1. Ordenamiento

Existen múltiples algoritmos con el propósito de ordenar los datos (*sorting*), usualmente dentro de un arreglo u otro tipo de *estructura de datos*. Es importante poder tener organizada la información, ya que esta es de vital importancia para la mayoría de las aplicaciones con las que se interactúan todos los días.

Existen muchos algoritmos para este propósito ya que algunos se ajustan mejor como solución en determinados escenarios. Con un fin pedagógico, aquí solo se mencionarán algunos de ellos y se desarrollará uno en particular denominado **ripple sort**.

1.1. Burbujeo (*bubble sort*)

Este método de ordenamiento se basa en recorrer el vector de datos comparando los elemento adyacentes entre sí, intercambiándolos si el elemento de la derecha es mayor o menor según el criterio que se desee emplear. Esto debe repetirse una y otra vez sobre el vector hasta dejarlo ordenado.

Puede encontrar una animación en el siguiente enlace:

`https://tute-avalos.com/images/bubblesort.gif`.

1.2. Por inserción (*insertion sort*)

A diferencia del burbujeo, en este algoritmo se recorre el vector y se toma el valor evaluado y se lo compara con cada uno de los elementos anteriores hasta *insertarlo en el lugar correcto*. Si bien, esta es una mecánica que sería la tendencia a aplicar para un ser humano, es bastante más compleja de programar.

Puede encontrar una animación en el siguiente enlace:

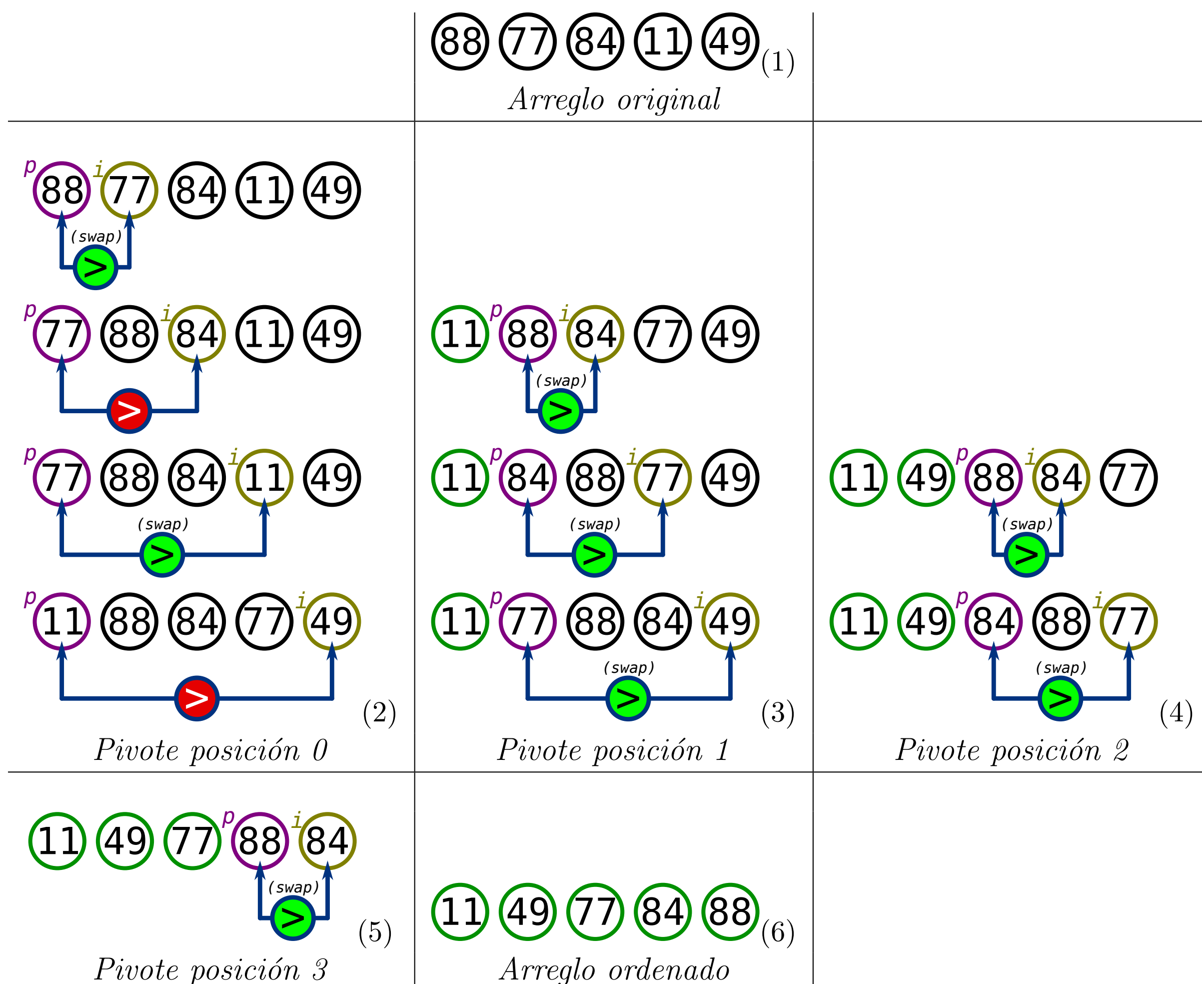
`https://tute-avalos.com/images/insertionsort.gif`.

