ORDENAMIENTO Y BÚSQUEDA EN PYTHON

UNPAZ

Introducción a la Programación

Agenda

Introducción

1. Definición Ordenación
2. Ordenación por intercambio
3. Ordenación por selección
4. Ordenación por burbuja
5. Otros métodos de ordenamiento.
6. Búsqueda en Listas: Búsqueda secuencial y binaria

Ordenamiento

Introducción

- Es una operación consistente en disponer un conjunto de estructuras de datos en algún determinado orden con respecto a uno de los campos de elementos del conjunto.

- Cuando los datos de estructura está almacenados en memoria se lo denomina ordenación interna

- Si los datos están almacenados en un archivo se los llama ordenación externa

Ordenamiento

Tipos de Ordenamiento

- **Directos (básicos)**

  –Selección
  –Burbuja
  –Inserción

- **Indirectos (avanzados)**

  –Shell
  –Ordenación Rápida
  –Ordenación por mezcla
  –Radixsort

Ordenamiento por Selección

Desarrollo

El algoritmo consiste en ordenar los valores del array de modo que el dato contenido en a[0] sea el valor más pequeño, el a[1] el siguiente, y así hasta la a[n-1].

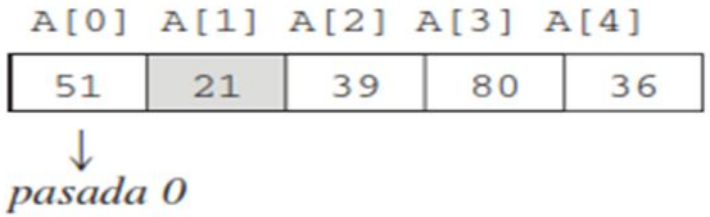
Pasos

1)Seleccionar el elemento más pequeño de la lista A. Intercambiarlo con el primer elemento A[0]. Ahora la entrada más pequeña está en la primera posición.

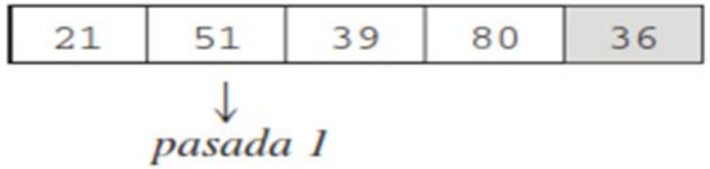
2)Considerar las posiciones de la lista a[1], a[2], a[3], seleccionar el elemento más pequeño e intercambiarlo con a[1]. Ahora las dos primeras entradas de A están en orden.

3)Continuar este proceso encontrado o seleccionado el elemento más pequeño de los restantes elementos de la lista, intercambiandolos adecuadamente

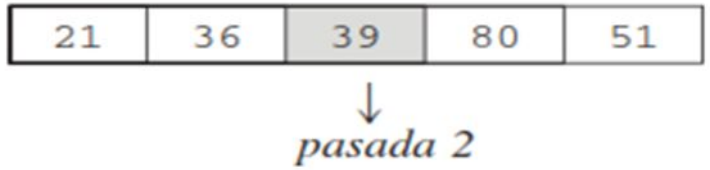
Ordenamiento por Selección



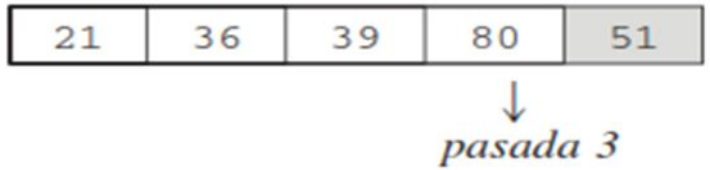
Pasada 0. Seleccionar 21
Intercambiar 21 y A[0]



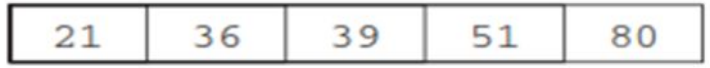
Pasada 1. Seleccionar 36
Intercambiar 36 y A[1]



Pasada 2. Seleccionar 39
Intercambiar 39 y A[2]



Pasada 3. Seleccionar 51
Intercambiar 51 y A[3]



Lista ordenada

Ordenamiento por Selección

```python
def seleccion(arreglo):
    longitud = len(arreglo)
    for i in range(longitud-1):
        min_idx = i
        for j in range(i+1, longitud):
            if arreglo[min_idx] > arreglo[j]:
                min_idx = j

        # Intercambiar
        if i != min_idx:
                aux = arreglo[i]
                arreglo[i] = arreglo[min_idx]
                arreglo[min_idx] = aux
```

Ordenamiento por Burbujeo

Desarrollo

La técnica utilizada se denomina ordenación por burbuja u ordenación por hundimiento debido a que los valores más pequeños «burbujean» gradualmente (suben) hacia la cima o parte superior del array de modo similar a como suben las burbujas en el agua, mientras que los valores mayores se hunden en la parte inferior del array. La técnica consiste en hacer varias pasadas a través del array. En cada pasada, se comparan parejas sucesivas de elementos. Si una pareja está en orden creciente (o los valores son idénticos), se dejan los valores como están. Si una pareja está en orden decreciente, sus valores se intercambian en el array.

