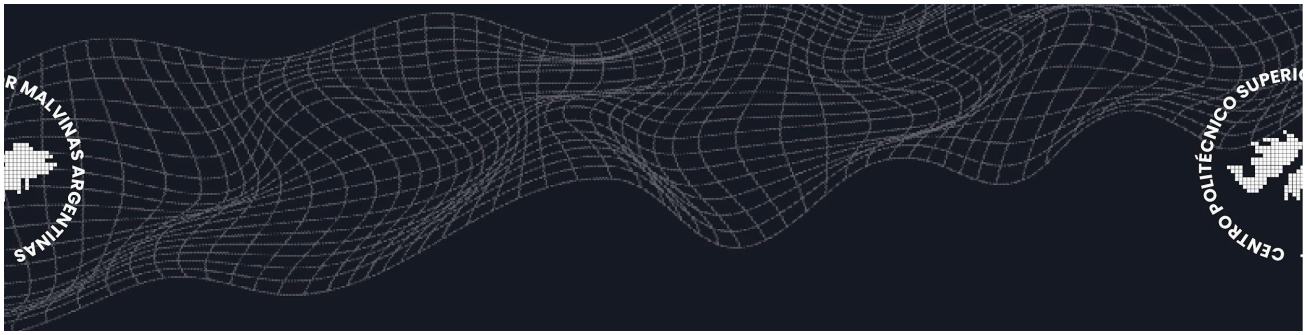


PROGRAMACIÓN 1

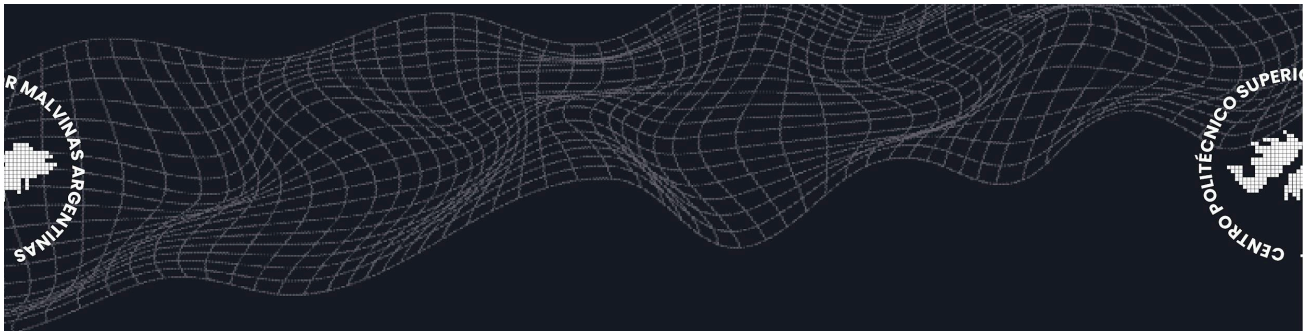
1º AÑO

CLASE Nº 8: ALGORITMOS DE BÚSQUEDA

Contenido

En la clase de hoy trabajaremos los siguientes temas:

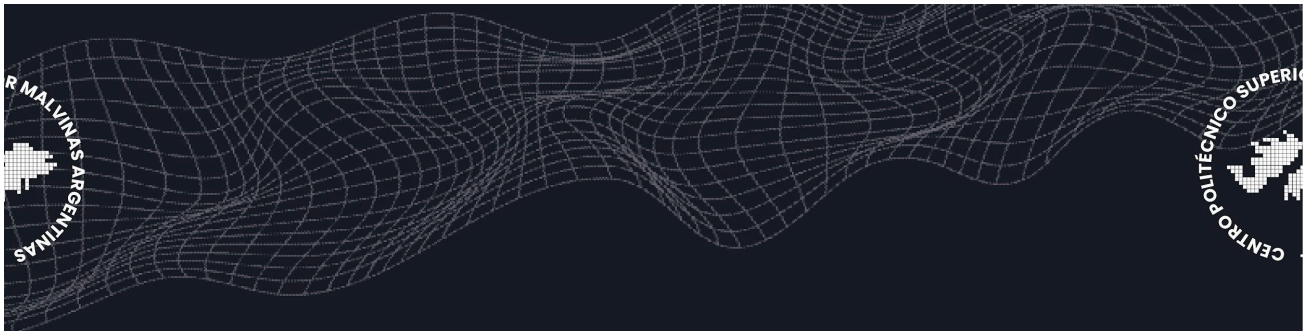
- Búsqueda Lineal
- Búsqueda Binaria
- Búsqueda de Salto
- Búsqueda Hash