1.3. Por “onda” (*ripple sort*)

Este es una variante de *bubble sort* para mejorar el tiempo de ejecución al evitar repetir constantemente y es relativamente sencillo de idear. Se basa en un sistema de *pívot*, donde se ordena una posición a la vez. Luego, ésta se va comparando con las posiciones siguientes hasta alcanzar el último elemento (`p+1, p+2, p+3, ..., p+(N-1)`), intercambiando los valores del vector (*swap*) si cumplen el criterio de ordenamiento (ascendente o descendente).

A continuación se expone un ejemplo gráfico de ordenamiento ascendente de un arreglo de 5 posiciones:

Matías S. Ávalos© CC BY NC SA 4.0



Como se observa, el pivote (indicado por el índice `p`) va avanzando e intercambiando elementos dejando el valor correspondiente en su posición. En cambio, el elemento a comparar (indicado por el índice `i`) empieza siempre en la posición siguiente al pivote (`p+1`), es por ello que `p` solo va hasta el anteúltimo elemento del arreglo, ya que el último quedará ordenado por descarte.

En la siguiente implementación de código, se generará un arreglo de N elementos aleatorios, mostrando el orden original en que fueron creados y luego se aplicará *ripple sort* **ascendente**, por último se mostrará el arreglo ordenando al final del programa:

Programa 1: Ejemplo práctico de *ripple sort*

```
1  import random
2
3  def ripple_sort(vector:list):
4    # El pívot se inicializa en 0 y se desplaza hasta el
5    # anteúltimo elemento.
6    for pivot in range(len(vector)-1):
7      # Luego se compara sucesivamente con los demás elementos,
8      # intercambiándolos cuando es necesario.
9      for i in range(pivot+1, len(vector)):
10       if vector[pivot] > vector[i]:
11         # nuevo valor de pivote:
12         vector[pivot], vector[i] = vector[i], vector[pivot]
13
```


Matías S. Ávalos© CC BY NC SA 4.0

```
14 def main():
15   CANT_ELEM = 10
16   valores = [int() for v in range(CANT_ELEM)]
17   # Se asignan valores alaeatorios entre 0 y 99 al vector
18   #  y se muestran en pantalla.
19   print("Vector original:")
20   for v in range(len(valores)):
21     valores[v] = random.randint(0,99)
22     print(f"[{valores[v]}]",end='')
23   print() # nueva línea
24 
25   # Se ordena el vector a través del algoritmo ripple sort
26   ripple_sort(valores)
27 
28   # Se muestra el arreglo ya ordenado:
29   print("\nVector ordenado: ")
30   for v in valores:
31     print(f"[{v}]",end='')
32   print() # nueva línea
33 
34 main()
```

El algoritmo de ordenamiento está comprendido entre las *líneas 3 y 12*, donde se observa que el pivote está inicializado en 0 y se desplaza consecutivamente hasta alcanzar el "último elemento -1", y que el índice con el cual se compararán los demás elementos con el pivote empieza siempre en `pivot+1` (*línea 9*) y se desplaza hasta el último elemento. El signo de la comparación (*línea 10*) es el que determinará si se ordena de forma ascendente o descendente. En este caso, si el pivote es mayor que el comparado en `pivot+i` se intercambian, de esta forma, *el menor valor es el nuevo pivote* (*línea 12*) y, al final, quedan ordenados **ascendentemente** (de menor a mayor).

