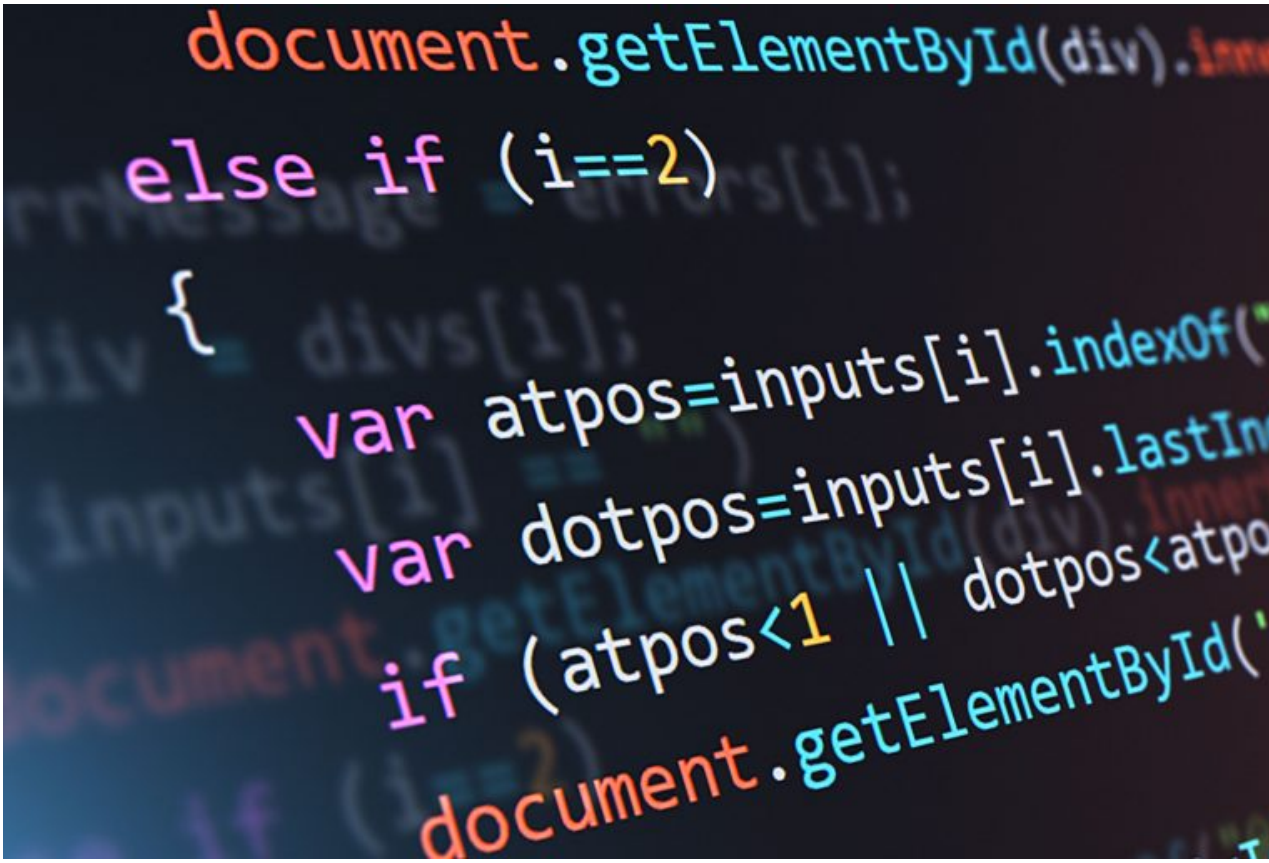
Hito grupal programación



Por Eloy Pérez, Iván Encinas, Sergio Maeso y Fernando Trujillo.

Cuáles son los diferentes algoritmos de ordenación y cómo funcionan

Bubble sort, u ordenamiento de burbuja en español, es un algoritmo de ordenamiento que busca mediante repetidas comparaciones de los elementos adyacentes ordenar un conjunto de elementos. Existen métodos más eficientes que el bubble sort, pero aún así tiene uso como herramienta de educación.

