UNIVERSIDAD TECNOLÓGICA NACIONAL

Algoritmos de Búsqueda y Ordenamiento

Alumnos:
Matias Carro - matiasmanuelcarro@gmail.com
Hugo Catalan - hugocatalan2@gmail.com

Materia: Programación I
Profesor: Ariel Enferrel
Fecha de Entrega: 09/06/25

Link al repositorio del Proyecto:
https://github.com/Hugocatalan/busqueda_ordenamiento

Link al video del proyecto:
https://www.youtube.com/watch?v=3GZ-ZmUZhr4

Índice

Introducción

Este trabajo práctico se centra en los algoritmos de búsqueda y ordenamiento que son herramientas fundamentales en la programación y manejo de datos, permiten la búsqueda y el organizamiento rápido de los datos.

La elección de este tema se debe al uso extenso de los mismos en el campo de la programación, desde manejar datos hasta la optimización de procesos en los programas.
Otro motivo importante para aprender en profundidad cómo funcionan estos algoritmos es poder ir más allá de utilizar alguna función de “sort” o búsqueda que tengamos disponible en nuestro lenguaje de elección. Al entender cómo funciona por debajo de estas funciones un código, se mejora la capacidad de comprender, pensar y desarrollar nuevo código y funciones para los programas.

En este trabajo se busca comprender cómo funcionan los algoritmos de búsqueda y ordenamiento, explicandolos paso a paso. Se eligió como teoría la demostración de pasos para organizar una lista mediante los siguientes algoritmos:

Búsqueda:

- Búsqueda lineal
- Búsqueda Binaria

Ordenamiento

- Bubble Sort
- Insertion Sort
- Selection Sort
- Quick Sort
- Merge Sort

Centrándose en un ejemplo práctico donde se muestra el funcionamiento de Busqueda binaria y Bubble sorting en un programa en Python.

Marco teórico

Historia de la búsqueda ordenada y la búsqueda binaria:

La idea de ordenar datos para encontrarlos más fácilmente no es nueva. Ya en la antigua Babilonia, alrededor del año 200 a.C., se usaban tablillas como la de Inakibit-Anu, que contenía unos 500 números y sus recíprocos ordenados de forma que facilitaba su búsqueda. También se han encontrado listas de nombres ordenadas alfabéticamente en las islas del Egeo.

Más adelante, en 1286, se completó el *Catholicon*, un diccionario de latín que fue el primero en explicar cómo ordenar palabras alfabéticamente, no solo por la primera letra.

En tiempos modernos, la búsqueda binaria fue mencionada por primera vez en 1946 por John Mauchly durante las Conferencias de la Escuela Moore, un curso clave en los inicios de la computación. Luego, en 1957, William Wesley Peterson presentó un nuevo método llamado búsqueda por interpolación.

Durante los años siguientes, se fueron mejorando los algoritmos. Por ejemplo:

- En 1960, Derrick Henry Lehmer desarrolló una versión de búsqueda binaria que funcionaba con cualquier tamaño de lista.
- En 1962, Hermann Bottenbruch propuso una forma de hacer la búsqueda binaria en el lenguaje ALGOL 60, optimizando el número de comparaciones.
- En 1971, A. K. Chandra de la Universidad de Stanford desarrolló la búsqueda binaria uniforme.
- En 1986, Bernard Chazelle y Leonidas J. Guibas introdujeron un método llamado cascada fraccionaria, útil para resolver problemas complejos de búsqueda en geometría computacional.

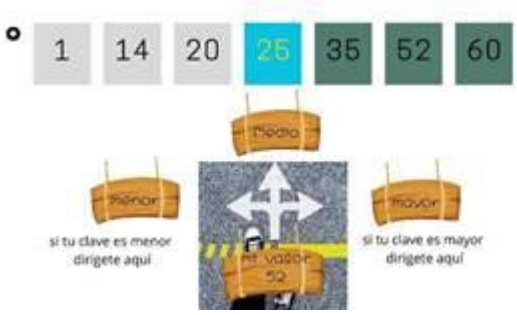
Teoría:

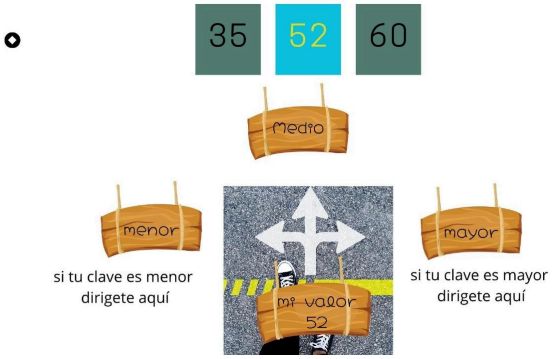
Algoritmo de búsqueda binaria: implementación iterativa y recursiva

El algoritmo de búsqueda binaria se utiliza en una matriz ordenada dividiendo **repetidamente** el intervalo de búsqueda por la mitad. La idea de la búsqueda binaria es utilizar la información de que la matriz está ordenada y reducir la complejidad temporal a O(log N).

Algoritmo de búsqueda binaria

- Divida el espacio de búsqueda en dos mitades **encontrando el índice medio**
- Compare el elemento central del espacio de búsqueda con la **clave**.
- Si la **clave** se encuentra en el elemento central, el proceso finaliza.
- Si la **clave** no se encuentra en el elemento central, elija qué mitad se utilizará como el próximo espacio de búsqueda.
    - Si la **clave** es más pequeña que el elemento del medio, se utiliza el lado **izquierdo para la siguiente búsqueda.**
    - Si la **clave** es más grande que el elemento del medio, se utiliza el lado **derecho para la siguiente búsqueda.**
- Este proceso continúa hasta que se encuentra la **clave** o se agota todo el espacio de búsqueda.





Búsqueda binaria versus otro esquema:

Usar arreglos ordenados con búsqueda binaria no siempre es la mejor opción cuando necesitamos agregar o quitar elementos con frecuencia. Esto se debe a que esas operaciones pueden tardar bastante, ya que muchas veces hay que mover varios elementos para mantener el orden.

Además, mantener estos arreglos ordenados puede hacer que el uso de la memoria sea menos eficiente, sobre todo si estamos insertando datos todo el tiempo.

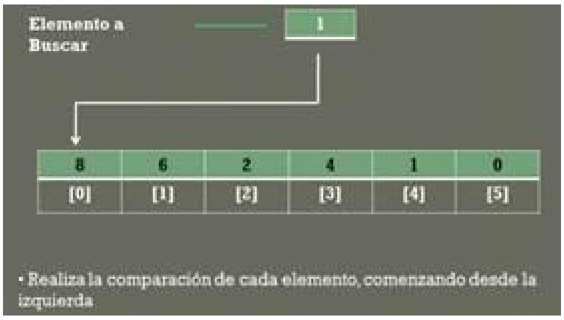
Existen otras formas de organizar la información que permiten agregar y eliminar elementos mucho más rápido. Aun así, la búsqueda binaria sigue siendo útil cuando queremos saber si un valor está presente o no, y lo hace bastante rápido.

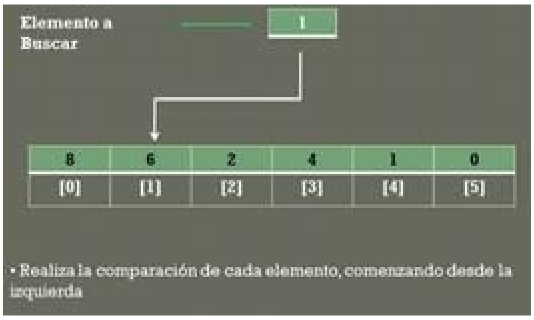
Una de sus ventajas más interesantes es que también puede ayudarnos a encontrar valores "aproximados", no solo exactos, y lo hace en muy poco tiempo. Además, si queremos encontrar el valor más pequeño o el más grande en un conjunto de datos ordenado, la búsqueda binaria también puede hacerlo de forma muy eficiente.

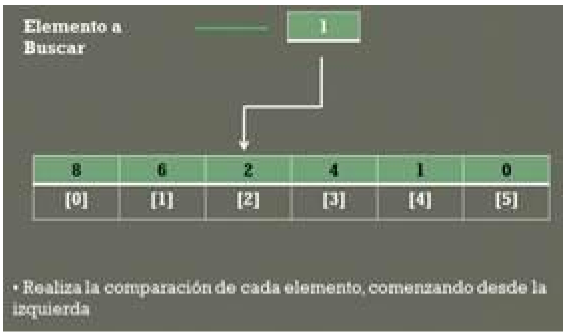
Búsqueda Lineal:

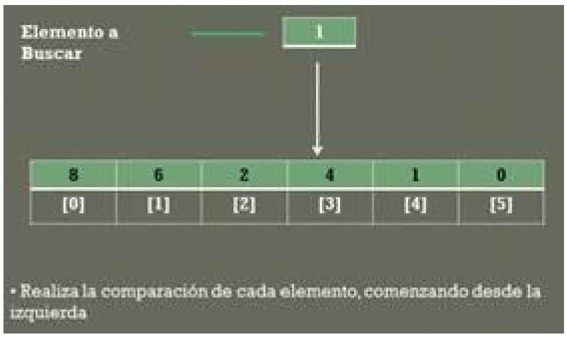
Cuando se trata de algoritmos de búsqueda, la búsqueda lineal es uno de los más sencillos. Si alguna vez has mirado una lista de elementos uno por uno hasta encontrar lo que buscabas, ¡entonces ya has hecho una búsqueda lineal!

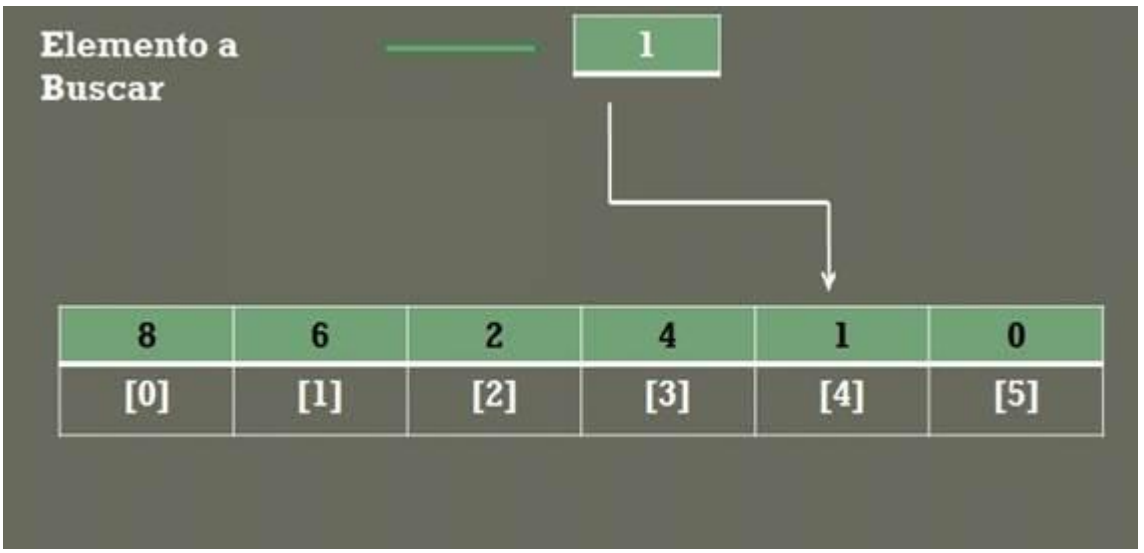
Además de ser fácil de entender, la búsqueda lineal es útil porque funciona con datos no ordenados, a diferencia de otros algoritmos de búsqueda. Esta versatilidad la convierte en una opción útil en situaciones en las que no es posible ordenar los datos.