Una posible salida al ejecutar este programa sería:

```
Vector original:
[44][39][2][94][62][14][0][75][13][16]

Vector ordenado:
[0][2][13][14][16][39][44][62][75][94]
```

Cuando se requiere ordenar vectores asociados, es decir, que ambos están relacionados por su índice, pero son datos en arreglos independientes, debe tener en cuenta que se aplica el criterio de ordenamiento en uno de ellos y luego se intercambian ambos datos. Analicemos el siguiente ejemplo donde se requieren 5 nombres junto a su promedios y luego se pide mostrarlos ordenados de mayor a menor según su promedio.

Programa 2: Ordenamiento de más de un arreglo a la vez

```
1 def ordenar_por_promedio(prom:list[float], nom:list[str]):
2   for p in range(len(prom)-1):
3     for i in range(p+1, len(prom)):
4       if prom[p] < prom[i]:
5         prom[p],prom[i] = prom[i],prom[p] # se ordenan promedios
6         nom[p],nom[i] = nom[i],nom[p]     # y nombres
```


Matías S. Ávalos© CC BY NC SA 4.0

```python

def pedir_texto(mensaje:str) -> str:
  while True:
    try:
      x = input(mensaje)
      if not x[0].isspace():
        return x
      else:
        print("Error de ingreso")
    except:
      print("Error de ingreso")

def pedir_numero(mensaje:str) -> float:
  while True:
    try:
      x = float(input(mensaje))
      return x
    except ValueError:
      print("Error de ingreso")

def pedir_numero_entre(mensaje:str,vmin:float,vmax:float)->float:
  if vmin > vmax:
    vmin, vmax = vmax, vmin
  while True:
    x = pedir_numero(mensaje)
    if x >= vmin and x <= vmax:
      return x
    else:
      print("Error de ingreso")

def main():
  CANT_ESTUDIANTES = 5
  nombres   = [str() for s in range(CANT_ESTUDIANTES)]
  promedios = [float() for f in range(CANT_ESTUDIANTES)]

  # Se solicita el ingreso de datos por el usuario
  for i in range(CANT_ESTUDIANTES):
    nombres[i] = pedir_texto(f"Ingrese nombre "
                             "del estudiante {i+1}: ")
    promedios[i] = pedir_numero_entre("Ingrese promedio:",1,10)

  # Ripple Sort descendente (mayor a menor) de promedios:
  ordenar_por_promedio(promedios, nombres)

  print("\nLos estudiantes ordenados por promedio:")
  for i in range(CANT_ESTUDIANTES):
    print(f"{i+1}. {nombres[i]} ({promedios[i]})")

main()
```

Si bien este tipo de ordenamientos *cruzados* ocurren, lo ideal es tener un solo arreglo

Matías S. Ávalos© CC BY-NC-SA 4.0

con los **datos estructurados** (registros), que es un tema que se abordará en la próxima unidad.

1.4. Algoritmos avanzados

1.4.1. Ordenamiento rápido (*quick sort*)

Este algoritmo de ordenamiento es uno de los más populares y estuvo impuesto por muchos años como la mejor solución genérica. Éste, al igual que muchos otros, está basado en la técnica de *divide y vencerás*. Se toma el último elemento como *pívot*, y se recorren los elementos comparando el primero y ante-último e intercambiándolos si el de la izquierda es mayor que el *pívot* y el de la derecha menor. Luego se traslada el *pívot* a la mitad del arreglo. Para este momento todos los mayores están a la derecha del pivot y los menores a la izquierda, pero desordenados, entonces se hace lo mismo con ambas mitades, y luego con la mitad de la mitad, etc.

A continuación se deja un link a una animación que clarifica la operación:

`https://tute-avalos.com/images/quicksort.gif`

1.4.2. Ordenamientos "mixtos"

Hoy en día los algoritmos que se utilizan para ordenar información de grandes o bajos volúmenes son, en realidad, una mezcla de varios algoritmos. En determinados casos, dependiendo de la longitud de los datos o la tecnología utilizada para almacenarlos se opta por uno u otro como parte del algoritmo en sí. La mayoría de los lenguajes ya los incluyen como herramientas de biblioteca, pero en este curso nos centramos en el pensamiento computacional y entender este tipo de algoritmos y no ser meros usuarios o *codificadores* de software. Aunque, cabe destacar, que en general es conveniente utilizar las herramientas provistas por los lenguajes, ya que han sido arduamente *testeadas* y evolucionan con las distribuciones nuevas de los mismos.

2. Búsqueda

Otro factor importante en un arreglo es la búsqueda, es decir, encontrar uno o varios datos que cumplan con un criterio dado. Para ello, existen varios algoritmos. En este curso expondremos únicamente 2 de ellos. Por un lado, la más fácil de implementar es la búsqueda lineal.