1. Presentación

¡Hola, bienvenidos/as a la sexta clase de Programación. Esta clase ha sido pensada para los/as recién iniciados/as en el mundo de la **programación 1**.

En esta clase se aprenderá sobre los diferentes algoritmos de búsqueda nos ayuda a comprender cómo funcionan, cuándo y cómo aplicarlos de manera efectiva, y cómo seleccionar la mejor opción según el contexto y los requerimientos del problema. También nos enseña la importancia del ordenamiento de datos y el uso de estructuras de datos apropiadas para mejorar la eficiencia de las operaciones de búsqueda.



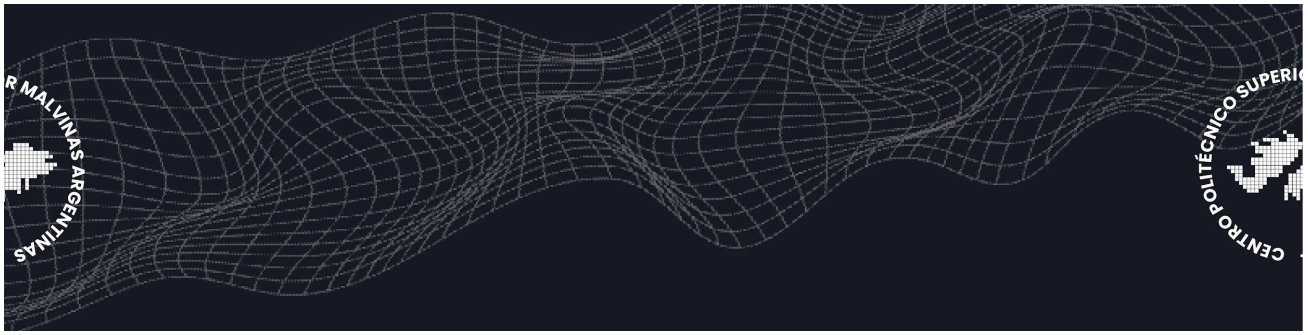
2. Desarrollo

Búsqueda Lineal

La búsqueda lineal, también conocida como búsqueda secuencial, consiste en recorrer secuencialmente una lista o array para encontrar un elemento deseado. Aquí tienes un ejemplo de implementación en Python:

```python
def busqueda_lineal(lista, elemento):
    for i in range(len(lista)):
        if lista[i] == elemento:
            return i  # Devuelve el índice donde se encontró el elemento
    return -1  # Si el elemento no se encuentra, devuelve -1

# Ejemplo de uso
numeros = [2, 5, 8, 10, 3, 1]
elemento_buscado = 10
indice = busqueda_lineal(numeros, elemento_buscado)
print(indice)  # Salida: 3
```

En este ejemplo, la función recibe una lista y un elemento a buscar. Itera sobre la lista y compara cada elemento con el elemento buscado. Si encuentra una coincidencia, devuelve el índice donde se encontró. Si no se encuentra el elemento, se devuelve -1.