Comparativa:

La **búsqueda lineal** es una forma sencilla de encontrar un valor en una lista: simplemente se revisa uno por uno hasta dar con el que se busca. Este método se puede usar en listas enlazadas, que permiten agregar o quitar elementos más fácilmente que los arreglos tradicionales.

Por otro lado, la **búsqueda binaria** suele ser mucho más rápida, pero solo funciona si los datos ya están ordenados. En listas cortas, la búsqueda lineal puede ser igual de efectiva o incluso más rápida, pero a medida que la lista crece, la búsqueda binaria se vuelve mucho más eficiente.

Eso sí, ordenar los datos antes de usar búsqueda binaria lleva tiempo. Los métodos más comunes para ordenar, como el ordenamiento rápido o el de fusión, necesitan hacer muchas comparaciones, especialmente cuando hay muchos elementos.

Una ventaja importante de la búsqueda binaria es que también puede ayudar a encontrar valores cercanos al que buscamos, no solo coincidencias exactas. Además, si queremos encontrar el valor más pequeño o el más grande en una lista ordenada, podemos hacerlo muy rápido, algo que no es tan fácil si la lista no está ordenada.

Notación O

¿Qué es la notación O?

Es una manera de representar la escalabilidad y eficiencia de un algoritmo , dando una aproximación de cómo el requerimiento de tiempo y número de operaciones para resolver del algoritmo crece con el tamaño de los datos que se le ingresan.

La notación O describe el peor caso posible de un algoritmo, es decir, el escenario en el que tarda más tiempo en ejecutarse.

¿Por qué usar notación O y no tiempo?

- La notación O es una forma de medir cuánto trabajo hace un algoritmo cuanto más datos le ingresas
- Describe el peor caso posible de complejidad de un algoritmo. Un algoritmo nunca va a tardar mas de O(n) de tiempo en ejecutarse, sin importar cuanto sea n
- Lo que nos importa es cómo crece el número de operaciones cuando aumentamos la cantidad de datos.
- Además, los tiempos individuales que le lleva a un algoritmo resolver, no puede predecir otros casos futuros

La notación O se puede expresar de las siguientes maneras: :

- **Peor caso**: El tiempo máximo que puede tardar el algoritmo. Es el más usado en notación O.
- **Mejor caso**: Mínimo tiempo posible.
- **Caso promedio**: Tiempo promedio esperado de ejecución.

La notación en general O se asocia con el **peor caso** para garantizar un rendimiento mínimo garantizado.

Es una herramienta esencial en el análisis de algoritmos, ya que permite evaluar la escalabilidad de las soluciones de forma abstracta y matemática. Entenderla es clave para diseñar sistemas eficientes, especialmente cuando se manejan grandes volúmenes de datos o se busca optimizar el rendimiento.

Tipos de complejidad con Notación O:

Notación	Nombre	Descripción breve	Ejemplo típico
O(1)	Constante	El tiempo no cambia con el tamaño de entrada	Acceso a un elemento de un array o lista
O(log n)	Logarítmica	Crece lentamente; divide el problema en partes	Búsqueda binaria
O(n)	Lineal	Escala directamente con el tamaño de entrada	Recorrido de una lista, o búsqueda lineal
O(n log n)	Lineal logarítmica	Eficiente, pero no óptimo	Mergesort, Quicksort
O(n^2)	Cuadrática	Tiempo crece rápidamente; ineficiente en general	Bubble sort, Insertion sort

Algoritmos de ordenamiento

Los algoritmos de ordenamiento son una serie de instrucciones que organizan una lista o array de una manera determinada.

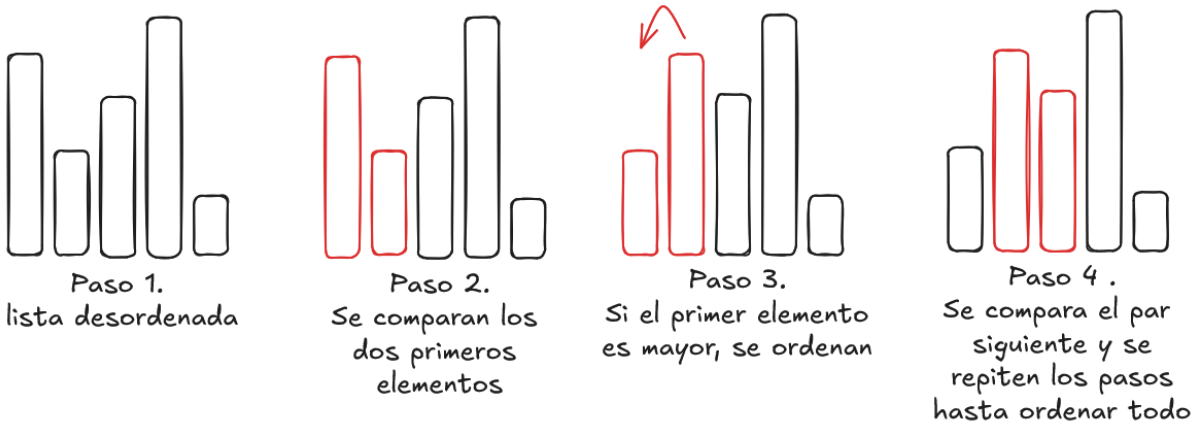
Los algoritmos de ordenamiento son importantes en la computación porque reducen la complejidad de un problema. Existe una variedad muy amplia de aplicaciones para estos algoritmos, como búsquedas, bases de datos, métodos de divide y conquistaras y estructura de datos

En este marco teórico vamos a profundizar en cómo funcionan los algoritmos de ordenamiento más utilizados, mostrando paso a paso cómo se resuelve el orden de una lista, para poder comprender en qué situaciones es conveniente aplicar cada uno de los algoritmos

Bubble sorting

Complejidad temporal, caso promedio: O(n^2)

Uno de los algoritmos más simples de implementar, si no es que es el más simple. Se comparan todos los elementos de la lista de a pares, y ordenándolos, así se repite toda la lista hasta que esté todo ordenado. Fácil de entender, no es rápido en listas largas.



Insertion sort:

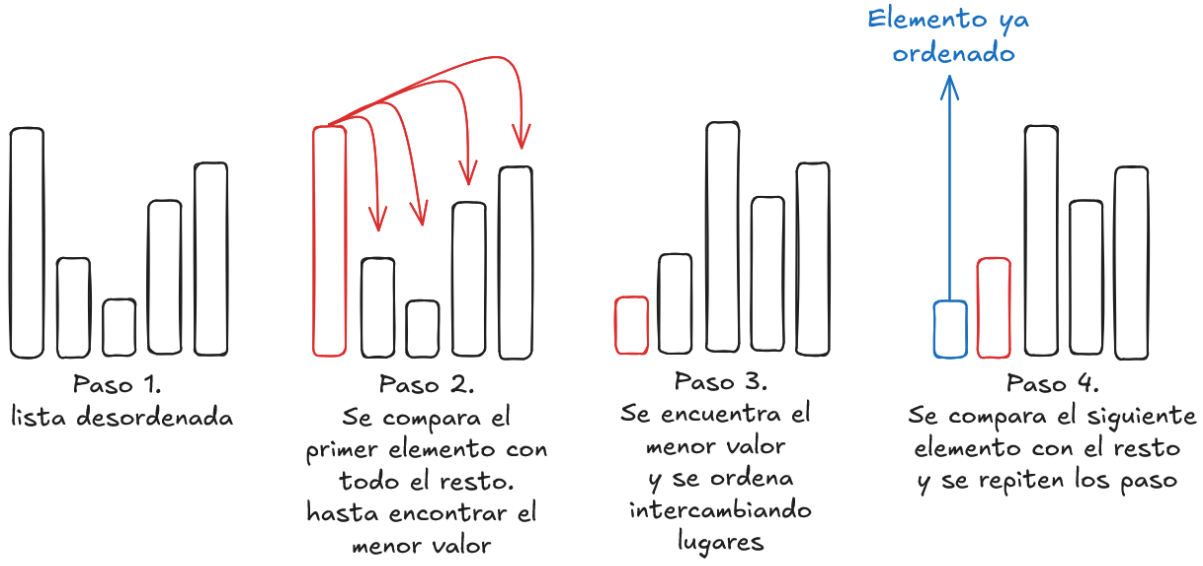
Complejidad temporal, caso promedio: O(n^2)

Este algoritmo compara un elemento (comenzado desde el segundo) hasta el inicio de la lista, comparando con todos los otros elementos e insertandolo donde corresponde

Selection Sort

Complejidad temporal, caso promedio: O(n^2)

Este algoritmo busca el elemento más pequeño de la lista y lo intercambia con el primer elemento. Continúa este proceso (buscando el segundo elemento más pequeño y acomodandolo) hasta que se encuentre ordenada la lista.



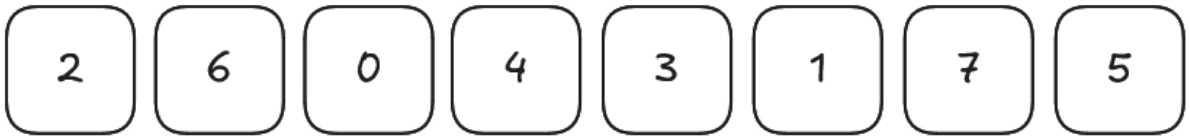
Quicksort

Complejidad temporal, caso promedio: O(n log n)

Este algoritmo divide la lista en dos partes, y luego ordena los dos segmentos de manera recursiva hasta completar el orden de la lista.

Funcionamiento del quicksort:

1. **Recibimos la lista desordenada**



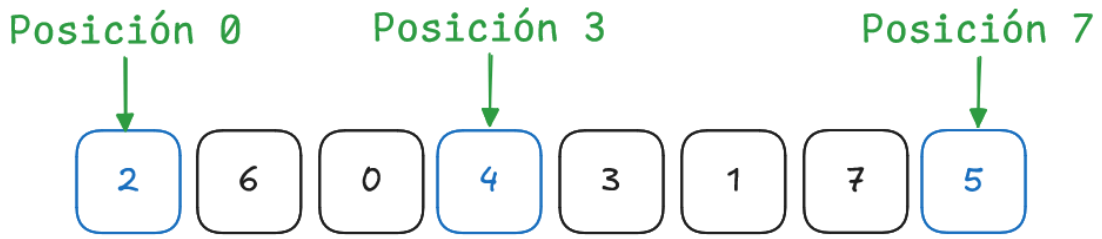
Paso 1. Lista desordenada

2. **Elegimos el Pivot**

Elegimos un “Pívot” que sería el elemento que divide la lista.
¿Cómo se elige un pivot? Tenemos varias opciones.