2.1. Lineal

Las búsquedas lineales son las que utilizaríamos naturalmente, recorriendo uno por uno los elementos hasta encontrar el o los que cumplan con el criterio deseado. Esto usualmente se logra con un bucle que recorre desde el primero (o segundo) elemento comparándolo con otro elemento que cumple con el criterio dado.

Matías S. Ávalos© CC BY NC SA 4.0

Supóngase que se quiere encontrar el índice del mejor elemento en un arreglo de números, entonces una función que logre esto se codificaría de la siguiente manera:

```
''' Busqueda lineal del indice con el mayor valor '''
def obterner_indice_del_mayor(vector):
  indiceDelMayor = 0
  for i in range(1, len(vector)):
    # Si el valor actual (en la posición i) tiene
    # un valor mayor al anteriormente detectado se
    # actualiza el indiceDelMayor con el actual.
    if vector[i] > vector[indiceDelMayor]:
      indiceDelMayor = i

  return indiceDelMayor
```

Como se observa el algoritmo anterior este es recorrido con un primer pívot en la posición 0 y comparándolo secuencialmente con los próximos elementos. Si se encontrara el que cumpliera con el criterio, este es reemplazado.

Por ejemplo, si tuviera arreglos desordenados de estudiantes y promedios y quisiera mostrar el estudiante de mejor promedio podría hacer:

```
mejor = obterner_indice_del_mayor(promedios)
print(f"Mejor promedio: {estudiantes[mejor]}")
```

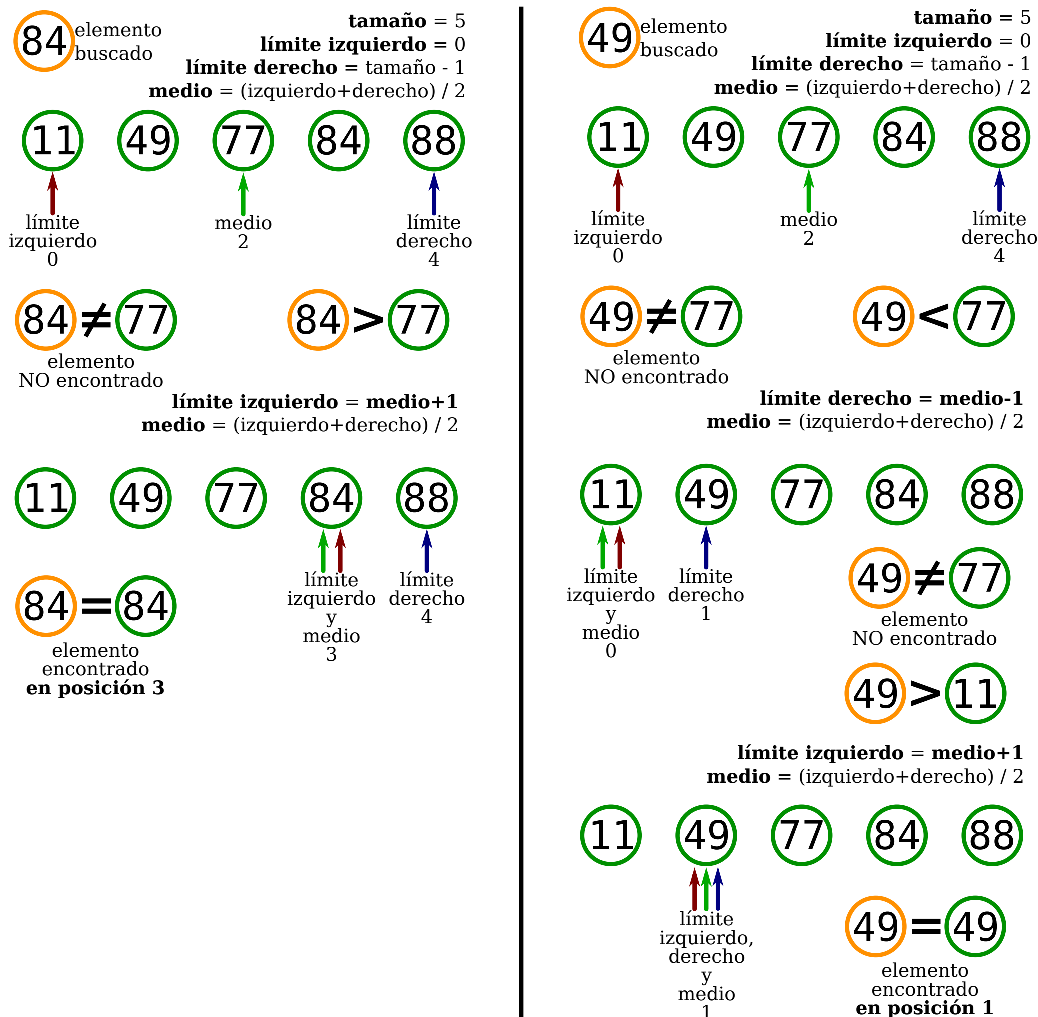
También podría *ahorrarse* la variable si se hiciera:

```
estudiantes[obterner_indice_del_mayor(promedios)]
```