En síntesis lo que se hace es comparar pares de elementos adyacentes

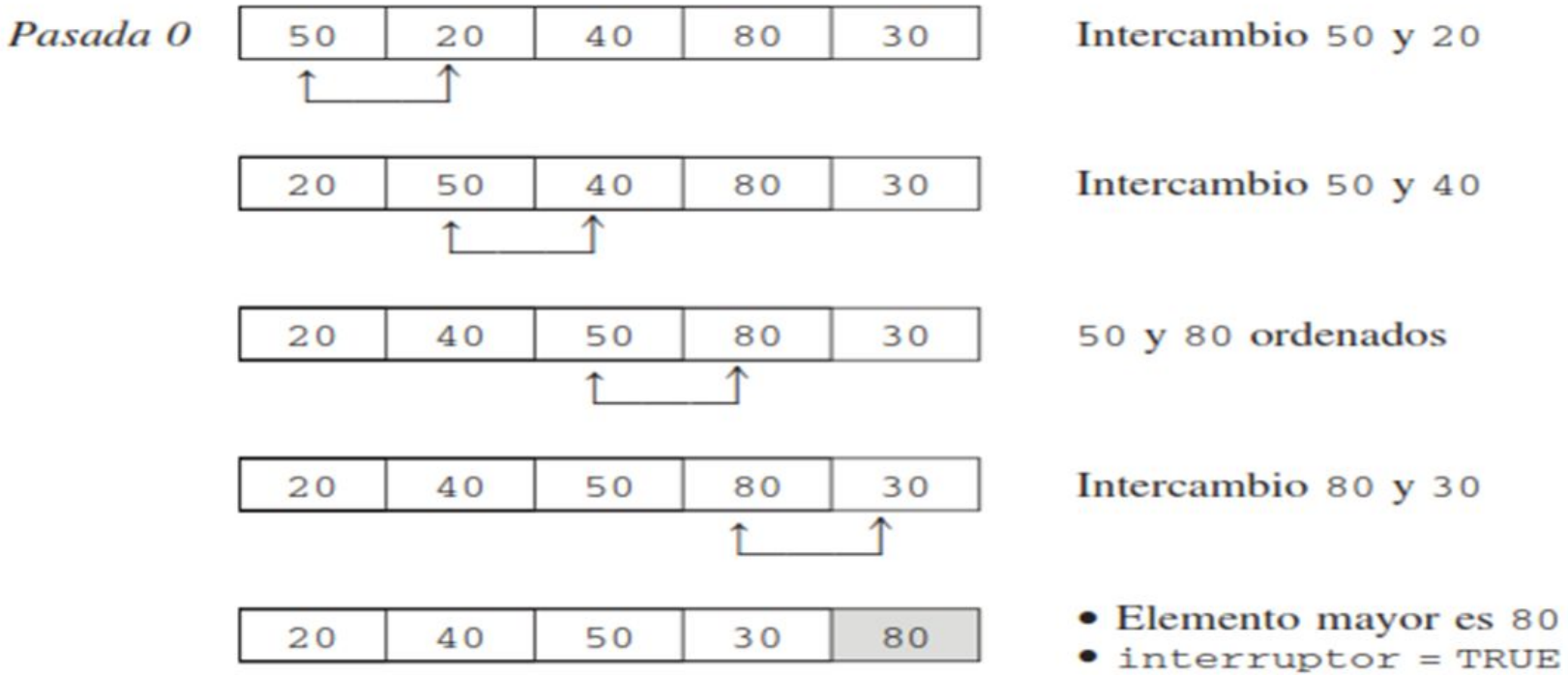
- Si están ordenados, me corro a la derecha
- Si no están ordenados, los intercambio

En cada iteración, se compara un elemento menos (el último)

Ordenamiento por Burbujeo

Pasos

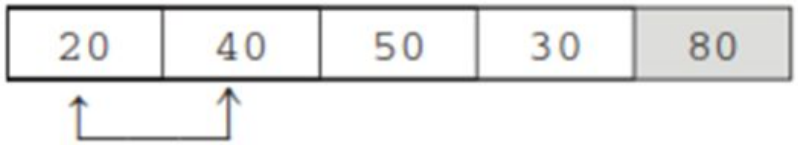
1) En la pasada 0 se comparan elementos adyacentes:
(A[0],A[1]),(A[1],A[2]),(A[2],A[3]),...(A[n-2],A[n-1])
Se realizan n-1 comparaciones, por cada pareja (A[i],A[i+1]) se intercambian los valores si A[i+1] < A[i]. Al final de la pasada, el elemento mayor de la lista está situado en A[n-1].