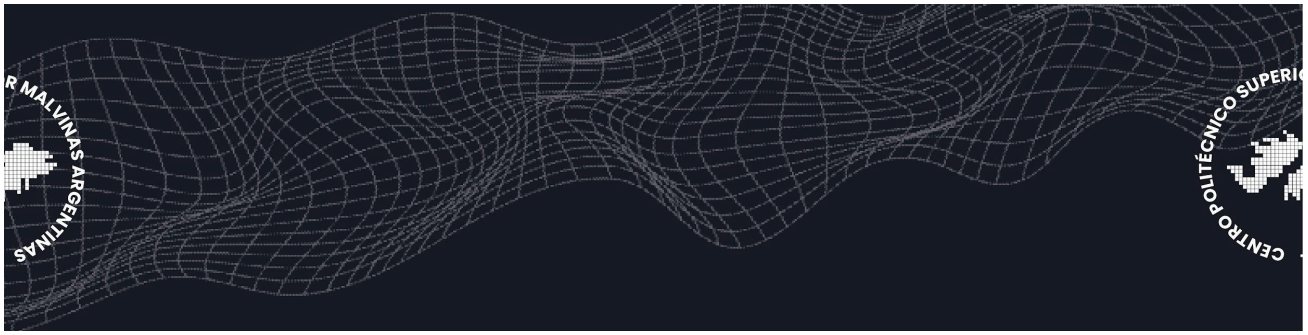


Búsqueda Binaria

La búsqueda binaria es un algoritmo eficiente para buscar en listas ordenadas. Divide repetidamente la lista a la mitad y compara el valor deseado con el elemento central. Aquí tienes un ejemplo de implementación en Python:

```python
def busqueda_binaria(lista, elemento):
    inicio = 0
    fin = len(lista) - 1

    while inicio <= fin:
        medio = (inicio + fin) // 2
        if lista[medio] == elemento:
            return medio
        elif lista[medio] < elemento:
            inicio = medio + 1
        else:
            fin = medio - 1
    return -1

# Ejemplo de uso
numeros = [1, 3, 5, 7, 9, 11, 13, 15]
elemento_buscado = 7
indice = busqueda_binaria(numeros, elemento_buscado)
print(indice)  # Salida: 3
```

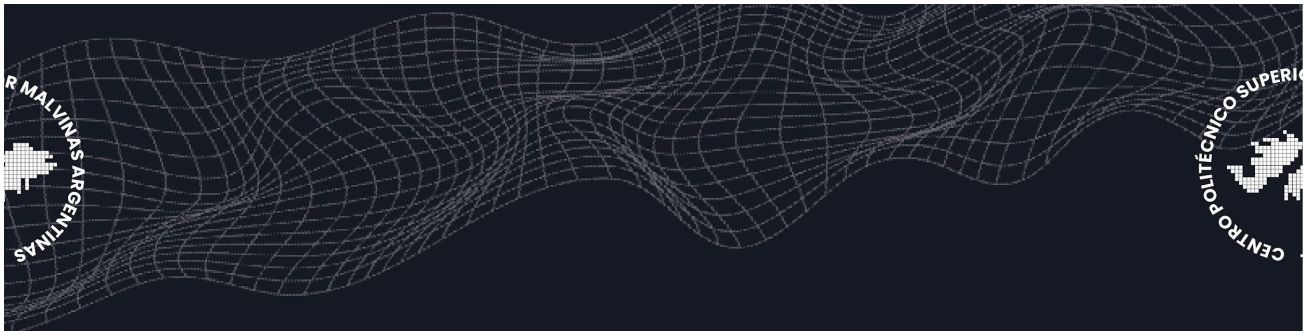
En este ejemplo, la función recibe una lista ordenada y un elemento a buscar. Utiliza una estrategia de dividir y conquistar para reducir el espacio de búsqueda a la mitad en cada iteración. Compara el elemento deseado con el elemento central y ajusta los límites según corresponda. Si encuentra una coincidencia, devuelve el índice correspondiente. Si no se encuentra el elemento, se devuelve -1.