2.2. Binaria

Este es un algoritmo muy sencillo, se basa en comparar el elemento del medio entre dos extremos. Primero se evalúa el elemento en el medio del arreglo, es decir entre el primer y último elemento, si este elemento es mayor al buscado entonces se busca entre el primero y el anterior al consultado, sino entre el siguiente y el último y así sucesivamente hasta encontrar el elemento deseado.

Matías S. Ávalos© CC BY NC SA 4.0



Es un pre-requisito para utilizar este algoritmo que el arreglo esté previamente ordenado. Sino, será imposible poder determinar donde se encuentro el elemento.

Ejemplo de búsqueda binaria

```
def busqueda_binaria(vector, buscado):
  izq = 0
  der = len(vector)-1
  # Mientras sean válidos los límites:
  while izq <= der:
    # se calcula el índice de la mitad
    med = int((izq + der) / 2)
    # Si se encontró el elemento buscado se devuelve el índice:
    if vector[med] == buscado:
      return med
    # Se calcula el nuevo límite:
    if buscado < vector[med]:
      der = med - 1
    else:
      izq = med + 1
  # Devolver un valor de índice no válido:
  return -1
```


Matías S. Ávalos© CC BY NC SA 4.0

Tener el vector ordenando es una ventaja a la hora de hacer búsquedas ya que para encontrar máximos y mínimos simplemente se obtiene el último o primer elemento respectivamente (sin necesidad de buscar).

3. Mezclar un arreglo

Desordenar un arreglo requiere de aplicar una lógica más compleja que al ordenar. Pero afortunadamente existe una función estándar dentro del módulo `random` que proporciona un algoritmo de mezcla llamada `shuffle()`.

Programa 3: Desordenar un arreglo con `random.shuffle()`

```
import random

def main():
  # vector ordenado
  vec = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

  # Desordenar el vector:
  random.shuffle(vec)

  # Se muestra el vector desordenado:
  for v in vec:
    print(f"[{v}]",end='')
  print()

main()
```

Como se observa simplemente debemos pasarle el vector como argumento a la función `random.shuffle()` para que este sea mezclado por el algoritmo.

Una posible salida de este programa se vería de la siguiente manera:

```
[6][4][7][1][0][5][8][3][9][2]
```

4. Ejercitación

Codifique los siguientes programas utilizando arreglos y funciones en `Python`.

1. Pida ingresar 10 nombres y muéstrelos ordenados alfabéticamente.
2. Pida ingresar 5 estudiantes por apellido y sus notas trimestrales (3 notas). Muestre los apellidos junto a su 3 notas ordenados por el promedio de los mismos.
3. Pida ingresar 10 fechas (día, mes y año) en formato numérico. Muestre las fechas ordenadas de la más actual a la más antigua en el formato `día/mes/año`.

Matías S. Ávalos© CC BY NC SA 4.0

4. Pida ingresar los nombres completos (un nombre de pila y primer apellido), guardados en vectores diferentes. Muéstrelos ordenados en el formato "Apellido, Nombre" sabiendo que si hay apellidos repetidos, debe utilizar el nombre como criterio de orden. P.e. `Ferrari, Alfredo` está antes que `Ferrari, Bruno`.

5. Desarrolle un software que pida ingresar 10 "nombres de usuario" cualquiera y asignarles un código de 4 dígitos diferentes entre sí. Luego se ingresa a un menú que permite:

   a) `Ingresar código` que pida ingresar un código y muestre su nombre o el texto `"Usuario no encontrado"`.

   b) `Listar` que muestra los usuarios ordenados por código.

   c) `Salir` que finaliza la ejecución del programa.

Matías S. Ávalos© CC BY NC SA 4.0