- **Elegir la primera posición.** Este método simplemente usa el primer elemento como Pivot para dividir la lista, es el más rápido en el caso de que la lista ya esté ordenada
- **Elegir la última posición.** En este caso, se utiliza el último elemento como pivot. No es eficiente si la lista ya se encuentra ordenada
- **Elemento al azar.** Simplemente se elige un elemento de la lista al azar, este método nos da menos probabilidad de elegir el peor elemento posible para utilizar de pívot (como sería por ejemplo elegir el último elemento si la lista está ordenada), que nos agregaría tiempo al ordenamiento de la lista.
- **Mediana de tres .** para calcular el pívot medio de entre 3 elementos, el primero, el último y el centro y se elige el valor del medio de entre los tres. Por ejemplo, si tenemos 2, 4 y 5 el valor sería 4.

Elegimos mediana de 3 para encontrar el pivot



Paso 2. Se elige el pivot por mediana de 3

- Se calcula el valor central entre la posición 0 y la 7. Que es la posición 3

- Se calcula el valor medio entre 2, 4 y 5 que es 4. Se asigna como pivot

Como calcular el pivot por mediana de 3:

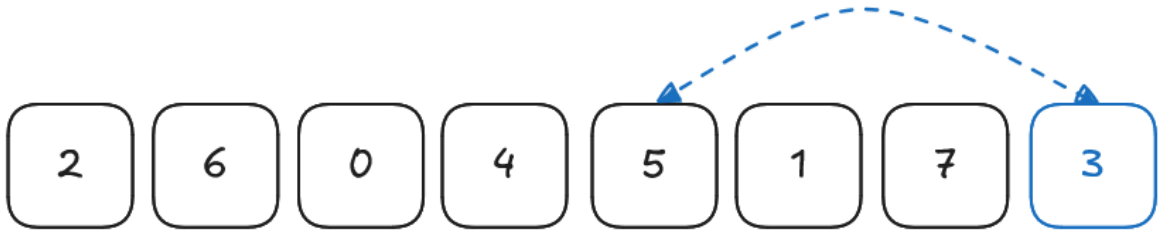
- Se encuentra primero el valor de la posición central que se calcula como **(posición más alta + posición más baja) // 2**. Esto utilizando los índices de la lista.

  En este caso seria: (Posición más alta + Posición más baja) // 2
  (0 + 7) // 2 = 3. La posición central es el **índice 3** que contiene el **valor 4**.

- Luego calculamos la mediana de 3 entre 2, 4 y 7
  En este caso ya se encuentran ordenados, el valor central es el 4

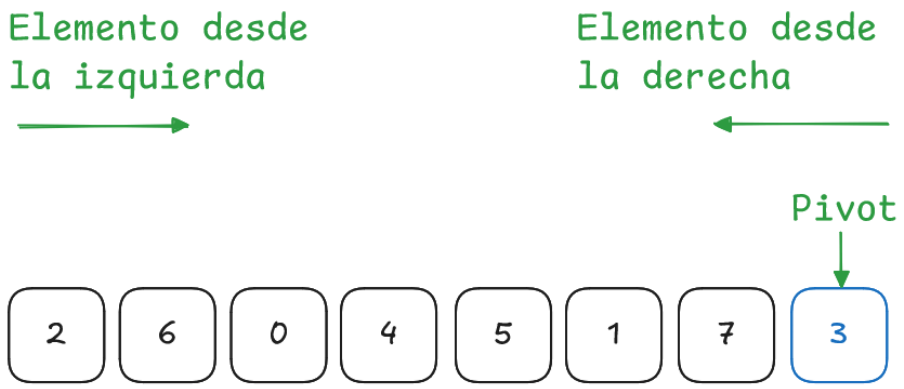
3. **Comenzamos a ordenar**

   Se mueve el Pivot al final de la lista.



Paso 3.1 Movemos el pivot al final de la lista, se intercambian posiciones con el ultimo valor

Se comienza a buscar dos cosas:
El primer elemento partiendo desde la izquierda que sea mayor que el pivot
El primer elemento partiendo desde la derecha que sea menor que el pivot

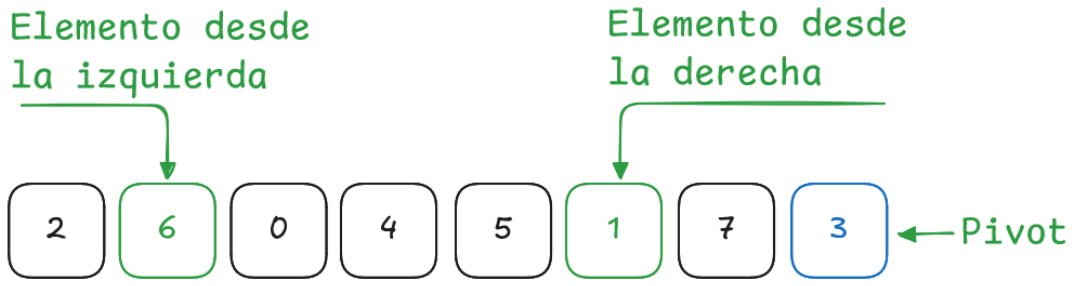


Paso 3.2

Se buscan 2 cosas:

1. Primer Item desde la izquierda que sea MAYOR que el pivot
2. Primer Item desde la derecha que sea MENOR que el pivot

Encontramos los elementos:
El valor 6 que es el primero mayor que 3 desde la izquierda.
El valor 1 que es el primero menor que 3 desde la derecha.

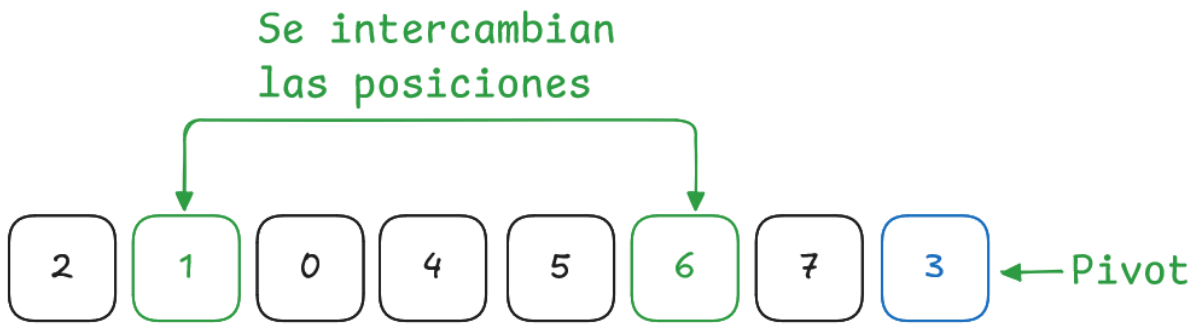


Paso 3.3

Se buscan 2 cosas:

1. Primer Item desde la izquierda que sea MAYOR que el pivot
2. Primer Item desde la derecha que sea MENOR que el pivot

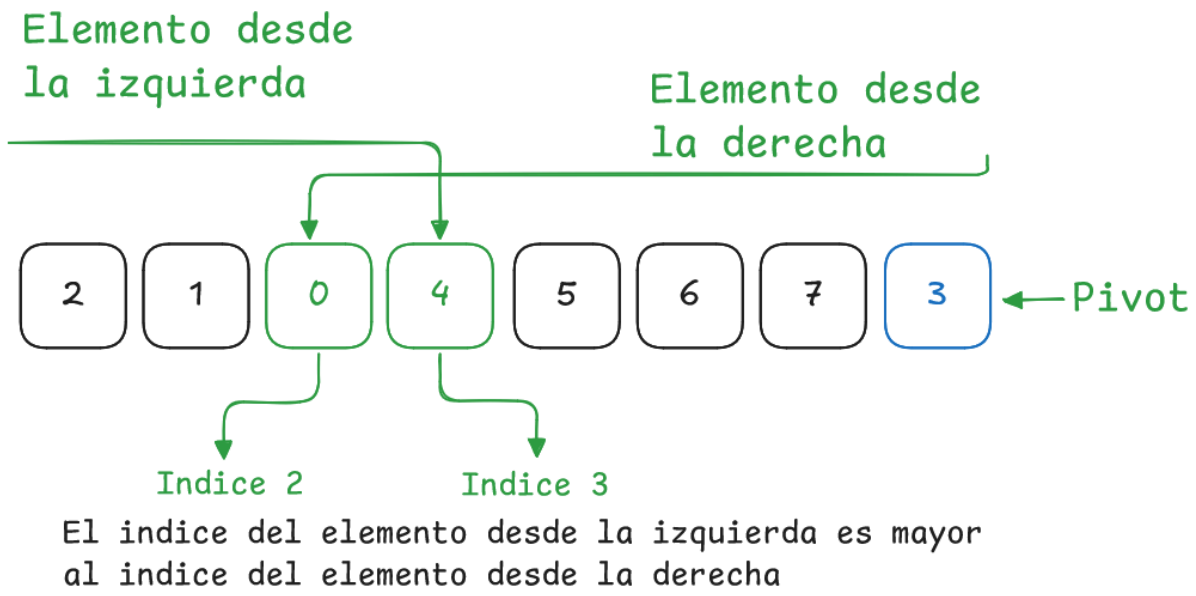
En el siguiente paso se intercambian estos 2 elementos de posición



Paso 3.4

Esto se repite, ordenando los números hasta que el **índice del elemento de la izquierda** sea mayor que el **índice del elemento de la derecha**

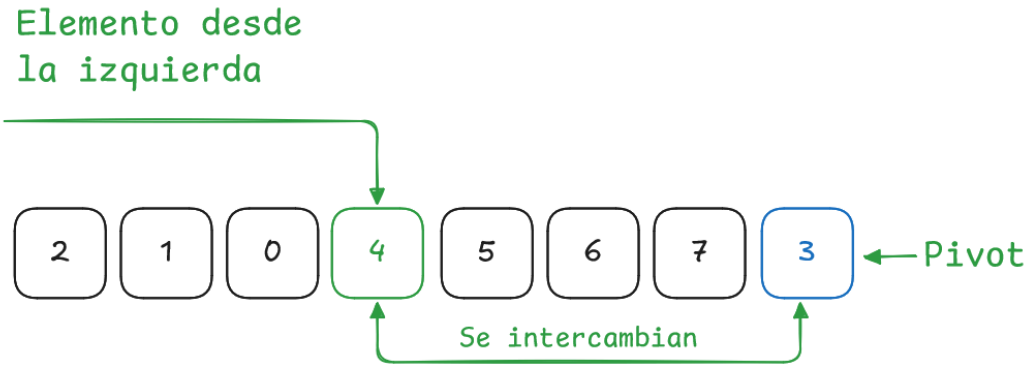
Como podemos ver en este paso:



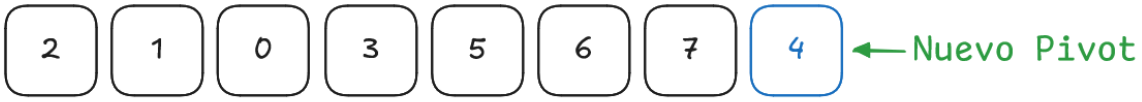
Paso 3.5

Cuando esto sucede, dejamos de buscar elementos y cambiamos el Pivot.

