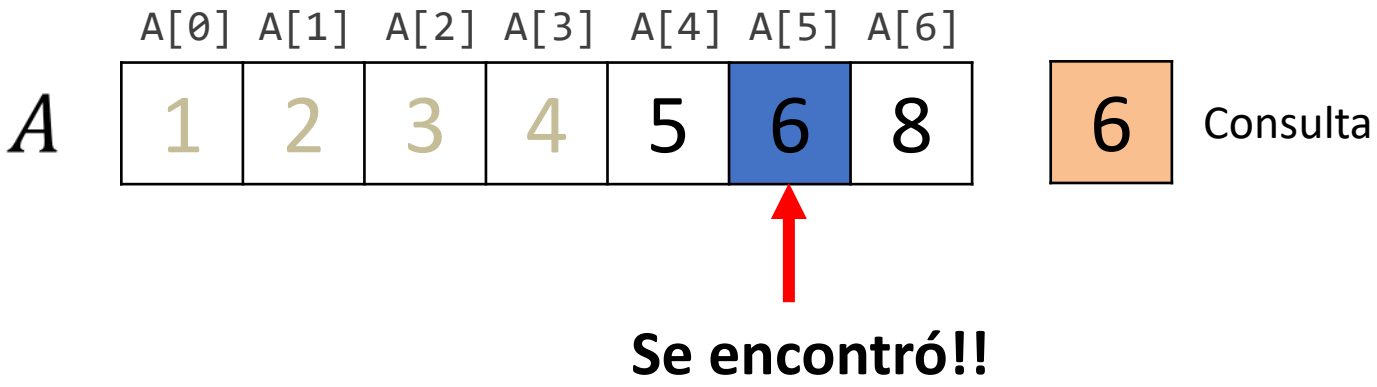
Búsqueda binaria

Empecemos posicionándonos en la mitad del arreglo.



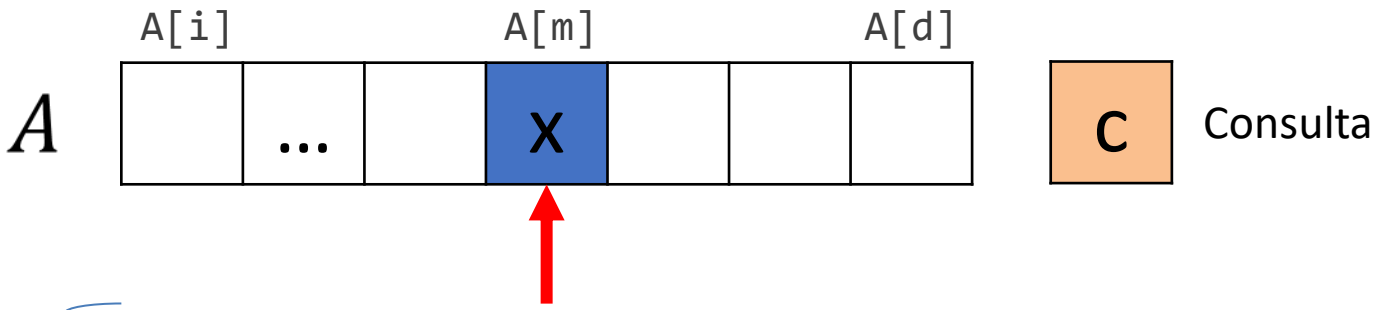
Búsqueda binaria

Ahora nos posicionaremos a la mitad del sub-arreglo restante:



Búsqueda binaria

Entonces, si i (izquierda) y r (derecha) son los extremos del subarreglo que queda, vamos al elemento que se encuentra a la mitad de dicho sub-arreglo; y tomamos una decisión en función a la comparación con la consulta.



Si:

- `x=c` Se encontró el elemento en la posición `m`
- `x>c` Se podría encontrar en el sub-arreglo izquierdo
- `x<c` Se podría encontrar en el sub-arreglo derecho

Búsqueda binaria

```
def busqueda_binaria(arr, x):
    izquierda, derecha = 0, len(arr) - 1
    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if arr[medio] == x:
            return medio
        elif arr[medio] < x:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    # Si el elemento no está presente en la lista
    return -1
lista = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
elemento_a_buscar = 13
resultado = busqueda_binaria(lista, elemento_a_buscar)
if resultado != -1:
    print(f"El elemento {elemento_a_buscar} está presente en el índice
{resultado}.")
else:
    print(f"El elemento {elemento_a_buscar} no está presente en la lista.")
```

Algoritmos de Búsqueda

3. Index:

El método index() se utiliza para obtener el índice de base cero en la lista del primer elemento cuyo valor se proporciona.

```
lista = [3, 6, 2, 8, 1, 9, 5]
elemento_a_buscar = 8
try:
    indice = lista.index(elemento_a_buscar)
    print(f"El elemento {elemento_a_buscar}
está presente en el índice {indice}.")
except ValueError:
    print(f"El elemento {elemento_a_buscar} no
está presente en la lista.")
```

Algoritmos de Búsqueda

Comparación:

```
Resultado de la búsqueda secuencial: 999998, Tiempo: 0.04899930953979492 segundos
Resultado de la búsqueda binaria: 999998, Tiempo: 0.0 segundos
Resultado de la búsqueda con index(): 999998, Tiempo: 0.010000228881835938 segundos
```

Ejercicio 1

Se cuenta con una lista L1 de números enteros en el cual existen numerosos valores repetidos. A fin de economizar el espacio de almacenamiento, se desea crear una nueva lista L2 en la cual cada valor diferente aparece una sola vez, sin repetición, pero indicando la cantidad de veces que se repite dicho valor en la lista L1.

Ejemplo:
Lista original L1 (dato para el algoritmo)

23	27	8	14	23	23	8	23	27	23	27	8	27

Lista final L2 (resultado del algoritmo)

8	3	14	1	23	5	27	4

La interpretación de la lista L2 es como sigue: el valor 8 aparece 3 veces en L1, el valor 14 aparece 1 vez en L1, el valor 23 aparece 5 veces en L1, y el valor 27 aparece 4 veces en L1.

Note que los valores de L1 deben aparecer ordenados ascendentemente en L2 (8-14-23-27).

Conclusión

Los algoritmos de ordenación son de uso diario hoy en día y prácticamente se lo aplica en forma inconsciente en algunos casos.

Contar con datos ordenados es uno de los objetivos del manejo de datos para análisis o búsqueda.

El algoritmo más fácil de implementar es el de «burbuja», sin embargo, no es eficiente para listas con muchos elementos.

El algoritmo «quick sort» es uno de los más rápidos para listas con una enorme cantidad de elementos, pero no es eficiente para listas con poca cantidad de elementos.

Bibliografía recomendada

- Marzal, Andres; Gracia, Isabel; García, Pedro. *Introducción a la programación con Python 3*. Universitat Jaume I, 2014.
- Matthes, Eric (traducción de Pineda, Beatriz). *Curso intensivo de Python: Introducción práctica a la programación basada en proyectos*. 2da edición. Anaya Multimedia, 2021.