

|  |
| --- |
| TECNICATURA UNIVERSITARIA EN PROGRAMACIUÓN A DISTANCIA |
| Trabajo Integrador - Propuesta de Investigación. Cátedra de Programación I |
| Búsqueda y ordenamiento |

|  |
| --- |
| German Luis Dagatti  Nicolas Gabriel Demiryi |

# **Introducción**

La búsqueda y el ordenamiento de datos son dos tareas fundamentales en la programación y el procesamiento de información. Estos algoritmos permiten organizar y recuperar datos de manera eficiente, lo cual es crucial en aplicaciones que manejan grandes volúmenes de información. En este informe, exploraremos los conceptos básicos de búsqueda y ordenamiento, y proporcionaremos ejemplos de implementación en Python.

# **Búsqueda de Datos**

## Búsqueda Lineal

La búsqueda lineal es el algoritmo más simple para encontrar un elemento en una lista. Consiste en recorrer la lista elemento por elemento hasta encontrar el valor deseado.

**Ejemplo de Búsqueda Lineal en Python**

# Definición de la función

def busqueda\_lineal(lista, objetivo):

    for i in range(len(lista)):

        if lista[i] == objetivo:

            return i  # Retorna la posición del elemento

    return -1  # Retorna -1 si el elemento no se encuentra

# Ejemplo de uso

if \_\_name\_\_ =="\_\_main\_\_":

    lista = [3, 5, 2, 8, 1, 9]

    objetivo = 8

    posicion = busqueda\_lineal(lista, objetivo)

    if posicion != -1:

        print(f"El elemento {objetivo} se encuentra en la posición {posicion}.")

    else:

        print(f"El elemento {objetivo} no se encuentra en la lista.")

Cuando corremos el script en Python, nos devuelve lo siguiente

Imagen que contiene Diagrama

El contenido generado por IA puede ser incorrecto.

**Implementación de la Búsqueda Lineal en Python**

A continuación, se presenta una implementación detallada de la búsqueda lineal en Python.

**Explicación Paso a Paso**

1. **Función Principal (busqueda\_lineal)**:
   * La función toma dos parámetros: lista (la lista de elementos) y objetivo (el elemento objetivo a encontrar).
   * Utiliza un bucle for para recorrer la lista elemento por elemento.
   * Utiliza la función range y len para obtener un vector que permita recorrer cada elemento en la lista.
2. **Comparación y Retorno**:
   * En cada iteración, se compara el elemento actual (lista[i]) con el elemento objetivo (objetivo).
   * Si se encuentra un elemento que coincide con el objetivo, la función retorna el índice de ese elemento. (posicion)
   * Si se completa el bucle sin encontrar el elemento objetivo, la función retorna -1

**Ejemplo de Uso**

import busqueda\_lineal as bl

lista = [4, 2, 7, 1, 9, 3]

objetivo = 7

resultado = bl.busqueda\_lineal(lista, objetivo)

if resultado != -1:

    print(f"El elemento {objetivo} se encuentra en la posición {resultado}.")

else:

    print(f"El elemento {objetivo} no se encuentra en la lista.")

* **Lista Original**: [4, 2, 7, 1, 9, 3]
* **Elemento Objetivo**: 7
* **Resultado**: El elemento 7 se encuentra en la posición 2.

Imagen que contiene Texto

El contenido generado por IA puede ser incorrecto.

**Complejidad del Algoritmo**

* **Complejidad Temporal**:
  + **Mejor Caso**: *O*(1) si el elemento objetivo está en la primera posición.
  + **Caso Promedio**: *O*(*n*) donde *n* es el número de elementos en la lista.
  + **Peor Caso**: *O*(*n*) si el elemento objetivo está en la última posición o no está en la lista.
* **Complejidad Espacial**:
  + **In-place**: *O*(1) ya que no requiere espacio adicional significativo.

**Ventajas y Desventajas**

**Ventajas**:

* **Sencillo de Implementar**: La búsqueda lineal es muy fácil de entender y codificar.
* **No Requiere Lista Ordenada**: Funciona con listas desordenadas.

**Desventajas**:

* **Ineficiente para Listas Grandes**: Tiene una complejidad temporal de *O*(*n*), lo que lo hace ineficiente para listas grandes.
* **No Utiliza Estructuras de Datos Especiales**: No aprovecha estructuras de datos más avanzadas que podrían mejorar el rendimiento.