Este se intercambia con el nuevo elemento desde la izquierda que encontramos (el que tenía el índice mayo al elemento desde la derecha)



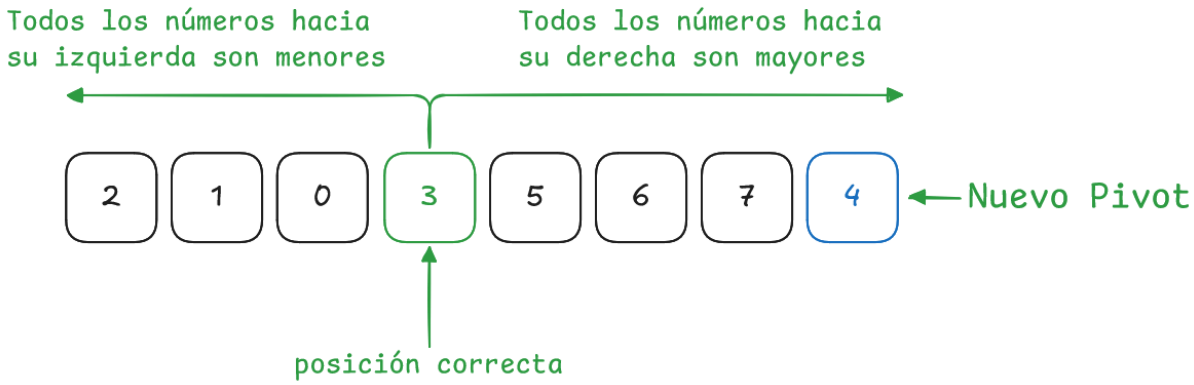
Para el siguiente paso, se intercambian el elemento desde la izquierda, con el Pivot. Quedando de la siguiente manera:



Paso 3.6

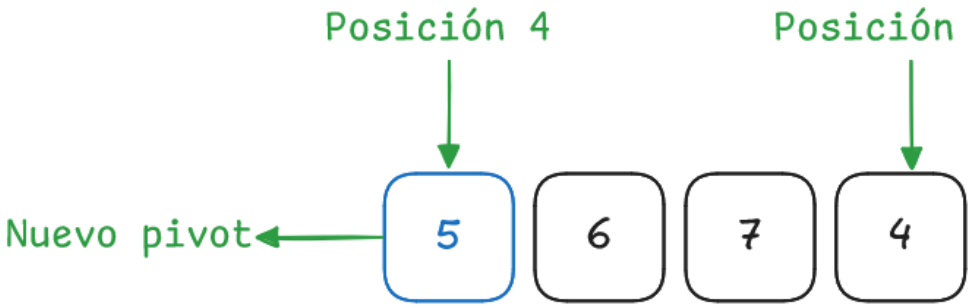
Una vez hecho esto, sabemos que el número 3 (posición 4) está en su posición correcta. Podemos comprobarlo viendo que todos los elementos a su izquierda son menores y todos los elementos a su derecha son mayores

Podemos comprobar que el numero 3 ya se encuentra en su posición correcta:



4. **Quicksort es recursivo.** continuamos ordenando la lista más grande que quedó, la de la derecha. Ya que ahora el 3 (que está en su posición correcta) divide la lista en 2 partes a ordenar.

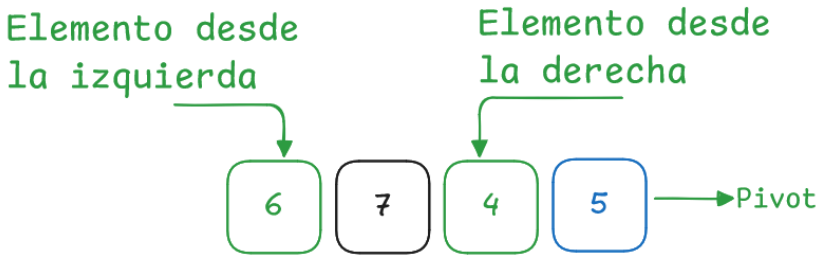
Volvemos a encontrar la mediana de 3 entre la lista de la derecha: [5, 6, 7, 4]
Y **la asignamos como pivot.**



Paso 4.1

Utilizamos mediana de 3 nuevamente para encontrar el pivot.
La mediana entre 5, 6 y 4 es: el numero 5

Volvemos a repetir los pasos de ordenamiento:

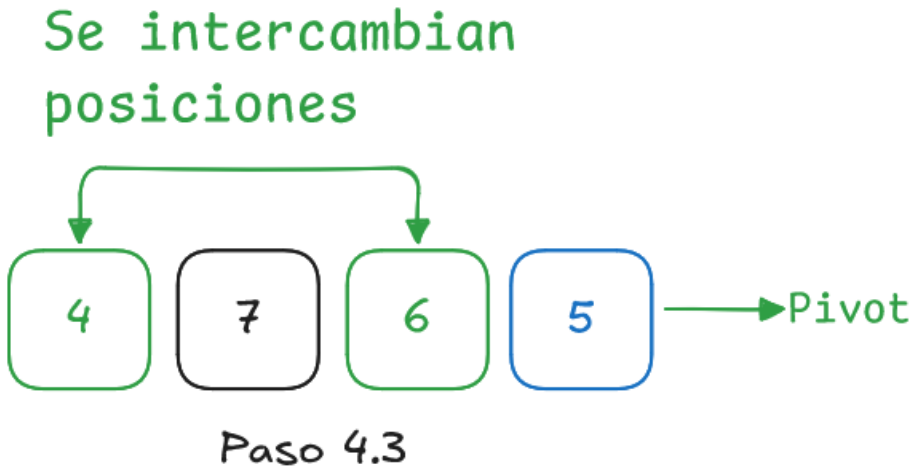


Paso 4.2

Volvemos a buscar los elementos:

1. Primer Item desde la izquierda que sea MAYOR que el pivot
2. Primer Item desde la derecha que sea MENOR que el pivot

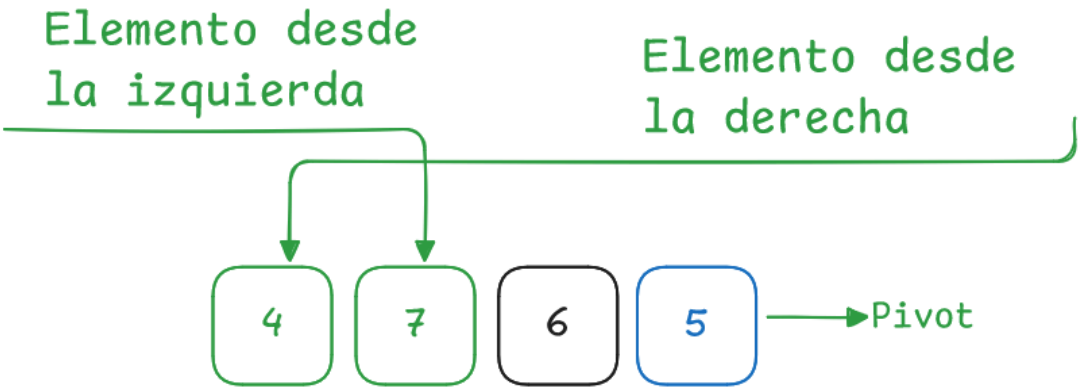
Se intercambian las posiciones de los elementos:



Paso 4.3

Se reitera la búsqueda de elementos:

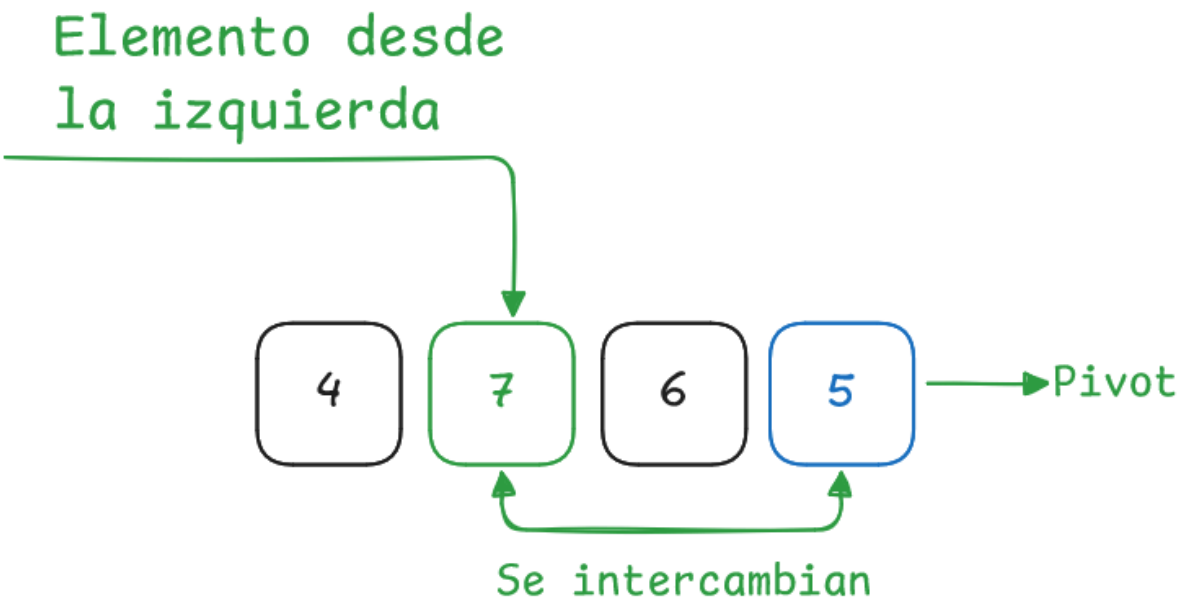
Nos damos cuenta que nuevamente que la posición del elemento desde la izquierda es mayor a la posición del elemento de la derecha