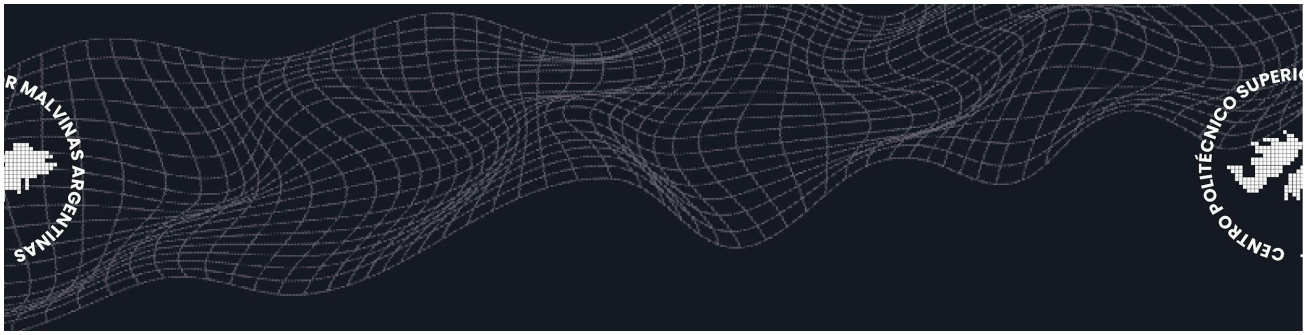
Búsqueda de Salto (Jump Search)

La búsqueda de salto es un algoritmo de búsqueda que aprovecha el hecho de que los datos están ordenados para reducir el número de comparaciones necesarias. En lugar de buscar elemento por elemento, salta hacia adelante en bloques de tamaño fijo y luego realiza una búsqueda lineal en el bloque para encontrar el elemento deseado. Aquí tienes un ejemplo de implementación en Python:

```python
import math

def busqueda_salto(lista, elemento):
    tamano_bloque = int(math.sqrt(len(lista)))
    bloque_actual = 0
    siguiente_bloque = tamano_bloque

    while lista[min(siguiente_bloque, len(lista)) - 1] < elemento:
        bloque_actual = siguiente_bloque
        siguiente_bloque += tamano_bloque
        if bloque_actual >= len(lista):
            return -1

    for i in range(bloque_actual, min(siguiente_bloque, len(lista))):
        if lista[i] == elemento:
            return i
    return -1

# Ejemplo de uso
numeros = [2, 5, 8, 10, 12, 15, 18, 20, 25, 30]
elemento_buscado = 15
indice = busqueda_salto(numeros, elemento_buscado)
print(indice)  # Salida: 5
```



En este ejemplo, la función recibe una lista ordenada y un elemento a buscar. Calcula el tamaño del bloque utilizando la raíz cuadrada del tamaño de la lista. Luego, avanza de bloque en bloque hasta encontrar un bloque que contenga un valor mayor o igual al elemento buscado. A continuación, realiza una búsqueda lineal dentro del bloque para encontrar el elemento deseado. Si se encuentra, se devuelve el índice correspondiente. Si no se encuentra el elemento, se devuelve -1.



Búsqueda Hash (Hash Search)

La búsqueda hash es un método de búsqueda que utiliza una función de hash para mapear claves a ubicaciones específicas en una estructura de datos conocida como tabla hash. La función de hash se utiliza para calcular un índice donde se espera que se encuentre el elemento buscado. Aquí tienes un ejemplo de implementación en Python utilizando un diccionario como tabla hash:

```python
def busqueda_hash(tabla_hash, clave):
    if clave in tabla_hash:
        return tabla_hash[clave]
    else:
        return None

# Ejemplo de uso
tabla = {
    "Juan": 25,
    "María": 30,
    "Pedro": 28,
    "Ana": 35
}
clave_busqueda = "Pedro"
resultado = busqueda_hash(tabla, clave_busqueda)
print(resultado)  # Salida: 28
```

En este ejemplo, la función recibe una tabla hash (en este caso, un diccionario en Python) y una clave a buscar. Utiliza la clave para acceder directamente al valor correspondiente en la tabla hash. Si la clave se encuentra en la tabla hash, se devuelve el valor asociado. Si no se encuentra la clave, se devuelve None o se puede utilizar otro valor predeterminado según sea necesario.

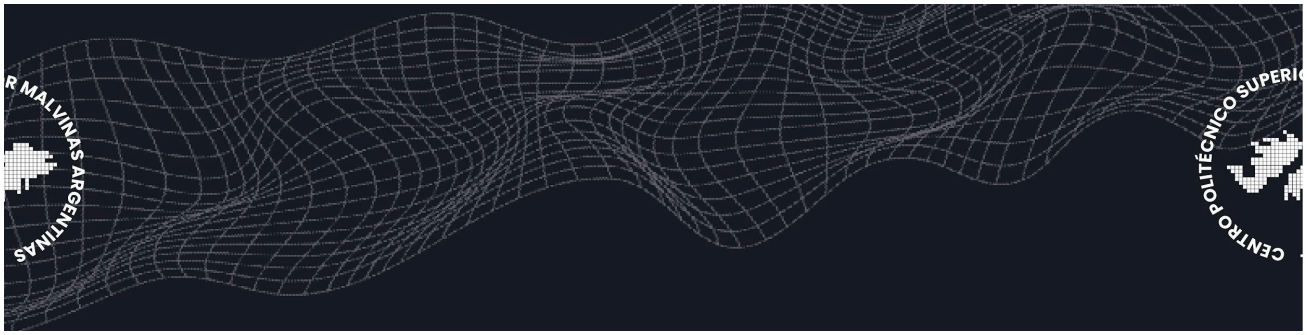
La búsqueda hash es eficiente en promedio, ya que evita la necesidad de recorrer una lista o estructura de datos completa. Sin embargo, su eficiencia depende de la calidad de la función de hash y la distribución de las claves en la tabla hash.

Actividad 1

a. Implementar la búsqueda lineal en una lista de números desordenados.
b. Aplicar la búsqueda binaria en una lista ordenada de palabras.

Actividad 2

a. Utilizar la búsqueda de salto para encontrar un número en una lista ordenada de manera eficiente.
b. Aplicar la búsqueda hash para encontrar el significado de una palabra en un diccionario.



3. Conclusión

Búsqueda Lineal

- Es sencillo de implementar y funciona en cualquier tipo de lista, sin necesidad de que esté ordenada.
- Sin embargo, su eficiencia es lineal y puede ser lenta en listas grandes.
- Es útil cuando no se tiene información adicional sobre la lista o cuando la lista es pequeña.

Búsqueda Binaria

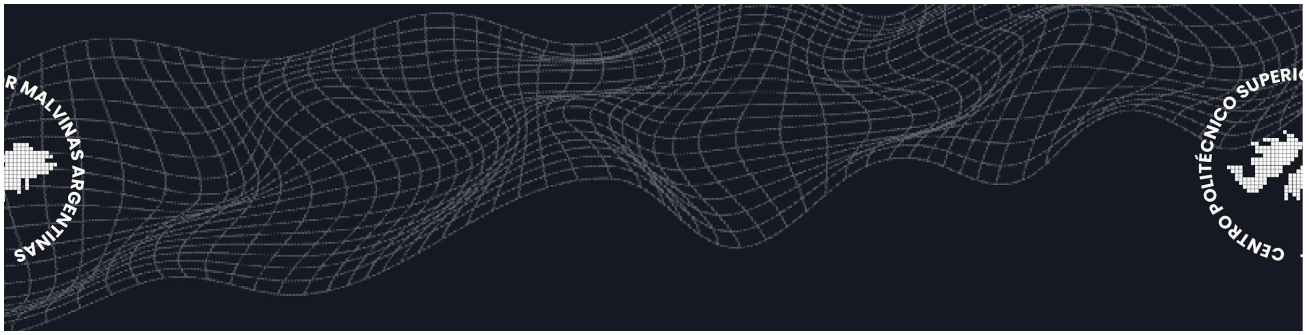
- Requiere que la lista esté ordenada, pero ofrece una eficiencia logarítmica, lo que significa que puede encontrar el elemento en menos iteraciones.
- Es más eficiente que la búsqueda lineal en listas grandes.
- Es especialmente útil cuando la lista está ordenada y se necesita una búsqueda rápida.

Búsqueda de Salto

- Combina elementos de la búsqueda lineal y la búsqueda binaria.
- Utiliza saltos en bloques de tamaño fijo para reducir el número de comparaciones necesarias.
- Es eficiente en listas ordenadas, pero no es tan eficiente como la búsqueda binaria.
- Puede ser útil cuando se tiene una idea aproximada de la ubicación del elemento buscado.

Búsqueda Hash

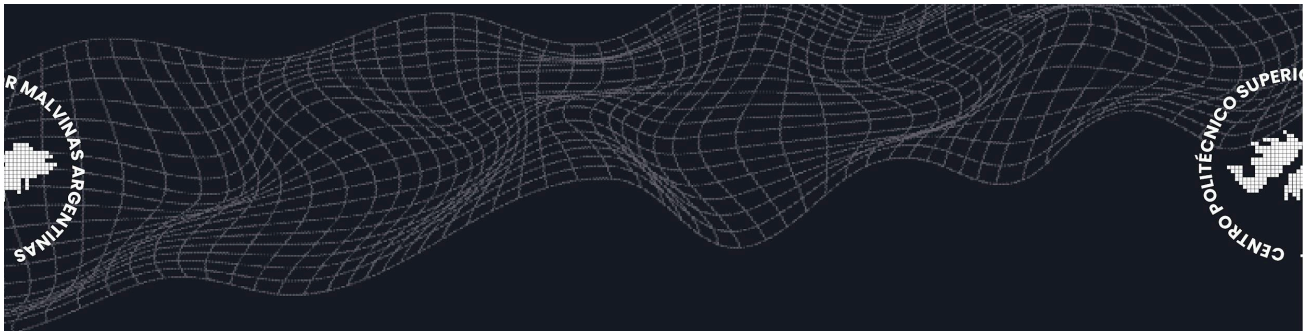
- Requiere una estructura de datos adicional, como una tabla hash, que puede requerir más memoria.
- Ofrece una búsqueda muy eficiente en promedio, ya que utiliza una función de hash para acceder directamente al elemento deseado.
- Es especialmente útil cuando se necesita una búsqueda rápida en grandes conjuntos de datos utilizando claves únicas.



En general, la elección del algoritmo de búsqueda depende del contexto y los requisitos específicos del problema.
Si se tiene una lista ordenada y se necesita una búsqueda eficiente, la búsqueda binaria es una buena opción. Si no se tiene información adicional sobre la lista o si la lista es pequeña, la búsqueda lineal puede ser suficiente.
La búsqueda de salto y la búsqueda hash son opciones intermedias que pueden ser útiles en determinados escenarios.

Es importante considerar el tamaño de los datos, la frecuencia de las búsquedas, la necesidad de ordenamiento y otros factores al seleccionar el algoritmo de búsqueda adecuado.

Recuerda que cada algoritmo tiene sus ventajas y desventajas, y es importante evaluar el compromiso entre eficiencia y complejidad para encontrar la mejor solución en cada caso.



Bibliografía Obligatoria:

- FUNDAMENTOS DE PROGRAMACIÓN. Algoritmos, estructura de datos y objetos-Cuarta edición - Luis Joyanes Aguilar- McGrawHill - 2008 - ISBN 978-84-481-6111-8

Bibliografía sugerida de la Unidad:

- "Introduction to Algorithms" by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein.
- "Algorithms, Part I" de Robert Sedgewick y Kevin Wayne.
- "Data Structures and Algorithms in Python" by Michael T. Goodrich, Roberto Tamassia, y Michael H. Goldwasser.
- "Algorithms, Part II" de Robert Sedgewick y Kevin Wayne.
- "Introduction to the Design and Analysis of Algorithms" by Anany Levitin.