**Uso Práctico**

La búsqueda lineal es especialmente útil en situaciones donde:

* La lista es pequeña.
* No se puede garantizar que la lista esté ordenada.
* Se prefiere una implementación simple y rápida.

En resumen, la búsqueda lineal es un algoritmo simple y eficiente para listas pequeñas o situaciones donde no se puede garantizar que la lista esté ordenada. Sin embargo, para listas grandes, se recomiendan algoritmos más eficientes como la búsqueda binaria (si la lista está ordenada) o estructuras de datos más avanzadas como tablas hash

## Búsqueda Binaria

La búsqueda binaria es un algoritmo más eficiente que funciona en listas ordenadas. Divide la lista en mitades sucesivas hasta encontrar el elemento deseado.

**Ejemplo de Búsqueda Binaria en Python**

def busqueda\_binaria(lista, objetivo):

    izquierda, derecha = 0, len(lista) - 1

    while izquierda <= derecha:

        medio = (izquierda + derecha) // 2

        if lista[medio] == objetivo:

            return medio  # Retorna la posición del elemento

        elif lista[medio] < objetivo:

            izquierda = medio + 1

        else:

            derecha = medio - 1

    return -1  # Retorna -1 si el elemento no se encuentra

# Ejemplo de uso

if \_\_name\_\_ =="\_\_main\_\_":

    lista\_ordenada = [1, 2, 3, 5, 8, 9]

    objetivo = 5

    posicion = busqueda\_binaria(lista\_ordenada, objetivo)

    if posicion != -1:

        print(f"El elemento {objetivo} se encuentra en la posición {posicion}.")

    else:

        print(f"El elemento {objetivo} no se encuentra en la lista.")

Cuando corremos el script en Python, nos devuelve lo siguiente

Imagen que contiene Diagrama

El contenido generado por IA puede ser incorrecto.

**Implementación de la Búsqueda Binaria en Python**

A continuación, se presenta una implementación detallada de la búsqueda binaria en Python.

**Explicación Paso a Paso**

1. **Inicialización**:
   * Se inicializan dos punteros, low y high, que representan los índices de inicio y fin de la lista, respectivamente.
2. **Bucle Principal**:
   * El bucle while se ejecuta mientras low sea menor o igual a high.
   * En cada iteración, se calcula el índice medio (mid) y se obtiene el valor en esa posición (mid\_value).
3. **Comparación y Actualización de Punteros**:
   * Si mid\_value es igual al target, se retorna el índice mid.
   * Si mid\_value es menor que target, se actualiza low a mid + 1 para buscar en la mitad derecha.
   * Si mid\_value es mayor que target, se actualiza high a mid - 1 para buscar en la mitad izquierda.
4. **Resultado**:
   * Si el bucle termina sin encontrar el target, se retorna -1.

**Ejemplo de Uso**

lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

objetivo = 7

resultado = binary\_search(lista, objetivo)

if resultado != -1:

print(f"El elemento {objetivo} se encuentra en la posición {resultado}.")

else:

print(f"El elemento {objetivo} no se encuentra en la lista.")

* **Lista Original**: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
* **Elemento Objetivo**: 7
* **Resultado**: El elemento 7 se encuentra en la posición 6.

**Complejidad del Algoritmo**

* **Complejidad Temporal**:
  + **Mejor Caso**: *O*(1) si el elemento objetivo está en el medio.
  + **Caso Promedio**: *O*(log*n*) donde *n* es el número de elementos en la lista.
  + **Peor Caso**: *O*(log*n*) si el elemento objetivo no está en la lista.
* **Complejidad Espacial**:
  + **In-place**: *O*(1) ya que no requiere espacio adicional significativo.

**Ventajas y Desventajas**

**Ventajas**:

* **Eficiente**: Tiene una complejidad temporal de *O*(log*n*), lo que lo hace muy eficiente para listas grandes.
* **Sencillo de Implementar**: La implementación es relativamente fácil de entender y codificar.

**Desventajas**:

* **Requiere Lista Ordenada**: Solo funciona con listas ordenadas.
* **No Aprovecha Estructuras de Datos Especiales**: No aprovecha estructuras de datos más avanzadas que podrían mejorar el rendimiento.

**Uso Práctico**

La búsqueda binaria es especialmente útil en situaciones donde:

* La lista está ordenada.
* Se busca un elemento específico de manera eficiente.
* Se prefiere una implementación simple y rápida.