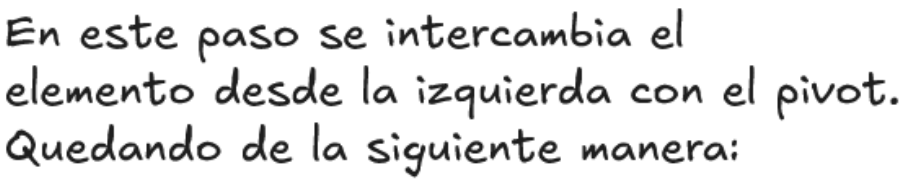


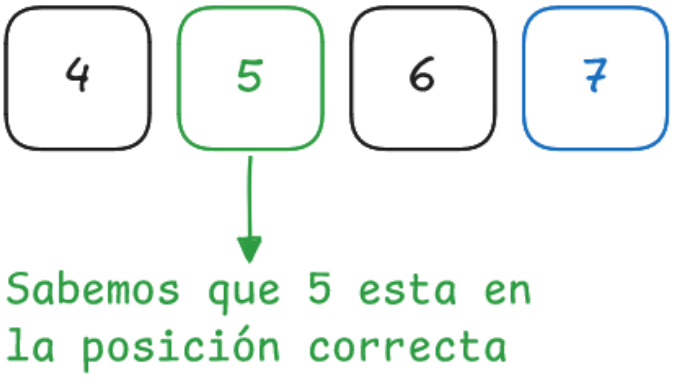
Paso 4.4

El indice del elementos de la izquierda es mayor al indice del elemento de la derecha

Se intercambia el elemento desde la izquierda con el pivot:



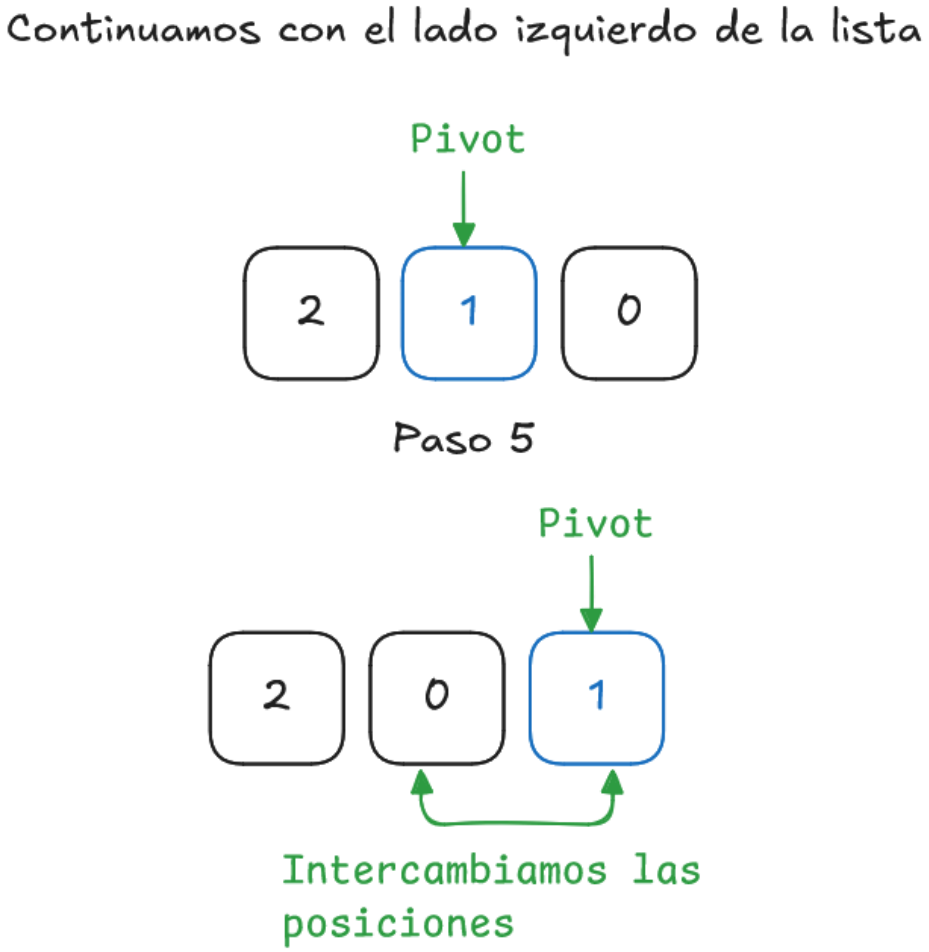




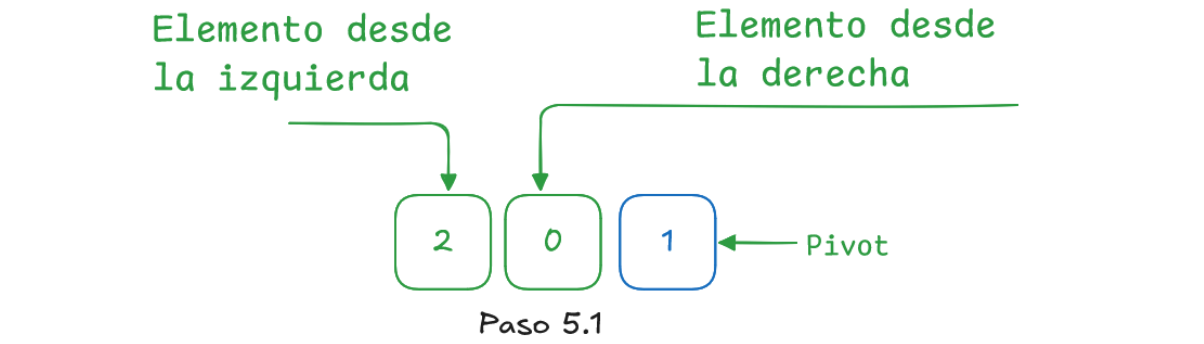
En este paso podemos verificar que esta parte de la lista está ordenada correctamente.
Ya que todos los elementos a su izquierda son menores y los elementos a su derecha son mayores

5. **Ordenar última parte de la división de la lista**
   Como último paso nos falta repetir nuevamente todo para ordenar la parte corta que quedó de la lista:

   Elegimos el número 1 como Pivot, de entre 0, 1 y 2



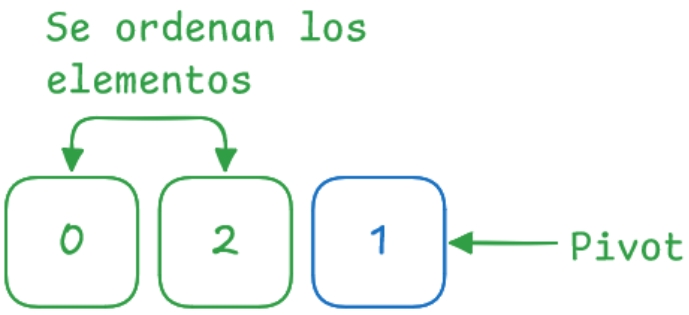
Se vuelve a buscar los elementos de la izquierda y de la derecha:



Se buscan 2 cosas:

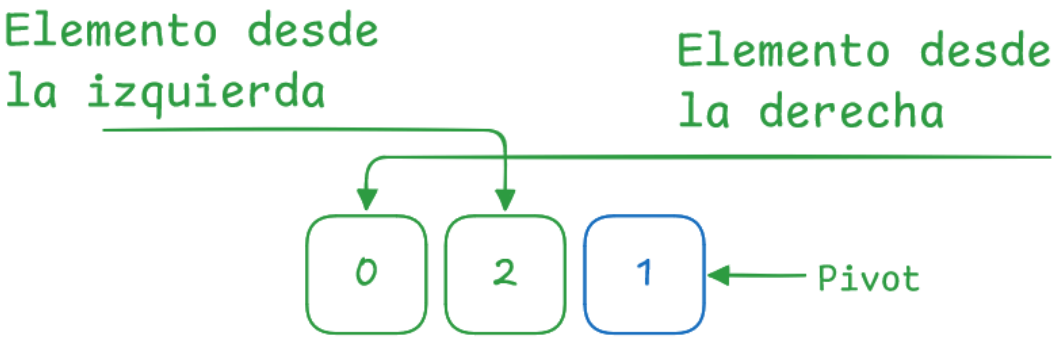
1. Primer Item desde la izquierda que sea MAYOR que el pivot
2. Primer Item desde la derecha que sea MENOR que el pivot

Se intercambian las posiciones de estos elementos:



Paso 5.2

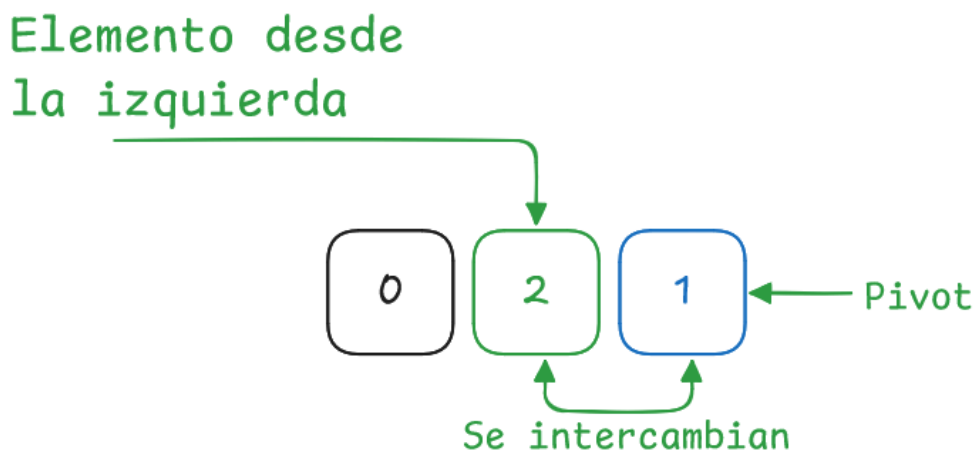
En la proxima busqueda podemos ver que otra vez el índice del elemento de la izquierda es mayor:



Paso 5.3

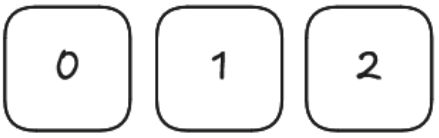
El indice del elemento de la izquierda es mayor al indice de elemento de la derecha

Quedando esta parte de la lista ordenada:



Paso 5.4

Aqui se intercambia el elemento de la izquierda con el pivot.
Quedando de la siguiente manera:



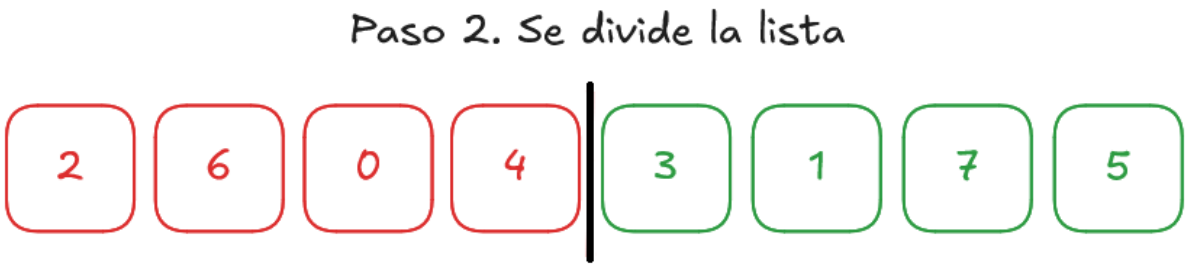
Finalmente tenemos la lista completa ordenada:

Lista ordenada

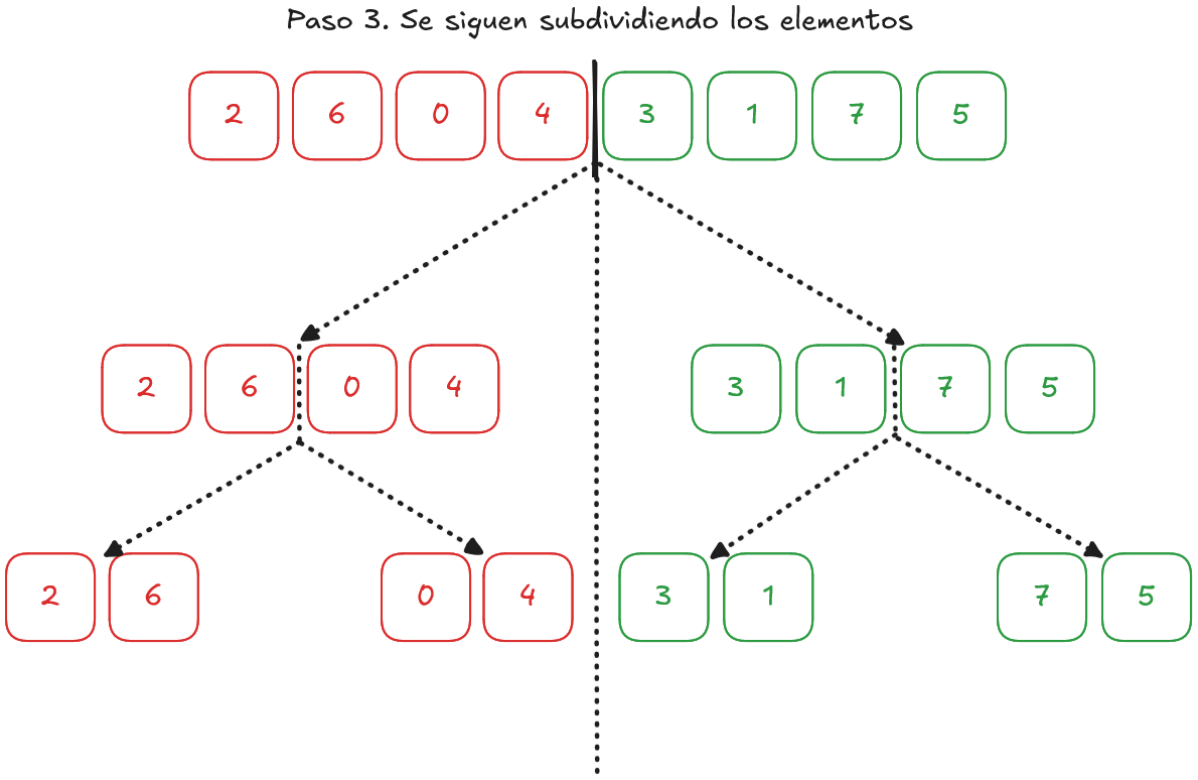
Merge Sort

Complejidad temporal, caso promedio: O(n log n)

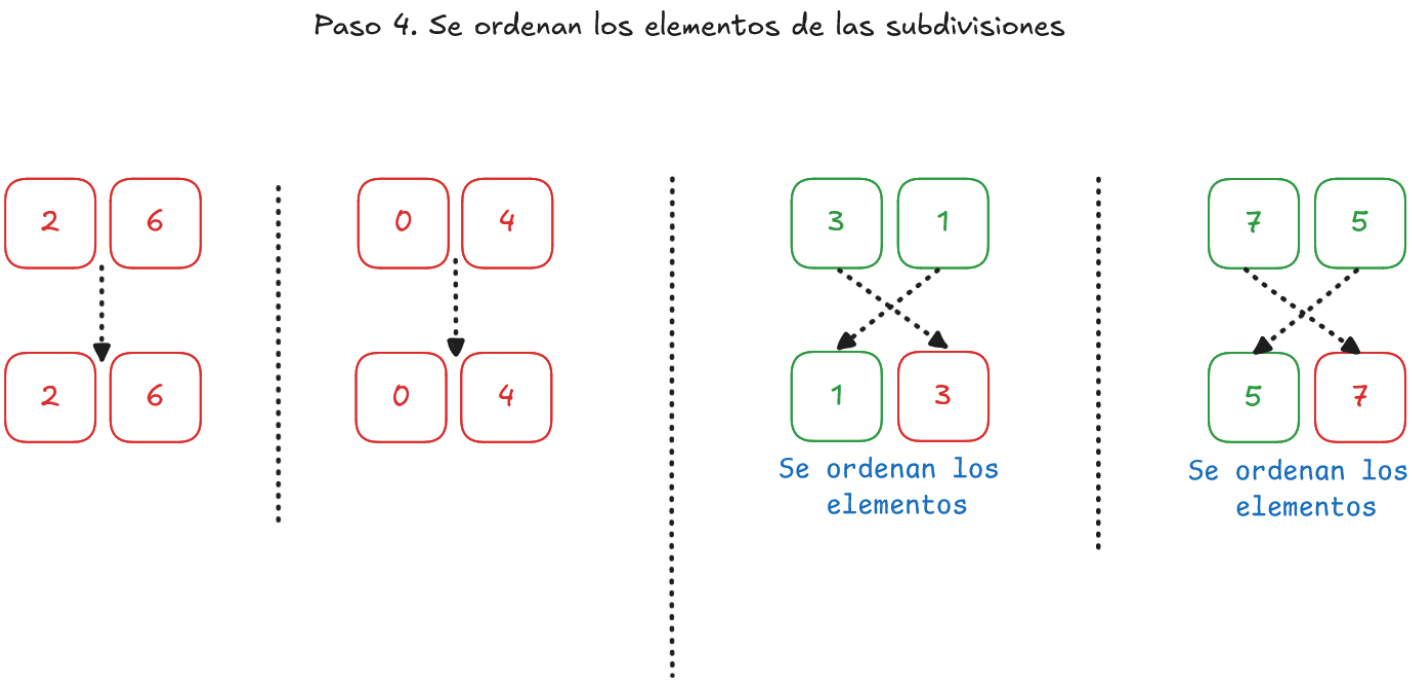
Este algoritmo de ordenamiento divide la listas en dos partes, y sus subdivisiones también, ordenandolas de manera recursiva y luego uniendo las partes ordenadas



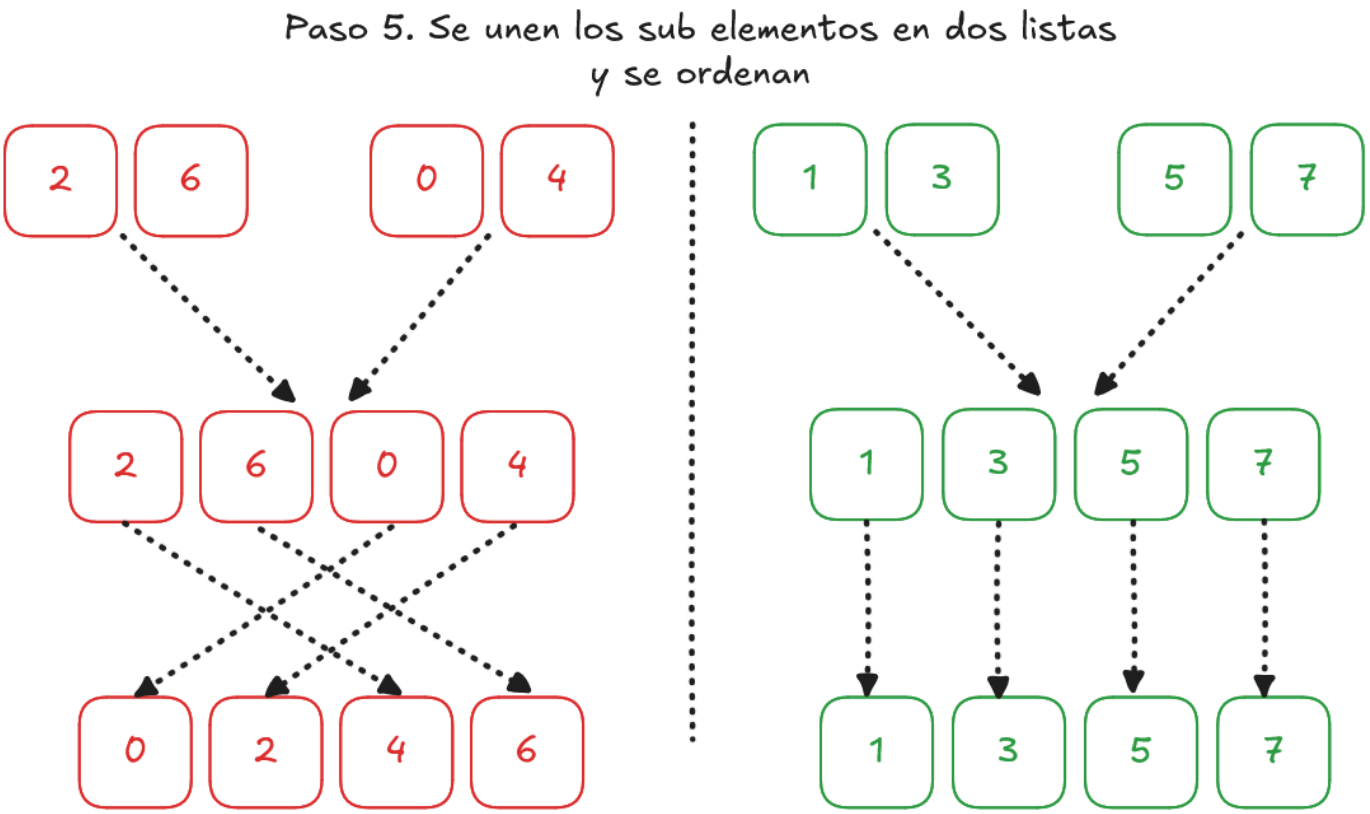
La lista se continúa dividiendo hasta formar grupos más pequeños:



Se ordenan los subelementos



Luego se unen en dos listas (que sería la lista completa dividida en dos) y se ordenan



Al final unimos las dos listas y las ordenamos.

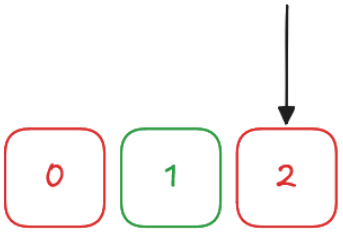
Paso 6. Se unen las dos listas y se ordena en una lista final ordenada

- Comparamos el primer elemento de la lista izquierda con los elementos de la lista derecha hasta encontrar uno menor.