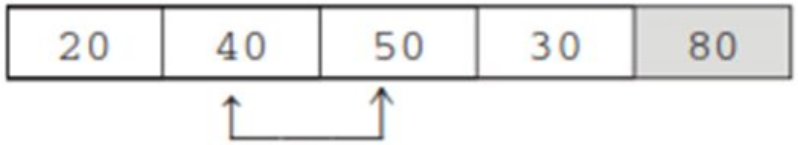
Ordenamiento por Burbujeo

Pasos

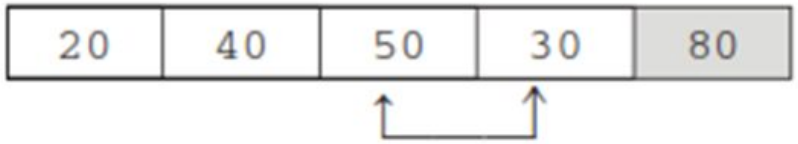
En la pasada 1:



20 y 40 ordenados



40 y 50 ordenados



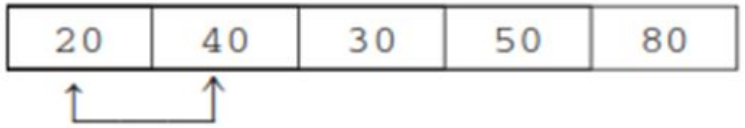
Se intercambian 50 y 30

- 50 y 80 elementos mayores y ordenados
- `interruptor = TRUE`

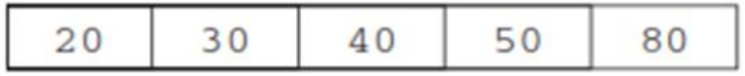
Ordenamiento por Burbujeo

Pasos

En la pasada 2, sólo se hacen dos comparaciones:

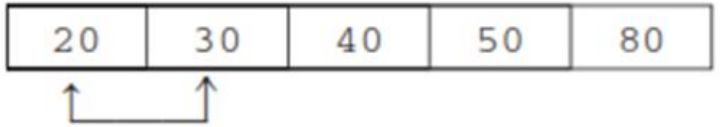


20 y 40 ordenados



- Se intercambian 40 y 30
- `interruptor = TRUE`

En la pasada 3, se hace una única comparación de 20 y 30, y no se produce intercambio:



20 y 30 ordenados

- Lista ordenada
- `interruptor = FALSE`

Ordenamiento por Burbujeo

Pasos

```python
def udfBubbleSort(arr):
    n = len(arr)
    # optimizar el código, por lo que si la matriz ya está ordenada, no es necesario
    # para pasar por todo el proceso
    hayCambios = False

    # Recorrer todos los elementos del arreglo o lista
    for i in range(n - 1):
        # range(n) también funciona, pero el bucle externo lo hará
        # repetir una vez más de lo necesario.
        # Los últimos elementos i ya están en su lugar
        for j in range(0, n - i - 1):

            # recorrer el arreglo de 0 a n-i-1
            # Cambiar si el elemento encontrado es mayor
            # que el siguiente elemento

            if arr[j] > arr[j + 1]:
                hayCambios = True
                #arr[j], arr[j + 1] = arr[j + 1], arr[j]
                aux = arr[j]
                arr[j] = arr[j + 1];
                arr[j + 1] = aux;

        if not hayCambios:
            # si no hemos necesitado hacer un solo intercambio,
            # puede simplemente salir del bucle principal.
            return
```

Ordenamiento por mezcla (mergesort)

Buscando mayor eficiencia

Volvamos una vez más a la técnica de divide y vencerás para resolver el problema de ordenar una lista de números enteros.