En resumen, la búsqueda binaria es un algoritmo eficiente y ampliamente utilizado para encontrar elementos en listas ordenadas. Su complejidad temporal de *O*(log*n*) lo hace muy eficiente para listas grandes.

**Comparación entre búsqueda binaria y búsqueda lineal**

La búsqueda binaria y la búsqueda lineal son dos algoritmos diferentes para encontrar un elemento en una lista. Cada uno tiene sus propias ventajas y desventajas, y su elección depende del contexto específico de la aplicación. A continuación

| **Característica** | **Búsqueda Lineal** | **Búsqueda Binaria** |
| --- | --- | --- |
| **Complejidad Temporal** | *O*(*n*) | *O*(log*n*) |
| **Complejidad Espacial** | *O*(1) | *O*(1) |
| **Requiere Lista Ordenada** | No | Sí |
| **Implementación** | Sencilla | Sencilla |
| **Uso Práctico** | Listas pequeñas o desordenadas | Listas grandes y ordenadas |

# **Ordenamiento de Datos**

## Ordenamiento por Selección

El ordenamiento por selección es un algoritmo simple que divide la lista en dos partes: la parte ordenada y la parte no ordenada. En cada iteración, selecciona el elemento más pequeño de la parte no ordenada y lo coloca al final de la parte ordenada.

**Ejemplo de Ordenamiento por Selección en Python**

def ordenamiento\_seleccion(lista):

n = len(lista)

for i in range(n):

indice\_minimo = i

for j in range(i + 1, n):

if lista[j] < lista[indice\_minimo]:

indice\_minimo = j

lista[i], lista[indice\_minimo] = lista[indice\_minimo], lista[i]

return lista

*# Ejemplo de uso*

lista = [64, 25, 12, 22, 11]

lista\_ordenada = ordenamiento\_seleccion(lista)

print("Lista ordenada:", lista\_ordenada)

## Ordenamiento por Inserción

El ordenamiento por inserción construye la lista ordenada insertando cada elemento en su posición correcta dentro de la parte ya ordenada.

**Ejemplo de Ordenamiento por Inserción en Python**

def ordenamiento\_insercion(lista):

for i in range(1, len(lista)):

clave = lista[i]

j = i - 1

while j >= 0 and clave < lista[j]:

lista[j + 1] = lista[j]

j -= 1

lista[j + 1] = clave

return lista

*# Ejemplo de uso*

lista = [12, 11, 13, 5, 6]

lista\_ordenada = ordenamiento\_insercion(lista)

print("Lista ordenada:", lista\_ordenada)

## Ordenamiento de Mezcla (Merge Sort)

El ordenamiento de mezcla es un algoritmo eficiente que utiliza la técnica de "divide y vencerás". Divide la lista en mitades, ordena cada mitad y luego las mezcla.

**Ejemplo de Ordenamiento de Mezcla en Python**

def mezclar(izquierda, derecha):

resultado = []

i = j = 0

while i < len(izquierda) and j < len(derecha):

if izquierda[i] < derecha[j]:

resultado.append(izquierda[i])

i += 1

else:

resultado.append(derecha[j])

j += 1

resultado.extend(izquierda[i:])

resultado.extend(derecha[j:])

return resultado

def merge\_sort(lista):

if len(lista) <= 1:

return lista

medio = len(lista) // 2

izquierda = merge\_sort(lista[:medio])

derecha = merge\_sort(lista[medio:])

return mezclar(izquierda, derecha)

*# Ejemplo de uso*

lista = [38, 27, 43, 3, 9, 82, 10]

lista\_ordenada = merge\_sort(lista)

print("Lista ordenada:", lista\_ordenada)

# **Conclusión**

La búsqueda y el ordenamiento de datos son tareas esenciales en la programación. En este informe, hemos explorado dos algoritmos de búsqueda (lineal y binaria) y tres algoritmos de ordenamiento (selección, inserción y mezcla). Cada uno tiene sus ventajas y desventajas, y la elección del algoritmo adecuado depende del contexto específico de la aplicación. Los ejemplos de programación en Python proporcionados demuestran cómo implementar estos algoritmos de manera sencilla y eficiente.

# **Referencias bibliógraficas**

* Downey, A (2020). *Pensar en Python Aprende a pensar como un informático*. Green Tea Press.
* Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.
* Python Software Foundation. (2025). *Python Programming Language*. [https://www.python.org](https://www.python.org/)

j