- Como no hay menor que 0, se pone en la primer posicion

-Continuamos con las comparaciones, comparamos el 2 con todos los elementos de la lista derecha encontramos el 1 que es menor, se ordena

- Se continua comparando el 2 con los elementos de la lista derecha. como no hay se ordena el 2



- Se continua de manera recursiva, hasta tener la lista ordenada

¿Cómo se ordena el paso final? Un método posible es comparar todos los elementos de la lista izquierda con los elementos de la lista derecha y se va ordenando a medida que se encuentra uno menor, o no se encuentra ninguno entonces se agrega el número el cual se está siendo comparado con otros.

Caso Práctico

En este caso práctico se desarrolló un programa en Python que muestra mediante prints en la consola, como funciona un algoritmo de Bubble Sort paso a paso, de manera que se pueda comprender cómo funciona este algoritmo más simple, que es preferible para listas cortas.

Se eligió Bubble Sort debido a que es uno de los más simples de explicar paso a paso, para de esta manera poder comprender más fácilmente los conceptos básicos de un algoritmo de ordenamiento.

Por la parte de búsqueda, se desarrolló un programa que explica paso a paso como realiza una búsqueda binaria y muestra los pasos que realiza para encontrar un número

Código del programa:

```
import time #Para hacer pausas y visualizar mejor el proceso
import random #para las listas radom
import os #para limpiar la consola en los pasos
from colorama import Fore, Back, Style #para colorear los prints

#declaracion de variables
posicion = 0     #Variable para las posiciones de la lista, se usa como
contador

#funcion for que imprime el grafico de la lista inicial
def imprimir_lista_inicial(lista):
      print(Fore.GREEN + f"lista inicial: {lista}" + Style.RESET_ALL)
#muestra la lista incial en verde con colorama
    for numero in lista_ordenada:
```

```
        print(Fore.GREEN + f"\n\n{numero} ", end ="" + Style.RESET_ALL)
#imprime los numeros de la lista
        for numero in range(numero):
              print("▫ ", end ="")     #imprime un cuadrado al lado de
cada numero de la lista

#funcion que imprime los pasos mientras ordenamos
def imprimir_lista_pasos_intermedios(lista_ordenada):
    global posicion
    for numero in lista_ordenada:
         if lista_ordenada[posicion] == numero:       #Si el numero a
imprimir uno de los dos que estamos comparando, se imprime en rojo
            print("\n")
            print(Fore.RED + f"{numero} ", end ="")
         elif lista_ordenada[posicion+1] == numero:       #el siguiente
numero que estamos comparando es posicion+1 tambien se marca en rojo
            print("\n")
            print(Fore.RED + f"{numero} ", end ="")
         else:            #si no es uno de los dos numeros, se imprime
sin estilo
            print("\n")
            print(Style.RESET_ALL + f"{numero} ", end ="")
        for numero in range(numero):
            print("▫ ", end ="")


#funcion que verifica si la lista esta ordenada
def funcion_lista_esta_ordenada(lista_ordenada):
    global posicion, tamaño_lista, lista_ordenada_chequeo

    if lista_ordenada == lista_ordenada_chequeo:        #se chequea que
la lista este ordenada
             print(Fore.GREEN + f"Lista inicial: {lista_inicial}" +
Style.RESET_ALL)
        print(Fore.YELLOW + "\nLISTA ORDENADA" + Style.RESET_ALL)
        print(Fore.YELLOW + f"{lista_ordenada}" + Style.RESET_ALL)

        #ciclo for que imprime el grafico de la lista ordenada al final
        for numero in lista_ordenada:
            print("\n")
            print(Fore.RED + f"{numero} "+ Style.RESET_ALL , end ="")
            for numero in range(numero):
                print(Fore.RED + "▫ " + Style.RESET_ALL, end = "")
```

```
            #mostramos la cantidad de iteraciones que llevo el programa
para finalizar el ordenamiento
            print(Fore.YELLOW + f"\n\nCantidad de iteraciones realizadas:
{pasos}" + Style.RESET_ALL)
            print("\nEste fue el programa sobre algoritmos de busqueda y
ordenamiento de Hugo Catalan y Matias Carro, Muchas gracias!\n")
                input(Fore.YELLOW + "\n\nPresione Enter para salir del
programa..." + Style.RESET_ALL)

        else:         #si no esta ordenada se vuelve a llamar a la funcion
que ordena
            funcion_ordenamiento(lista_ordenada)

#funcion que ordena la lista
def funcion_ordenamiento(lista_ordenada):
    #variable globales que maneja la funcion
    global posicion, lista_inicial, pasos

    #titulo con lista inicial
            print(Fore.GREEN + f"Lista inicial: {lista_inicial}" +
Style.RESET_ALL)
    print(Fore.YELLOW + f"\nOrdenando lista: " + Style.RESET_ALL, end =
"")

    #si la posicion es igual al largo de la lista menos 1, se vuelve a
0 para reiniciar el ordenamiento
    #esto reinicia el ordenamiento para recorrer la lista nuevamente
    if posicion == len(lista_ordenada)-1:
        posicion = 0

    #Imprime los dos numeros que se comparan al momento
      print(f"\nNumeros comparados en esta iteracion: " + Fore.RED +
f"{lista_ordenada[posicion]}    y    {lista_ordenada[posicion+1]}"    +
Style.RESET_ALL)
       #desicion que verifica si el primer numero que esta siendo
comparado es mayor al segundo numero, si lo es se rotan

            if  lista_ordenada[posicion]  >  lista_ordenada[posicion+1]:
#se chequea si la posicion es mayor a la posicion +1
                        numero_triangulado   =   lista_ordenada[posicion]
#si es asi, se intercambian los valores de lugar, se utiliza una tercer
variable para triangular
```

```
        lista_ordenada[posicion] = lista_ordenada[posicion+1]
        lista_ordenada[posicion+1] = numero_triangulado
        #se muestra al usuario el paso que se realizo
          print(f"Se ordenan los numeros {lista_ordenada[posicion+1]} y
{lista_ordenada[posicion]}      porque      "      +      Fore.RED      +
f"{lista_ordenada[posicion]} es menor a {lista_ordenada[posicion+1]}" +
Style.RESET_ALL)
    else:
                print(f"Estos  dos  numeros  se  encuentran  ordenados
correctamente")

    #se imprime como la lista va siendo ordenada al momento
         print(Fore.YELLOW  +  f"\nLista  ordenada  al  momento:  "  +
Style.RESET_ALL)
    print(f"{lista_ordenada}")

     #se  llama  a  la  funcion  que  imprime  la  lista  mientras  la  estamos
ordenando
    imprimir_lista_pasos_intermedios(lista_ordenada)

    posicion += 1           #se sube la posicion en un valor
    pasos += 1      #cantidad de pasos realizados

    #se espera que el usuario presione enter para continuar
        input(Fore.YELLOW  +"\n\nPresione  ENTER  para  continuar..."  +
Style.RESET_ALL)
    #limpia la pantalla para que el programa sea mas facil de entender
    os.system('cls')

     #se  llama  a  la  funcion  lista  ordenada,  que  verifica  si  se  logro
ordenar en este paso, si no, se vuelve a realizar el ordenamiento
    funcion_lista_esta_ordenada(lista_ordenada)

def validacion_datos(numero_ingresado): #Valida que el numero ingresado
sea correcto
     while True: #Se repite el loop hasta que la funcion retorne el
numero
        try: #intenta pasar el ingreso a un integer
            numero = int(numero_ingresado) #si es integer, se guarda en
la variable numero
              if numero >= 3 and numero <= 10: #verifica que el entero
sea positivo y menoro igual a 20 si no lo es, vuelve a pedir ingreso de
datos
```

```
                return numero #si es entero, positivo, y menor a 999, devuelve el numero
            else:
                print("\nDatos ingresados incorrectos")
                numero_ingresado = input("Ingrese otro numero, recuerde que tiene que ser un entero positivo entre 3 y 10:\n") #volvemos a pedir ingreso
            except ValueError: #en caso de error, el ingreso no era correcto. Tenia otros caracteres o era decimal
            print("\nDatos ingresados incorrectos")
            numero_ingresado = input("Ingrese otro numero, recuerde que tiene que ser un entero positivo entre 3 y 10:\n") #volvemos a pedir ingreso


#inicio del programa titulo
print(Fore.GREEN +"Bienvenido al programa de Algoritmos de Busqueda y ordenamiento, realizado por Hugo Catalan y Matias Carro."+ Style.RESET_ALL)
print("Para el trabajo practico integrador 2 de la materia de Programacion UTN TUPaD")

#Comienza parte busqueda binaria

def busquedaBinaria(lista, objetivo): # Defino la función de búsqueda binaria
    inicio = 0 #Defino el inicio de la lista
    fin = len(lista) - 1 #Defino el fin de la lista

    while inicio <= fin:# Mientras el inicio sea menor o igual al fin
        medio = (inicio + fin) // 2# Calculo el punto medio de la lista
        print(f"📍 Verificando índice {medio} (valor: {lista[medio]})") # Mostramos la posicion y el valor que estamos analizando
        time.sleep(1)  # Espera 1 segundo para visualizar

         if lista[medio] == objetivo: # comparo el valor del medio con el objetivo
                print(f"✅ El número: {objetivo} se encuentra en la posición {medio}")
                print(f" Felicitaciones, has encontrado el número {objetivo} en la lista.")
            return medio
```