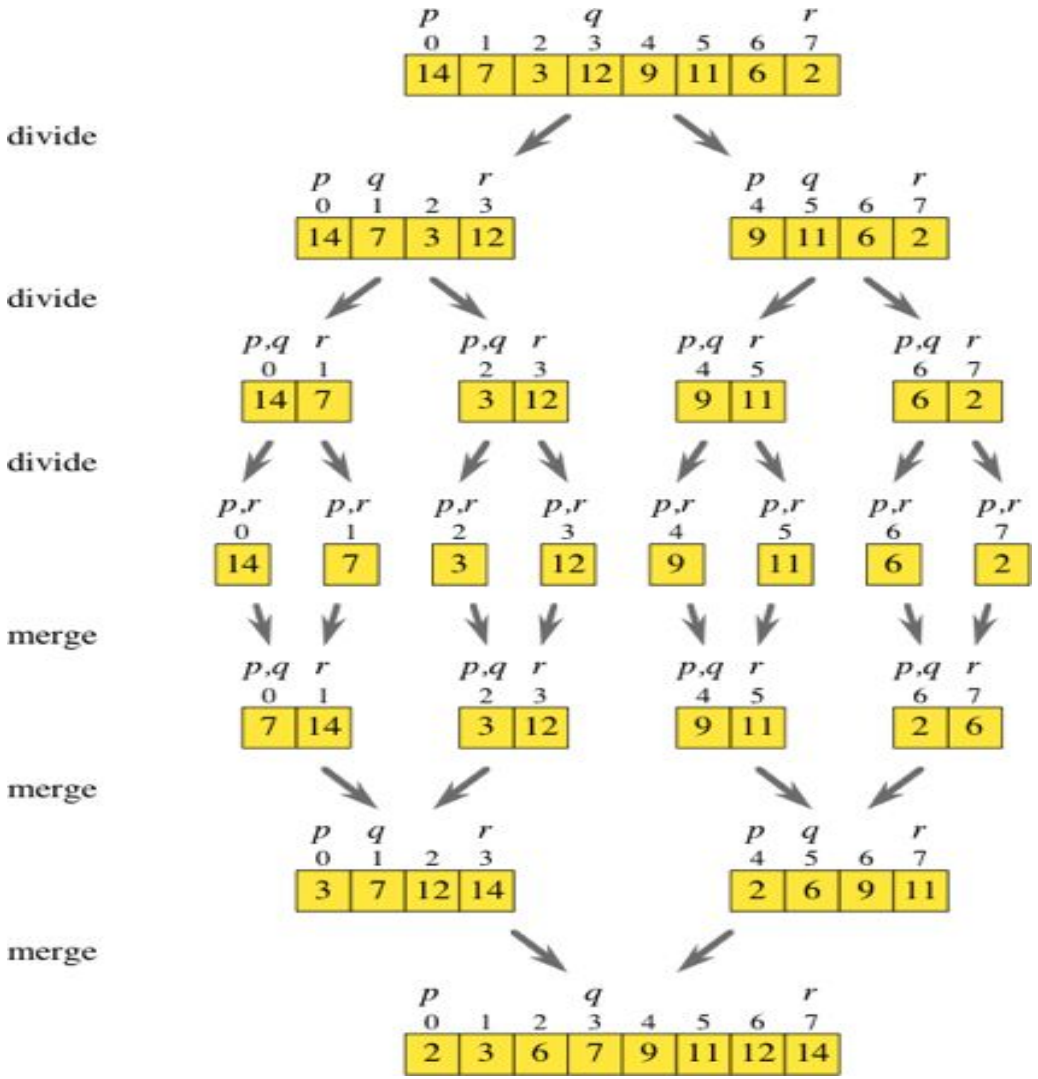
Vamos a crear una función a la que llamaremos merge_sort(), que toma como parámetros:

- La lista de números;
- Una lista temporal con el tamaño predeterminado de la lista de números;
- El índice inicial;
- El índice final de la lista.

Y se realizarán los siguientes pasos:

1. Dividiremos la lista en dos y llamaremos recursivamente a la función merge_sort(), pasando como parámetros:
2. Los datos de la primera lista;
3. Los datos de la segunda lista,
4. Finalmente, llamaremos a una función merge() para fusionar las dos listas ordenadas con el apoyo de la lista temporal, reescribiendo la lista original.

Ordenamiento por mezcla (mergesort)



Ordenamiento por mezcla (mergesort)

```
def merge_sort(lista, lista_temporaria, inicio, fin):
    if inicio < fin:
        medio = (inicio + fin) // 2
        merge_sort(lista, lista_temporaria, inicio, medio)
        merge_sort(lista, lista_temporaria, medio + 1, fin)
        merge(lista, lista_temporaria, inicio, medio + 1, fin)
```

Ordenamiento por mezcla (mergesort)

```python
def merge(lista, lista_temporaria, inicio, medio, fin):
    fin_primera_parte = medio - 1
    indice_temporario = inicio
    tamano_de_lista = fin - inicio + 1
    while inicio <= fin_primera_parte and medio <= fin:
        if lista[inicio] <= lista[medio]:
            lista_temporaria[indice_temporario] = lista[inicio]
            inicio += 1
        else:
            lista_temporaria[indice_temporario] = lista[medio]
            medio += 1
        indice_temporario += 1
    while inicio <= fin_primera_parte:
        lista_temporaria[indice_temporario] = lista[inicio]
        indice_temporario += 1
        inicio += 1
    while medio <= fin:
        lista_temporaria[indice_temporario] = lista[medio]
        indice_temporario += 1
        medio += 1
    for i in range(0, tamano_de_lista):
        lista[fin] = lista_temporaria[fin]
        fin -= 1
```

Ordenamiento por mezcla (mergesort)

```
def main():
    lista_de_numeros = [14, 7, 3, 12, 9, 11, 6, 2]
    tamano_de_lista = len(lista_de_numeros)
    lista_temporaria = [0] * tamano_de_lista
    merge_sort(lista_de_numeros, lista_temporaria, 0, tamano_de_lista - 1)
    for nombre in lista_de_numeros:
        print(nombre)


main()
```

Ordenamiento rápido (quicksort)

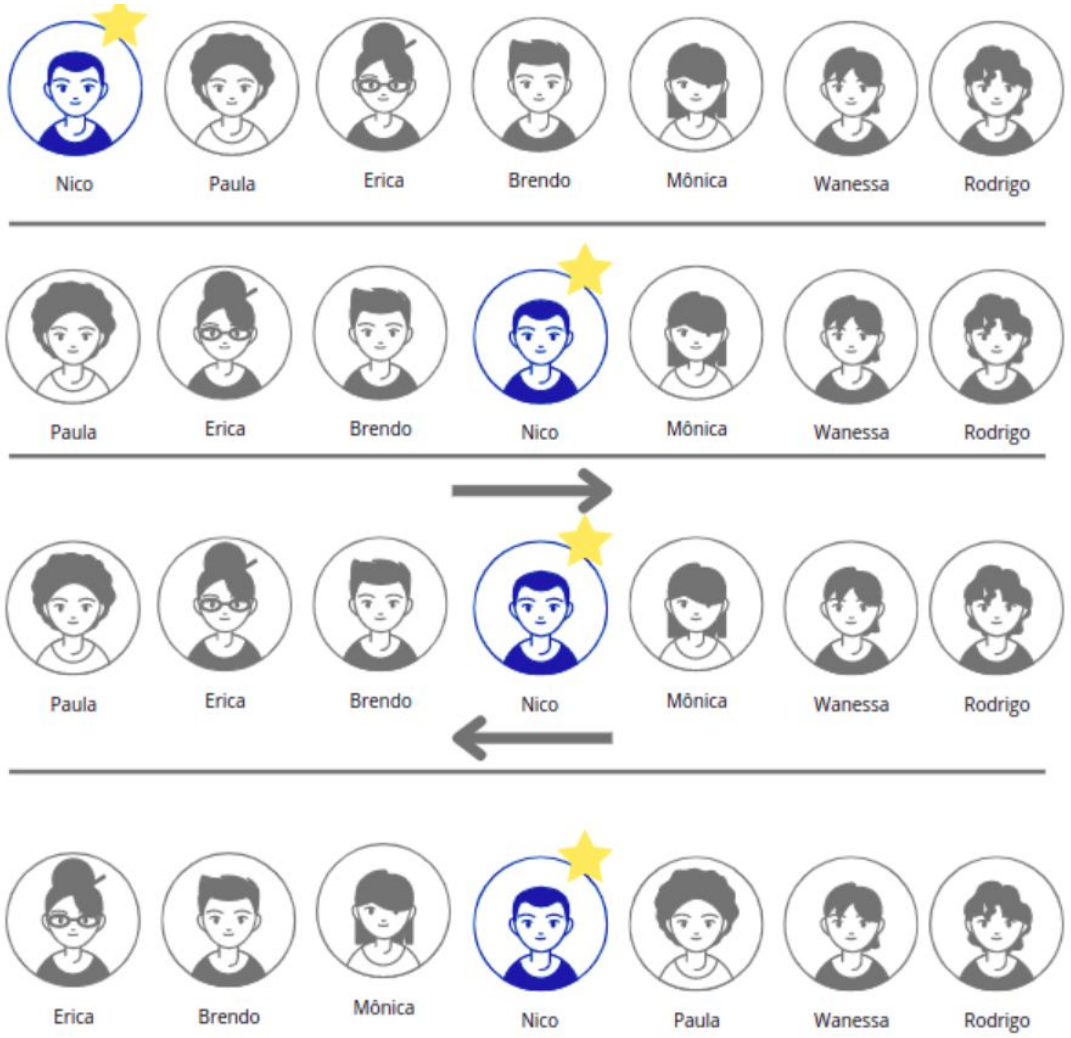
Buscando mayor eficiencia

Continuamos con la técnica de divide y vencerás para resolver el problema de ordenar una lista pero esta vez vamos a trabajar sobre una lista de nombres(str) para ordenarlos alfabéticamente.No dividiremos a la mitad sino que lo haremos a partir de un pivote.

Seguiremos los siguientes pasos para alcanzar nuestro objetivo:

1. Elegiremos un pivote (en este caso, el primer elemento de la lista) y cambiaremos su posición con el elemento en el medio
2. Repasemos toda la lista y verifiquemos elemento por elemento, comparándolos con el pivote. A partir de entonces: Si el ítem está en una posición más baja que el pivote en orden alfabético, será transferido o mantenido en la lista de la izquierda
3. Si el elemento está en una posición superior al pivote en orden alfabético, se transferirá o se mantendrá en la lista de la derecha. Haciendo esto recursivamente, al final tendremos una lista ordenada

Ordenamiento rápido (quicksort)



Ordenamiento rápido (quicksort)

```python
def quick_sort(lista, inicio, fin):
    if inicio > fin:
        return
    anterior = inicio
    posterior = fin
    pivot = lista[inicio]
    while anterior < posterior:
        while anterior < posterior and lista[posterior] > pivot:
            posterior = posterior - 1
        if anterior < posterior:
            lista[anterior] = lista[posterior]
            anterior = anterior + 1
        while anterior < posterior and lista[anterior] <= pivot:
            anterior = anterior + 1
        if anterior < posterior:
            lista[posterior] = lista[anterior]
            posterior = posterior - 1
        lista[anterior] = pivot
    quick_sort(lista, inicio, anterior - 1)
    quick_sort(lista, anterior + 1, fin)
```

Búsqueda Secuencial

Cuando los ítems de datos se almacenan en una colección, por ejemplo en una lista, decimos que tienen una relación lineal o secuencial. Cada ítem de datos se almacena en una posición relativa a los demás. En las listas de Python, estas posiciones relativas son los valores de los índices de los ítems individuales. Dado que estos valores de los índices están ordenados, es posible para nosotros visitarlos en secuencia. Este proceso da lugar a nuestra primera técnica de búsqueda, la búsqueda secuencial.

La Figura muestra cómo funciona esta búsqueda. Comenzando en el primer ítem de la lista, simplemente nos trasladamos de un ítem a otro, siguiendo el orden secuencial subyacente hasta que encontremos lo que buscamos o nos quedemos sin ítems. Si nos quedamos sin ítems, hemos descubierto que el ítem que estábamos buscando no estaba presente.

Figura 1: Búsqueda secuencial en una lista de enteros

Búsquedas en Listas y Arreglos

El proceso de encontrar un elemento específico de un array se denomina búsqueda. En esta sección se examinarán dos técnicas de búsqueda:

- Búsqueda lineal o secuencial, la técnica más sencilla
- Búsqueda binaria o dicotómica, la técnica más eficiente.

Búsqueda Secuencial

```
def busqueda_secuencial(unaLista, item):
    pos = 0
    encontrado = False
    while pos < len(unaLista) and not encontrado:
        if unaLista[pos] == item:
            encontrado = True
        else:
            pos = pos + 1
    return encontrado


listaPrueba = [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23]
print(busqueda_secuencial(listaPrueba, 3))
print(busqueda_secuencial(listaPrueba, 21))
```

Búsqueda Binaria

¿Podemos hacer algo mejor? Asumamos que la lista está ordenada y vamos a hacer algo distinto: nuestro espacio de búsqueda se irá achicando a segmentos cada vez menores de la lista original. La idea es descartar segmentos de la lista donde el valor seguro no está:

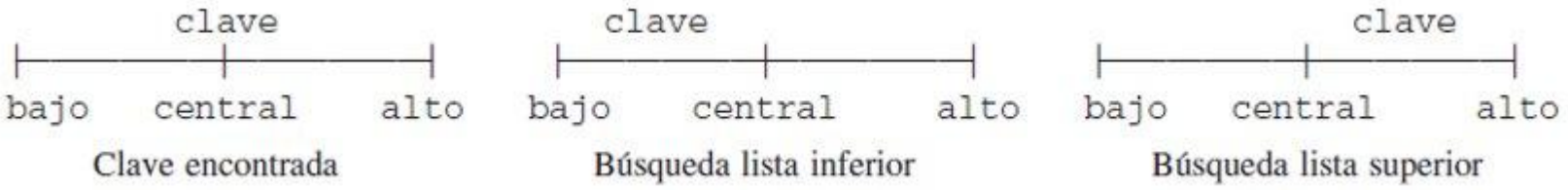
Consideramos como segmento inicial de búsqueda a la lista completa.
Analizamos el punto medio del segmento (el valor central), si es el valor buscado, devolvemos el índice del punto medio.
Si el valor central es mayor al buscado, podemos descartar el segmento que está desde el punto medio hacia la derecha.
Si el valor central es menor al buscado, podemos descartar el segmento que está desde el punto medio hacia la izquierda.
Una vez descartado el segmento que no nos interesa, volvemos a analizar el segmento restante, de la misma forma.

Búsqueda Binaria

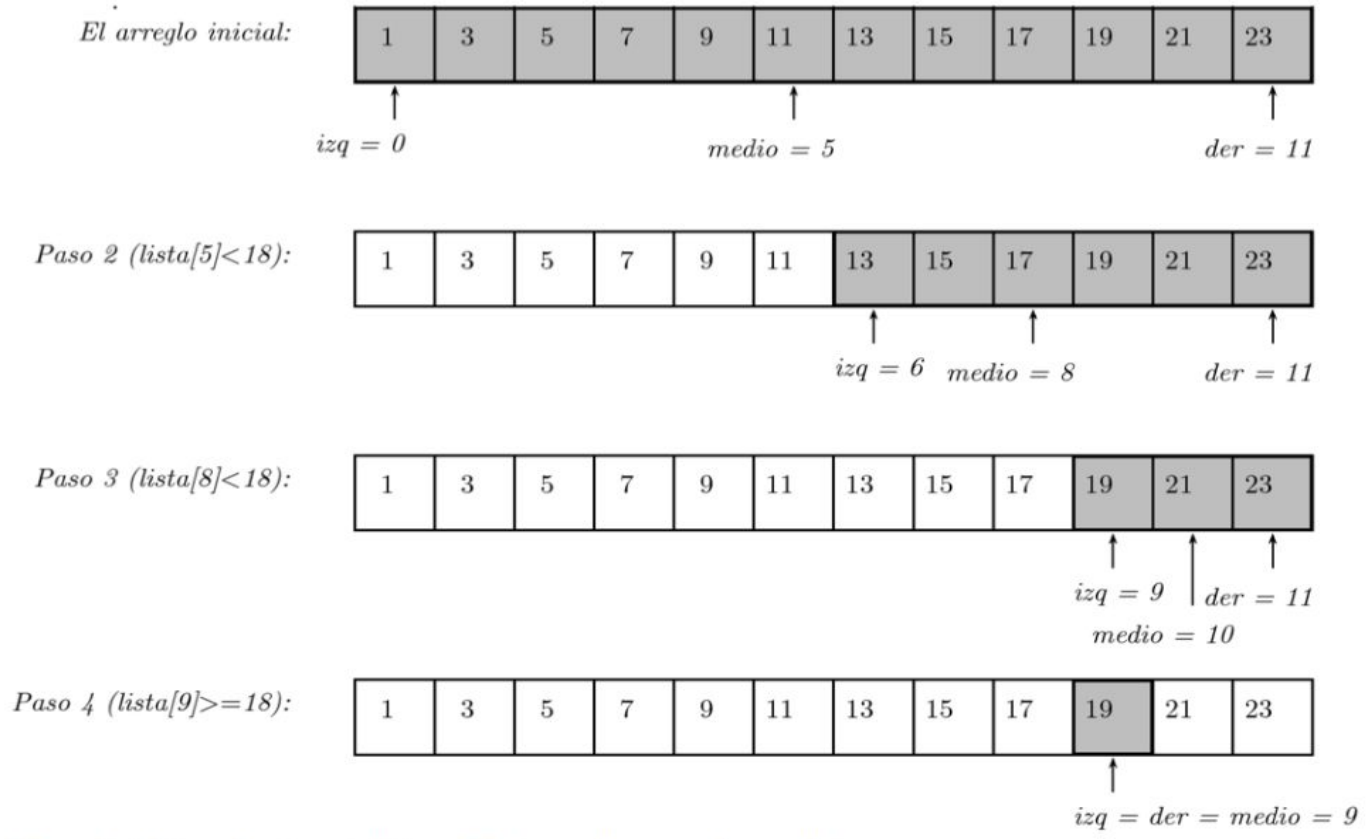
Si en algún momento el segmento a analizar tiene longitud 0 o negativa significa que el valor buscado no se encuentra en la lista.

Para señalar la porción del segmento que se está analizando a cada paso, utilizaremos dos variables (izq y der) que contienen la posición de inicio y la posición de fin del segmento que se está considerando. De la misma manera usaremos la variable medio para contener la posición del punto medio del segmento.

En el gráfico que se incluye a continuación, vemos qué pasa cuando se busca el valor 18 en la lista [1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23].



Búsqueda Binaria



Como no se encontró al valor buscado, devuelve -1

Búsqueda Binaria

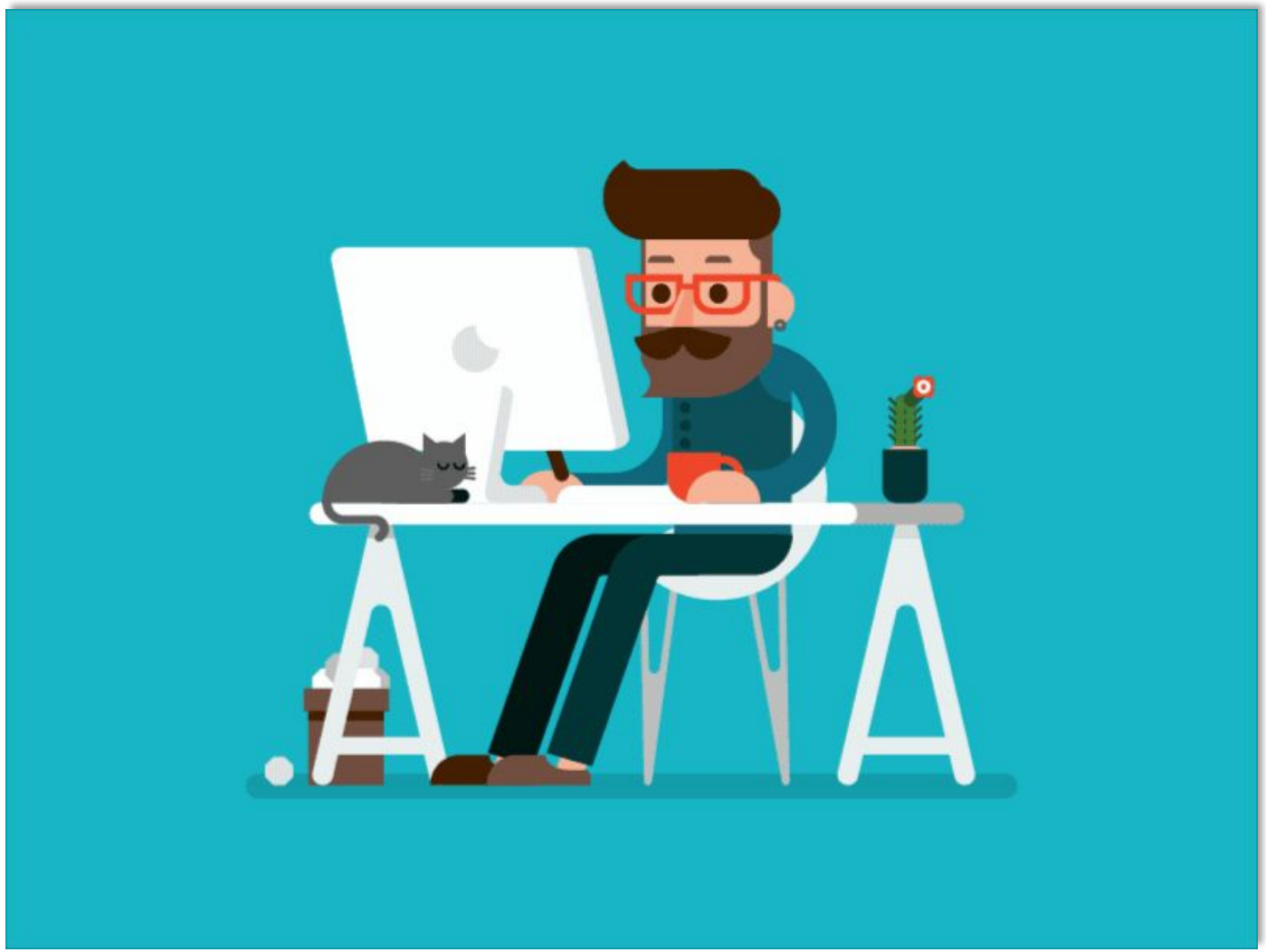
```python
def busqueda_binaria(lista, x):
    """
    Precondición: lista está ordenada
    Devuelve -1 si x no está en lista;
    Devuelve p tal que lista[p] == x, si x está en lista
    """
    izq = 0 # izq guarda el índice inicio del segmento
    der = len(lista) -1 # der guarda el índice fin del segmento
    # un segmento es vacío cuando izq > der:
    while izq <= der:
        medio = (izq+der)/2 # el punto medio del segmento
        if lista[medio] == x: # si el medio es igual al valor buscado, lo devuelve
            return medio
        elif lista[medio] > x:
            der = medio-1
        else:
            izq = medio+1

    return -1
```

Búsqueda Binaria

Números de elementos examinados		
Tamaño de la lista	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1.000	11	1.000
5.000	14	5.000
100.000	18	100.000
1.000.000	21	1.000.000

Gracias



Fuente: https://www.programiz.com/python-programming/