El algoritmo de **ordenamiento *por inserción*** consiste en seleccionar un elemento de la lista como clave y compararlo con sus adyacentes.

El algoritmo de **ordenamiento *por selección*** se basa en encontrar el menor de todos los elementos del vector e intercambiarlo con el que está en primera posición. Después del primero se hace con el segundo, con el tercero, y así sucesivamente.

Por último la **ordenación *Shell*** funciona comparando elementos que estén distantes. La distancia entre comparaciones decrece conforme el algoritmo va funcionando hasta la última fase, en la que se comparan los elementos adyacentes.

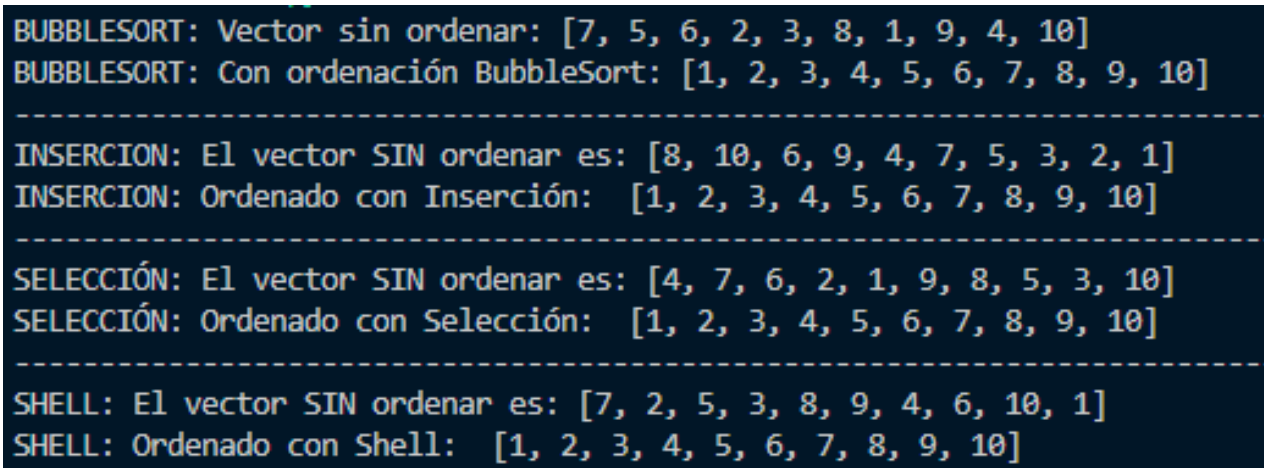
Introducción:

Hemos desarrollado el código en **python**, utilizando **visual studio code**. Hemos utilizado también **programación imperativa**.

Realmente todos los métodos *hacen lo mismo*. Primero se genera una lista aleatoria de 10 números comprendidos entre el 1-10 sin repetición y luego se va *ordenando* de una forma o de otra.

Siempre se *ordena* de forma *creciente* (1-2-3..-8-9-10) pero el código es lo que cambia. **Distintas formas de hacer lo mismo**.
Al final, simplemente es un contador del vector que se va sumando según respeta unos criterios o no. Así es como se va ordenando progresivamente según *sea mayor o menor el número a ordenar respecto del anterior*.

Lo común de estos métodos es que utilizan el método '*sample*' de la librería '*random*' para coger 10 elementos de la lista aleatoria. Al final, todos ordenados quedarán igual. *Este es el resultado que buscamos*:



Ahora vamos con el código:

Lo primero que todo, importamos lo que necesitaremos:

```
from random import sample
```

Y ahora procedemos a hacerlo de la primera manera, **BUBBLESORT**:

```
#CON BUBBLESORT:

#importamos el método 'sample' de la biblioteca random para generar
listas aleatorias

#hacemos una lista que almacene los numeros comprendidos entre el 1 y
el 11 (1,2,3,4,5,6,7,8,9,10)
lista_de_numeros = list(range(1, 11)) #Creamos la lista principal con
números del 0 al 10

#creamos una lista aleatoria con el método sample con los 10 elementos
aleatorios de la lista base
vector_bubblesort = sample(lista_de_numeros,10)

#creamos una función para hacer los métodos
def bubblesort(vector_bubblesort):
    #printeamos la lista SIN ordenar
    print("BUBBLESORT: Vector sin ordenar:",vector_bubblesort)

    n = 0 #creamos un contador del largo del vector
    for _ in vector_bubblesort: #bucle for _ in _:
        n += 1 #cuenta los caracteres de dentro del vector / se puede
poner también de forma: n = +1

    for l in range(n-1): #primer bucle del bucle anidado
    # Le damos un rango n para que complete el proceso.
        for u in range(0, n-l-1): #segundo bucle del bucle anidado
            #revisa la matriz de 0 hasta n-i-1 (que será 10 en nuestro
caso)
            if vector_bubblesort[u] > vector_bubblesort[u+1]:
                vector_bubblesort[u], vector_bubblesort[u+1] =
vector_bubblesort[u+1], vector_bubblesort[u]
            # Se intercambian SÓLO SI el elemento encontrado es mayor
(vector(u)>vector(u+1))
            #y posteriormente pasa al siguiente
```

```
    #ahora printeamos ya una vez hecha la ordenación BubbleSort
    print ('BUBBLESORT: Con ordenación BubbleSort:', vector_bubblesort)
bubblesort(vector_bubblesort) #invocamos la function

print('------------------------------------------------------------------
----------------------------------------')
```

Seguiremos avanzando, y nos toca hacer otra vez lo mismo, sólo que ahora lo ordenaremos por **INSERCIÓN**:

```
#CON INSERCIÓN

#hacemos una lista que almacene los numeros comprendidos entre el 1 y
el 11 (1,2,3,4,5,6,7,8,9,10)
lista_de_numeros_2 = list(range(1, 11)) # Creamos la lista base con
números del 1 al 100

#creamos una lista aleatoria con el método sample con los 10 elementos
aleatorios de la lista base
vector_insercion = sample(lista_de_numeros_2,10)

#creamos una función para hacer los métodos
def insertionsort(vector_insercion):
    #printeamos la lista SIN ordenar
    print("INSERCION: El vector SIN ordenar es:", vector_insercion)

    largo = 0 #creamos un contador del largo del vector

    for i in vector_insercion:
        largo += 1 #obtenemos el largo del vector. También se puede
escribir: largo = +1

    #recorremos la lista desde el primer valor hasta el último (1-11)
    for i in range(1, largo): #bucle for _ in _ :
        elemento = vector_insercion[i]

        #movemos los elementos de vector_insercion[0...i-1], que son
mayores que el elemento a una posición adelante de su posición actual
        j = i-1

        while j >= 0 and elemento < vector_insercion[j] : #bucle
while...
                vector_insercion[j+1] = vector_insercion[j]
                j -= 1 #se puede escribir también como: j = -1
```

```
        vector_insercion[j+1] = elemento

    #printeamos el vector YA ordenado por inserción
    print("INSERCION: Ordenado con Inserción: ", vector_insercion)
insertionsort(vector_insercion)

print('-------------------------------------------------------------------
----------------------------------------')  #separador
```

Una vez acabado, nos pondremos a trabajar para ordenarlo por **SELECCIÓN**:

```
#POR SELECCIÓN

#hacemos una lista que almacene los numeros comprendidos entre el 1 y
el 11 (1,2,3,4,5,6,7,8,9,10)
lista_de_numeros_3 = list(range(1, 11)) # Creamos la lista base con
números del 1 al 10

#creamos una lista aleatoria con el método sample con los 10 elementos
aleatorios de la lista base
vector_seleccion = sample(lista_de_numeros_3,10)

#creamos una función para hacer los métodos
def selectionsort(vector_seleccion):
    #imprimimos la lista sin la ordenación por selección
    print ("SELECCIÓN: El vector SIN ordenar es:",vector_seleccion)

    #establecemos el contador del largo del vector
    largo = 0

    #bucle for _ in _ : para obtener cada vez uno en el largo (hasta el
límite marcado)
    for _ in vector_seleccion:
        largo += 1 #obtenemos el largo del vector

    #otro bucle for _ in _ :
    for i in range(largo):
        #encontrar el minimo elemento de los restantes sin ordenar
        minimo = i
        #otro bucle for _ in _ : anidado al primero (bucle anidado)
        for j in range(i+1, largo):
            if vector_seleccion[minimo] > vector_seleccion[j]:
                minimo = j
```

```
        #cambiamos el elemento minimo encontrado con el primer elemento
de la "matriz"
        vector_seleccion[i], vector_seleccion[minimo] =
vector_seleccion[minimo], vector_seleccion[i]
        #repetimos el proceso hasta terminar con todos

    #printeamos el vector ordenado por selección
    print("SELECCIÓN: Ordenado con Selección: ",vector_seleccion)
selectionsort(vector_seleccion)

print('-----------------------------------------------------------------
----------------------------------------')  #separador
```

Y por último, nos tocará ordenarlo por **INCREMENTO DECRECIENTE** o **SHELL**:

```
#SHELL O INCREMENTO DECRECIENTE

#hacemos una lista que almacene los numeros comprendidos entre el 1 y
el 11 (1,2,3,4,5,6,7,8,9,10)
lista_de_numeros_4 = list(range(1, 11)) # Creamos la lista base con
números del 1 al 10

#creamos una lista aleatoria con el método sample con los 10 elementos
aleatorios de la lista base
vector_shell = sample(lista_de_numeros_4,10)

#creamos una función para hacer los métodos
def shellsort(vector_shell):
    #printeamos los números sin ordenar según se han generado
    print("SHELL: El vector SIN ordenar es:", vector_shell)

    #contador del largo del vector
    largo = 0

    #bucle for _ in _ : para ir sumando al contador
    for i in vector_shell:
        largo += 1
    distancia = largo // 2

    #creamos un bucle según las distancias
    while distancia > 0:
        #bucle for _ in _ : con rango entre la distancia y el largo
        for i in range(distancia, largo):
            val = vector_shell[i]
```

```
            j = i
            while j >= distancia and vector_shell[j - distancia] > val:
                vector_shell[j] = vector_shell[j - distancia]
                j -= distancia #se escribe igual así: j = -distancia
            vector_shell[j] = val
        distancia //= 2 #acotamos la distancia nuevamente y continua el
ciclo hasta el final

    #printeamos el vector ordenado con shell o incremento decreciente
    print("SHELL: Ordenado con Shell: ", vector_shell)
shellsort(vector_shell)
```

Finalmente, habríamos acabado, sólo nos queda ver el resultado…

RESULTADO:

```
BUBBLESORT: Vector sin ordenar: [7, 5, 6, 2, 3, 8, 1, 9, 4, 10]
BUBBLESORT: Con ordenación BubbleSort: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
------------------------------------------------------------------------
INSERCION: El vector SIN ordenar es: [8, 10, 6, 9, 4, 7, 5, 3, 2, 1]
INSERCION: Ordenado con Inserción:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
------------------------------------------------------------------------
SELECCIÓN: El vector SIN ordenar es: [4, 7, 6, 2, 1, 9, 8, 5, 3, 10]
SELECCIÓN: Ordenado con Selección:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
------------------------------------------------------------------------
SHELL: El vector SIN ordenar es: [7, 2, 5, 3, 8, 9, 4, 6, 10, 1]
SHELL: Ordenado con Shell:  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

Qué es la búsqueda dicotómica o binaria

La búsqueda ***binaria*** es un algoritmo que permite encontrar un elemento en una lista ordenada de elementos de manera eficiente. La forma en la que el algoritmo trabaja es reduciendo a la mitad la parte de la lista en la que podríamos encontrar el elemento que buscamos, hasta llegar a la opción correcta. Por ejemplo, el catálogo estelar Tycho-2 contiene 2,539,913 estrellas. Si buscáramos una estrella concreta podríamos en el peor de los casos tener que examinar una a una todas las estrellas del catálogo. Sin embargo, si aplicamos el algoritmo de búsqueda **binaria**, solo tendríamos que examinar 22 opciones en el peor de los casos.

Este **algoritmo** funciona eligiendo el elemento del medio en una lista ordenada de elementos, comparándolo con el elemento que queremos encontrar (ver si es menor, igual o mayor) y creando dos subgrupos. Por ejemplo, en una lista del 1 al 9 elegimos el 3 como número a encontrar, y por ello usamos el 5 como separador, ya que es el **número del medio**, y creamos dos subgrupos de cuatro números cada uno. En el primero irán los cuatro números menores que 5, y en el segundo los cuatro números mayores que cinco. Como nuestro número es menor que 5, dividimos el primer subgrupo de nuevo. Si elegimos el 3 dentro de este subgrupo nuestra búsqueda habrá finalizado, ya que es el número que estábamos buscando.

Para saber cuántas operaciones tendremos que realizar para encontrar el elemento en el peor de los casos podemos realizar la operación matemática **log(2,N+1)**. También es aplicable de forma recursiva.

Vamos a realizar un programa en **Visual Studio Code** con el ejemplo anterior para demostrar cómo funciona la búsqueda binaria.

En el programa tenemos una lista con números del 1 al 9 y queremos encontrar la posición del número 7 dentro de esta. El código también contempla la posibilidad de que se pida un número fuera de la lista.

Hito.py > hito

```python
def hito(numero):
    numeros=[1,2,3,4,5,6,7,8,9] #Establecemos las variables que vamos a usar
    maximo=len(numeros)-1
    minimo=0
    medio=0

    while minimo <= maximo:
        medio=(maximo+minimo)//2 #El operador // nos devuelve el resultado de la división como un entero
        #La operación sirve para encontrar el elemento del medio de la lista
        if numeros[medio] < numero: #Si el elemento del medio es menor que nuestro número
            minimo= medio+1 #nuestro nuevo mínimo es el siguiente elemento de la lista

        elif numeros[medio] > numero: #Si el medio es mayor que nuestro número
            maximo= medio -1 #Nuestro nuevo máximo es el anterior elemento de la lista

        else:
            return medio #En el caso de que nuestro número y el medio coincidan
            #el programa devuelve "medio", que es el índice en el que se
            #encuentra nuestro número

    return -1 #Si el elemento no estaba en nuestra lista, devuelve -1

resultado= hito(7)
if resultado == -1:
    print('El número no esta en la lista')
else:
    print(f'El número se encuentra en el índice {resultado}')

```

Al ejecutar el programa correctamente nos indica que el número 7 está en la posición (índice) de la lista 6.

```
El número se encuentra en el índice 6
```

BIBLIOGRAFÍA / WEBGRAFÍA:

https://edukativos.com/apuntes/archives/10565

https://es.khanacademy.org/computing/computer-science/algorithms/binary-search/a/binary-search

https://www.javatpoint.com/binary-search-in-python

https://github.com/GBaudino/MetodosDeOrdenamiento

https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoDeShell.html#:~:text=El%20ordenamiento%20de%20Shell%2C%20a,mediante%20un%20ordenamiento%20por%20inserci%C3%B3n.

https://juncotic.com/ordenamiento-por-insercion-algoritmos-de-ordenamiento/#:~:text=El%20algoritmo%20de%20ordenamiento%20por%20inserci%C3%B3n%20es%20un%20algoritmo%20de,insert%C3%A1ndolo%20en%20el%20lugar%20correspondiente.

https://blog.pleets.org/article/algoritmo-de-ordenamiento-de-burbuja

http://lwh.free.fr/pages/algo/tri/tri_selection_es.html

http://lwh.free.fr/pages/algo/tri/tri_shell_es.html