# Algoritmo de Ordenamiento por Inserción

Computer Science

CS1100 - Introducción a Ciencia de la Computación



# Logro de la Sesión

### Al finalizar esta sesión, estarás en la capacidad de:

■ Comprender el algoritmo de ordenamiento por inserción.



# Logro de la Sesión

### Al finalizar esta sesión, estarás en la capacidad de:

- Comprender el algoritmo de ordenamiento por inserción.
- Implementar el algoritmo de ordenamiento por inserción en Python.



## **Insertion Sort**

## Revisión del algoritmo...

■ £Cúal es el algoritmo del Insertion Sort?



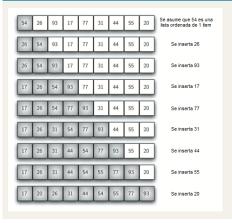
## **Insertion Sort**

### Revisión del algoritmo...

- £Cúal es el algoritmo del Insertion Sort?
- Tal como observamos en el vídeo, el proceso consiste en revisar cada elemento del conjunto, y, de ser este elemento menor al que está a su izquierda, deberá cambiar de posición con el de su izquierda. Esta comparación con el de su izquierda se repite hasta que encuentre su posición adecuada. Finalmente, se pasa al siguiente elemento, y se vuelve a realizar el proceso.

## **Insertion Sort**

### £Qué ocurre luego de cada pasada.





#### Ordenar los elementos

£En qué nos fijamos primero cuando queremos ordenar una lista de menor a mayor?

```
[23,29, <@\textcolor{red}{13}@>, 34, 17, 25]
```

- Buscamos inmediatamente el menor elemento.
- Un procedimiento natural al ordenar es pensar primero en que el primer número va ala principio.
- £Cómo puede hacer esto el computador?



Escribe una función que reciba una lista y encuentre el menor elemento de la lista.

```
def indice menor elemento(lista):
       menor = lista[0]
2
       indice = 0
3
       for i in range(len(lista)):
4
           if lista[i] < menor:</pre>
5
                menor = lista[i]
6
                indice = i
       return indice
8
9
  lista = [23, 29, 13, 34, 17, 25]
10
  print(indice menor elemento(lista))
11
```

2



### £Qué imprime el siguiente código?

```
lista = [3,2,1]
i = indice_menor_elemento(lista)
aux = lista[i]
lista[i] = lista[0]
lista[0] = aux
print(lista)
```



### £Qué imprime el siguiente código?

```
lista = [3,2,1]
i = indice_menor_elemento(lista)
aux = lista[i]
lista[i] = lista[0]
lista[0] = aux
print(lista)
```

```
1 [1, 2, 3]
```



#### Ordenar los elementos

Una vez que pusimos el primer elemento en la primera posición, £Qué buscamos?

```
[ <@\textcolor{red}{13}@>, 29, 23, 34, <@\textcolor{
   blue}{17}@>, 25]
```

- Buscamos inmediatamente el menor elemento.
- Continuamos buscando el segundo elemento para situarlo en la segunda posición.
- Cuando queremos situar el segundo elemento en la seguna posición, ya sabemos que el menor elemento está en la primera posición.
- En este caso, buscar el segundo menor elemento es lo mismo que buscar el menor elemento en la sublista que parte de la segunda posición.

#### Ordenar los elementos

Una vez que pusimos el primer elemento en la primera posición, £Qué buscamos?

```
[ <@\textcolor{red}{13}@>, 29, 23, 34, <@\textcolor{
   blue}{17}@>, 25]
```

■ En general, al buscar el n-ésimo menor elemento sabremos que los n-1 menores elementos ya están donde corresponde



## £Cómo generalizamos este código?

```
lista = [3,4,2,1]
i = indice_menor_elemento(lista)
aux = lista[i]
lista[i] = lista[0]
lista[0] = aux
print(lista)
```

```
[1, 4, 2, 3]
```



## £Cómo generalizamos este código?

```
def ordenar(lista):
      for i in range(len(lista)):
           indice = indice_menor_elemento(
              lista[i:])
           aux = lista[indice + i]
           lista[indice + i] = lista[i]
           lista[i] = aux
6
7
  lista = [3, 4, 2, 1]
8
  ordenar(lista)
  print(lista)
10
```

[1, 2, 3, 4]

## £Cuántas operaciones realizar el algoritmo?

```
def ordenar (lista):
       for i in range(len(lista)):
           indice = indice_menor_elemento(
              lista[i:])
           aux = lista[indice + i]
           lista[indice + i] = lista[i]
           lista[i] = aux
6
7
  lista = [3, 4, 2, 1]
8
  ordenar(lista)
  print(lista)
10
```

- Para situar el menor elemento, realiza n-1 operaciones. UTEC
- Para situar el segundo menor, *n-2* operaciones.

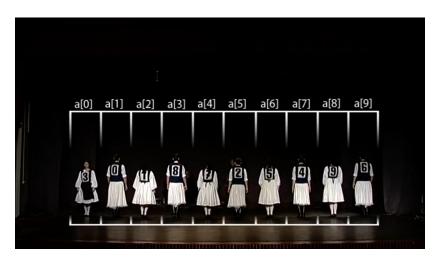
### £Cómo mejoramos el algoritmo?

```
def ordenar(lista):
       for i in range(len(lista)):
2
           indice = indice menor elemento(
3
              lista[i:])
           aux = lista[indice + i]
4
           lista[indice + i] = lista[i]
5
           lista[i] = aux
6
7
  lista = [3, 4, 2, 1]
8
  ordenar(lista)
  print(lista)
10
```

- Cambiaremos el enfoque de nuestro algoritmo.
- En vez de preocuparnos de situar el i-ésimo elemento en la



## **Insert Sort**



Ver este video



## **Insert Sort**

- £Cómo lo programamos?
- Debemos mover cada elemento (de izquierda a derecha), tan a la izquierda como sea necesario, intercambiándolo en cada paso con su vecino.

```
def ordenar (lista):
       for i in range(1, len(lista)):
2
            h=i
3
            while h > 0 and lista[h] < lista[h</pre>
4
               - 11:
                aux = lista[h]
5
                 lista[h] = lista[h - 1]
6
                lista[h - 1] = aux
7
                h=h-1
8
9
  lista = [3, 4, 2, 1]
10
  ordenar(lista)
11
```



# Insertion Sort - Tiempo de ejecución

### Tiempo de ejecución en diversos casos

- El peor caso: O(n x n).
- El mejor caso: O(n).
- El caso promedio para un arreglo aleatorio: O(n x n).
- El caso "casi ordenado": O(n).



# **Ejemplo 1**

#### Enunciado

Suponga que usted tiene que ordenar la siguiente lista de números: [15, 5, 4, 18, 12, 19, 14, 10, 8, 20] £Cuál de las siguientes listas representa la lista parcialmente ordenada tras tres pasadas completas del ordenamiento por inserción?

```
[4, 5, 12, 15, 14, 10, 8, 18, 19, 20]
[15, 5, 4, 10, 12, 8, 14, 18, 19, 20]
[4, 5, 15, 18, 12, 19, 14, 10, 8, 20]
[15, 5, 4, 18, 12, 19, 14, 8, 10, 20]
```



## Cierre

### Conclusiones

£Qué podemos decir sobre el tiempo de ejecución de este algoritmo?