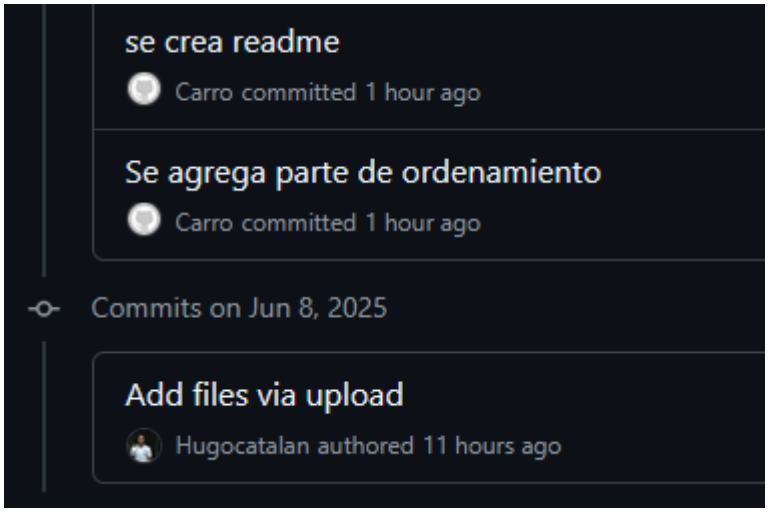
```
        elif lista[medio] < objetivo:# Si el valor del medio es menor que el objetivo a la derecha
                print(f"🔽 Buscando el {objetivo} en la mitad derecha...\n")
            inicio = medio + 1 # actualizo el inicio a la mitad derecha
        else:#si el valor del medio es mayor que el objetivo busco a la izquierda
                print(f"🔼 Buscando el {objetivo} en la mitad izquierda...\n")
            fin = medio - 1 # actualizo el fin a la mitad izquierda
        time.sleep(1)

        print(f"❌ El numero:{objetivo} que esta buscado no fue encontrado")
    return -1

# Prueba
lista = range(0,100,2) # Defino una lista de números del 0 al 100, de 2 en 2
objetivo = int(input("Ingrese un número a buscar en la lista: "))
busquedaBinaria(lista, objetivo)

#Comienza seccion de ordenamiento

input(Fore.YELLOW + "\n\nPresione Enter para continuar con la parte de ordenamiento..." + Style.RESET_ALL)
os.system('cls')
print(Fore.GREEN + "\nPrograma que muestra como se ordena con Bubble Sorting una lista al azar, paso a paso" + Style.RESET_ALL)
print("\nPara poder demostrar el ejemplo, ingrese un numero para elegir el total de elementos de la lista aleatoria")
print("Por favor un numero entre 3 y 10 ya que mas de 10 el programa seria demasiado largo para su demostracion practica")
#Se solicita ingreso al usuario
numero_ingresado = input("Ingrese un numero:\n")

#verificacion de ingreso de datos
tamaño_lista = validacion_datos(numero_ingresado)

#Se crea la lista random
lista = random.sample(range(1, tamaño_lista+1), tamaño_lista)
lista_inicial = list(lista)     #guardamos la lista inicial
```

```
lista_ordenada = list(lista)       #vamos a utilizar esta lista para
ordenar
lista_ordenada_chequeo = sorted(lista)       #lista ordenada por python
para chequear que hayamos ordenado correctamente
pasos = 0 #contador con los pasos que realiza bubble sort para resolver

#se llama a la funcion que imprime la lista inicial
imprimir_lista_inicial(lista)

#pausa en el programa, espera a que el usuario presione enter para
continuar
input(Fore.YELLOW + "\n\nPresione Enter para comenzar a ordernar la
lista..." + Style.RESET_ALL)
#limpia la pantalla para que el programa sea mas facil de entender
os.system('cls')

#inicio del ordenamiento, se llama a la funcion que verifica si la
lista esta ordenada
funcion_ordenamiento(lista_ordenada)
```

Herramientas y recursos utilizados

Librerías de Python:
Random: Para generar las listas al azar
Time: Para hacer pausas y visualizar mejor el proceso
os: Para utilizar comandos del sistema, en este caso limpiar la consola para mantener claro los pasos en el ordenamiento
colorama: Para darle color a los prints de la consola

Se utilizó GitHub para el trabajo colaborativo y el control de versiones.



Metodología utilizada

1. Se buscó en varias fuentes confiables una explicación detallada de los algoritmos de ordenamiento

2. Investigación previa:
    - https://www.freecodecamp.org/news/sorting-algorithms-explained-with-examples-in-python-java-and-c/
    - https://medium.com/basecs/pivoting-to-understand-quicksort-part-1-75178dfb9313
    - https://www.geeksforgeeks.org/quick-sort-algorithm/#ChoosingAPivot
    - https://www.youtube.com/watch?v=RGuJga2Gl_k&list=PLFKlI4LYZ8Kw-RQ7Bqjq1FPSUt1oMmDIT&index=9
    - https://joomat.wordpress.com/2012/06/20/algoritmos-de-busqueda/
    - https://academia-lab.com/enciclopedia/algoritmo-de-busqueda-binaria/
    - https://numerentur.org/busqueda/
    - https://numerentur.org/busqueda/

3. Se desarrolló la Teoría sobre búsqueda y ordenamiento en base a la investigación.

4. Se implementó en python uno de los algoritmos, centrándose en la explicación de su funcionamiento.

5. Se utilizó GitHub para el desarrollo en conjunto del programa.

6. Se explicó el programa en grupo, dividiendo el funcionamiento para cada uno explicar una parte.

Conclusión:

Los algoritmos de búsqueda y ordenamiento están en muchos de los software o páginas web que utilizamos día a día. Comprender su funcionamiento y aplicación es de suma importancia como programadores .

Poder comprender cómo nuestro lenguaje ordena o realiza búsquedas nos convierte en mejores programadores, ya que deja de ser una simple función la cual le pedimos al lenguaje que solucione por nosotros y pasa a ser un código el cual sabemos como funciona, porque se utiliza y cómo se aplica.

El aprender cómo funciona la Notación O nos da las bases para comprender el gasto operacional de un algoritmo que vayamos a implementar, de esta manera podemos optimizar nuestro código y volverlo más eficiente.

En este trabajo se investigó la teoría sobre búsqueda y algoritmos de ordenamiento, se realizó un trabajo en equipo, dividiendo tareas de investigación, desarrollo de teoría y software.

Resultados obtenidos

- El programa ordena correctamente una lista de números en bubble sort y realiza una búsqueda binaria de una lista- Además de mostrar los pasos que realiza en pantalla y explicarlos.
- Se aprendió en profundidad el funcionamiento de los algoritmos de búsqueda y ordenamiento y los pasos que realizan para llegar a su objetivo.
- Se comprendió la complejidad de los algoritmos y en qué casos es conveniente utilizarlos.
- Se realizaron varias pruebas, se utilizaron las herramientas adquiridas durante la cursada y se aprendieron varios metodos nuevos durante la investigación y desarrollo

Dificultades que surgieron:

La principal dificultad que surgió fue al momento de grabar el video y explicar la teoría y funcionamiento de los algoritmos, se vuelve difícil el condensar tanta cantidad de información en pocos minutos, lo que nos lleva a tener un video mas largo de lo esperado en cuanto a teoría, intentaremos en un futuro tratar de reducir más el tiempo de la explicación.

Bibliografía:

- Stanford University - CS106B Programming Abstractions - Sorting Lecture.
- Washington State University - Advanced Data Structures - Sorting AlgorithmsSorting Algorithms - Cpt S 223. School of EECS, WSU
- Stanford University - CS106B: Programming Abstractions - Big-O Notation and Algorithmic Analysis
- HarvardX: CS50's Introduction to Computer Science - Linear Search - 2018
- HarvardX: CS50's Introduction to Computer Science - Binary Search - 2018

CS50’s Introduction to Computer Science

https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1218/lectures/21-sorting/Lecture21Slides.pdf

https://eecs.wsu.edu/~ananth/CptS223/Lectures/sorting.pdf

https://web.stanford.edu/class/archive/cs/cs106b/cs106b.1218/lectures/07-bigo/Lecture7Slides.pdf

https://cs50.harvard.edu/ap/2020/assets/pdfs/linear_search.pdf

https://cs50.harvard.edu/ap/2020/assets/pdfs/binary_search.pdf

Anexo:

Link al repositorio:

https://github.com/Hugocatalan/busqueda_ordenamiento

Link al video de youtube:

https://www.youtube.com/watch?v=3GZ-ZmUZhr4

Links a las diapositivas:

Diapositiva búsqueda binaria:
https://drive.google.com/file/d/13w1sOvvL2Ayo1NsD25zBTXnbw1D0asnd/view?usp=sharing

Búsqueda Lineal:
https://drive.google.com/file/d/1nN_t6slLhdBT0LqPq3hmxpUbQGiBUZS0/view?usp=sharing

Algoritmos de ordenamiento:
https://drive.google.com/file/d/1hdvCzxR_FPDOa1CYFsDZBKrnZQaDqzCc/view?usp=sharing

Programa en ejecución:

Pantalla inicial:

```
Programa que muestra como se ordena con Bubble Sorting una lista al azar, paso a paso

Para poder demostrar el ejemplo, ingrese un numero para elegir el total de elementos de la lista aleatoria
Por favor un numero entre 3 y 10 ya que mas de 10 el programa seria demasiado largo para su demostracion practica
Ingrese un numero:
6
lista inicial: [1, 2, 5, 4, 3, 6]


1 □

2 □□

5 □□□□□

4 □□□□

3 □□□

6 □□□□□□

Presione Enter para comenzar a ordernar la lista...
```

Paso intermedio del ordenamiento:

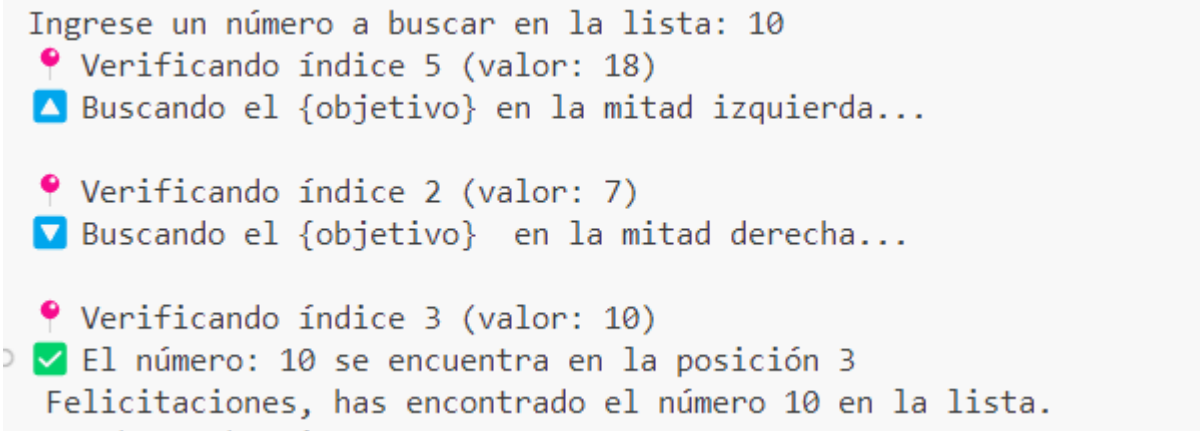
```
Lista inicial: [1, 2, 5, 4, 3, 6]

Ordenando lista:
Numeros comparados en esta iteracion: 2 y 5
Estos dos numeros se encuentran ordenados correctamente

Lista ordenada al momento:
[1, 2, 5, 4, 3, 6]


1 □

2 □□

5 □□□□□

4 □□□□

3 □□□

6 □□□□□□

Presione ENTER para continuar...
```

Lista ya ordenada:

```
Lista inicial: [2, 1, 6, 5, 4, 3]

LISTA ORDENADA
[1, 2, 3, 4, 5, 6]


1 □

2 □□

3 □□□

4 □□□□

5 □□□□□

6 □□□□□□

Cantidad de iteraciones realizadas: 13


Presione Enter para salir del programa...
```

Programa de búsqueda binaria en funcionamiento:

Encontrando un numero:



Si el numero ingresado no está en la lista, se le informa al usuario: