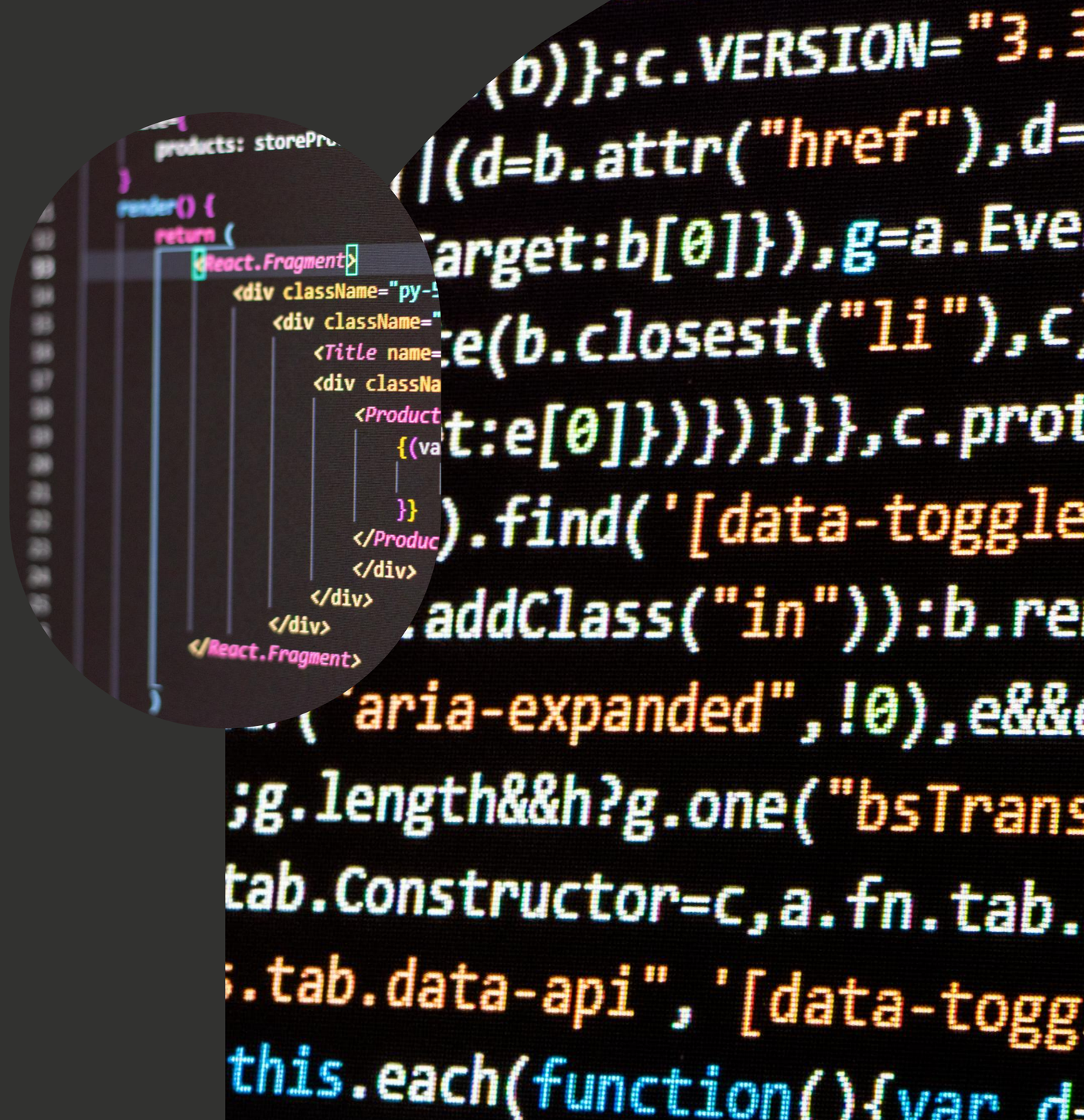
Búsqueda y ordenamiento en programación

Trabajo Integrador Programación I para la UTN (Universidad tecnológica Nacional.)

Alumnos:
Marcelo Oviedo
Federico Panella

Fecha: 9 de Junio del 2025



Objetivos de la investigación

1

Conocer algunos de los métodos más conocidos para realizar búsquedas y ordenación en algoritmos.

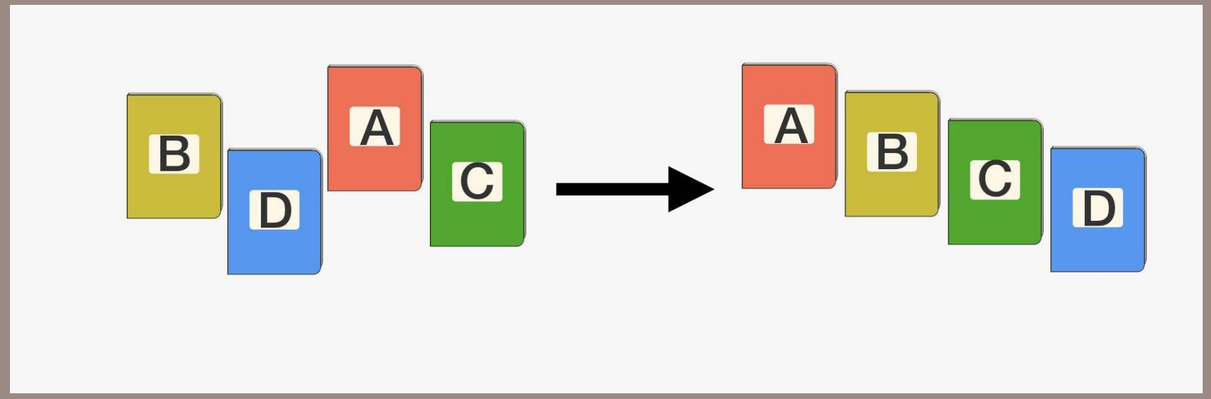
2

Poder desarrollar en Python estos códigos para ver resultados reales.

3

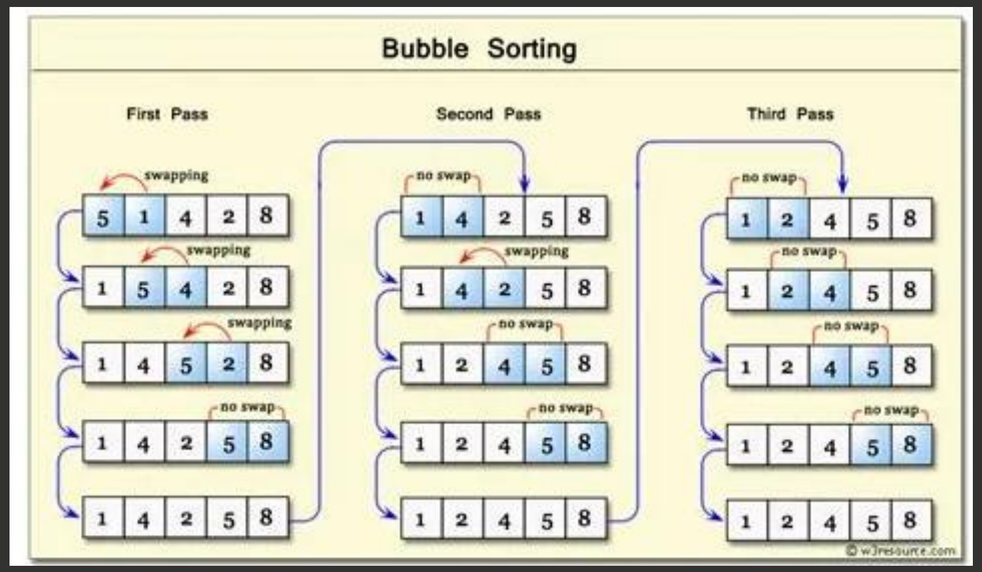
Identificar en cuanto a lo aprendido que método es más adecuado en cada ocasión.

Algoritmos de ordenamiento



Bubble Sort

```python
def bubble_sort(lista):
    """
    Función que ordena recibida la lista con el método burbuja y devuelve una lista ordenada.
    Comparando cada elemento con su adyacente e intercambiando si es necesario.
    Imprime en consola el tiempo que llevó el ordenamiento y los elementos ordenados.
    """
    n = len(lista) #Obtengo el largo de la lista
    inicio = time.time() #inicializo el tiempo
    for i in range(n):
        for j in range(0, n - i - 1):
            if lista[j] > lista[j + 1]:
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
    fin = time.time() #obtengo el tiempo al finalizar el ordenamiento
    tiempo_ms = (fin - inicio) * 1000  # Convertierto la diferencia a milisegundos
    print(f"Tiempo de ejecución bubble sort para {n} elementos: {tiempo_ms:.3f} ms")
    return lista
```

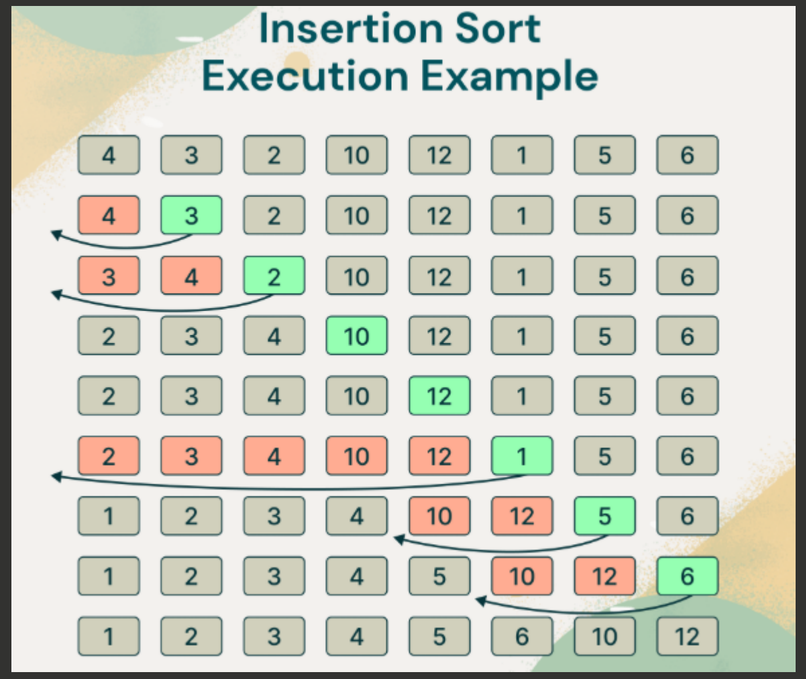


Detalles del algoritmo:

- Este algoritmo compara valores adyacentes y los intercambia si es necesario.
- Repite este procedimiento para todos los índices de la lista.
- Es fácil de implementar y comprender, pero poco eficiente para listas grandes.
- En el peor de los casos es de $O(n^2)$.

Insertion Sort

```python
def insertion_sort(lista):
    """
    Función que ordena la lista recibida con el método de insertion y devuelve una lista ordenada.
    Este método reconstruye la lista de forma ordenadade izquierda a derecha tomando
    1 elemento a la vez y lo colaca en la posición correcta.
    Imprime en consola el tiempo que llevó el ordenamiento y los elementos ordenados.
    """
    n = len(lista) #Obtengo el largo de la lista
    inicio = time.time() #inicializo el tiempo
    #Inicio el recorrido
    for i in range(1, n):
        actual = lista[i] #Agarro el último elemento no ordenado de la lista para la primera vuelta es el elemento [1] para compara con el [0]
        j = i - 1 #Este J equivale al último elemento ordenado de la lista
        while j >= 0 and lista[j] > actual: #Si no me salgo del array y si el elemento anterior ordenado es mayor empiezo a desplazarlo
            lista[j + 1] = lista[j]
            j -= 1
        lista[j + 1] = actual #una vez llegada a la posición ordenada correcta intercambio y avanzo al siguiente elemento no ordenado
    #Fin del recorrido
    fin = time.time() #obtengo el tiempo al finalizar el ordenamiento
    tiempo_ms = (fin - inicio) * 1000  # Convertierto la diferencia a milisegundos
    print(f"Tiempo de ejecución bubble sort para {n} elementos: {tiempo_ms:.3f} ms")
    return lista
```

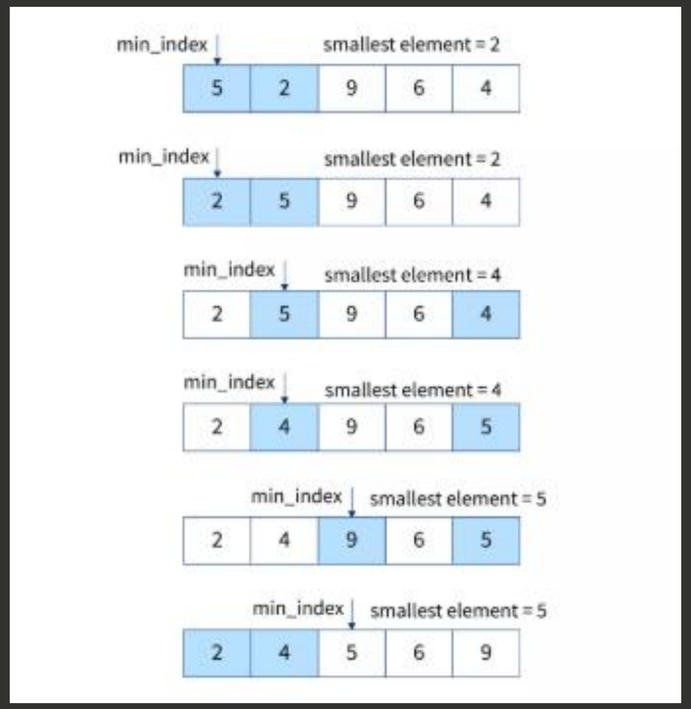


Detalles del algoritmo:

- Este algoritmo reconstruye la lista ordenada de izquierda a derecha.
- Toma 1 elemento a la vez y lo coloca en la posición correcta del tramo de la lista ya ordenado.
- Es más eficiente que bubble sort para listas pequeñas.
- Pero en el peor de los casos también es de $O(n^2)$.

Selection Sort

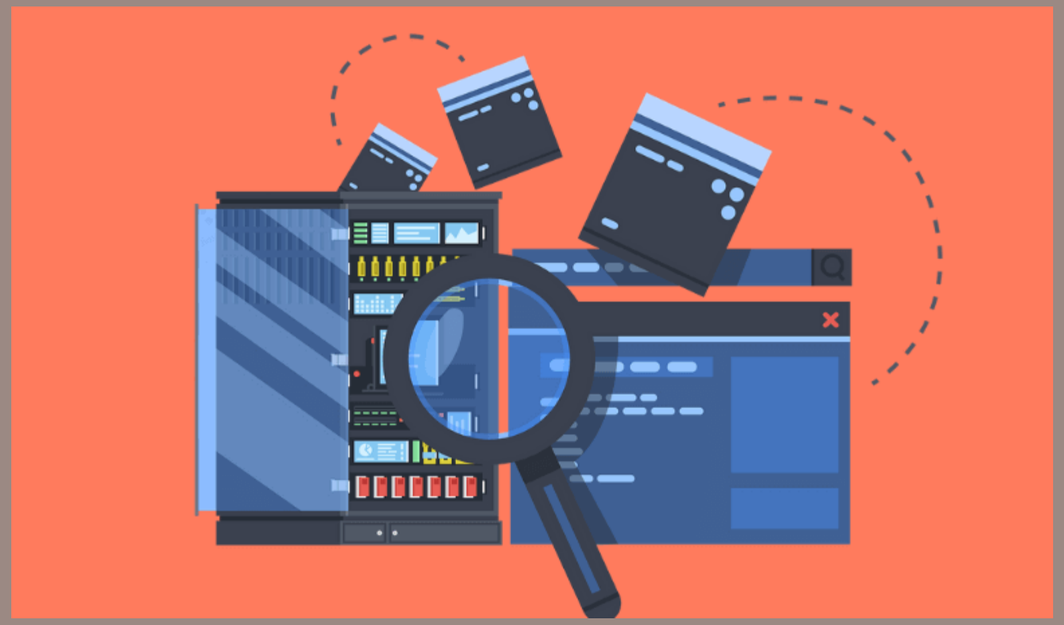
```python
def selection_sort(lista):
    """
    Función que ordena la lista recibida con el método de selección y devuelve una lista ordenada.
    Este método selecciona el elemento más pequeño de la lista y lo coloca en la posición que le corresponde.
    Imprime en consola el tiempo que llevó el ordenamiento y los elementos ordenados.s
    """
    n = len(lista)
    inicio = time.time() #inicializo el tiempo
    #Inicio el recorrido
    for i in range(n):  #Recorro todos los elementos del array
        min_index = i   #Index del elemento más pequeño
        for j in range(i + 1, n): #recorro del elemento más pequeño hasta el final
            if lista[j] < lista[min_index]:
                min_index = j #Guardo el index del elemento más pequeño encontrado en recorrido
        lista[i], lista[min_index] = lista[min_index], lista[i] #Al final algo el intercambi con el mínimo index encontrado en la vuelta
    #Fin del recorrido
    fin = time.time() #obtengo el tiempo al finalizar el ordenamiento
    tiempo_ms = (fin - inicio) * 1000  # Convertierto la diferencia a milisegundos
    print(f"Tiempo de ejecución bubble sort para {n} elementos: {tiempo_ms:.3f} ms")
    return lista
```



Detalles del algoritmo:

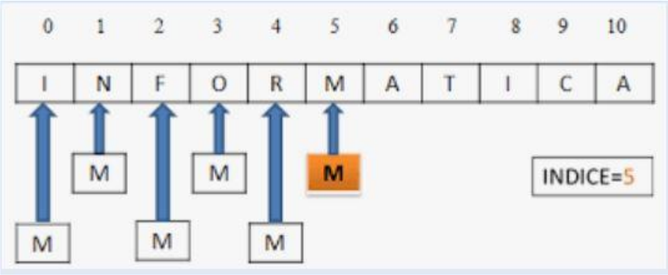
- En selection se selecciona en cada recorrido el elemento más pequeño de la lista.
- Coloca ese elemento en el index más pequeño actual.
- Es muy fácil de comprender.
- No es eficiente para listas largas, con complejidad $O(n^2)$ en el peor de los casos.

Algoritmos de búsqueda



Búsqueda lineal

```python
def busqueda_lineal(lista, objetivo):
    """Búsqueda lineal (O(n)) que devuelve (índice, tiempo)"""
    inicio = time.perf_counter()        You, 29 minutes ago • feat busqueda done …

    for i in range(len(lista)):
        if lista[i] == objetivo:
            fin = time.perf_counter()
            return i, fin - inicio

    fin = time.perf_counter()
    return -1, fin - inicio
```



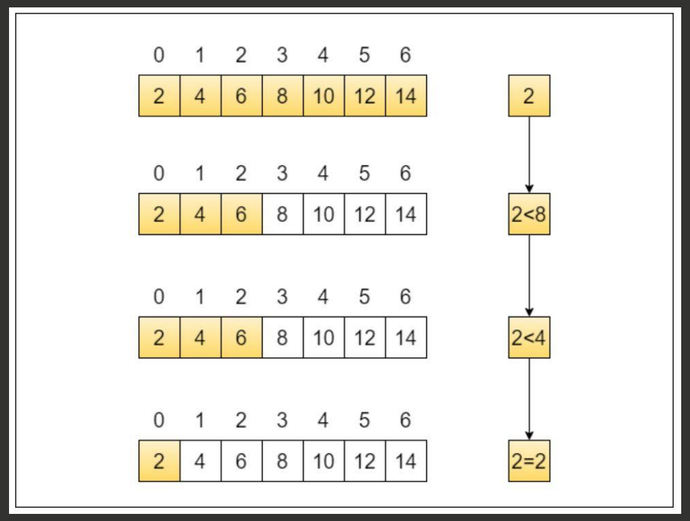
Detalles del algoritmo:

- Itera sobre todos los elementos de la lista comparando cada elemento con el objetivo hasta encontrarlo.

- Si lo encuentra devuelve el índice, caso contrario, generalmente devuelve "-1" que es un valor que no puede pertenecer a ningún índice.

Búsqueda binaria

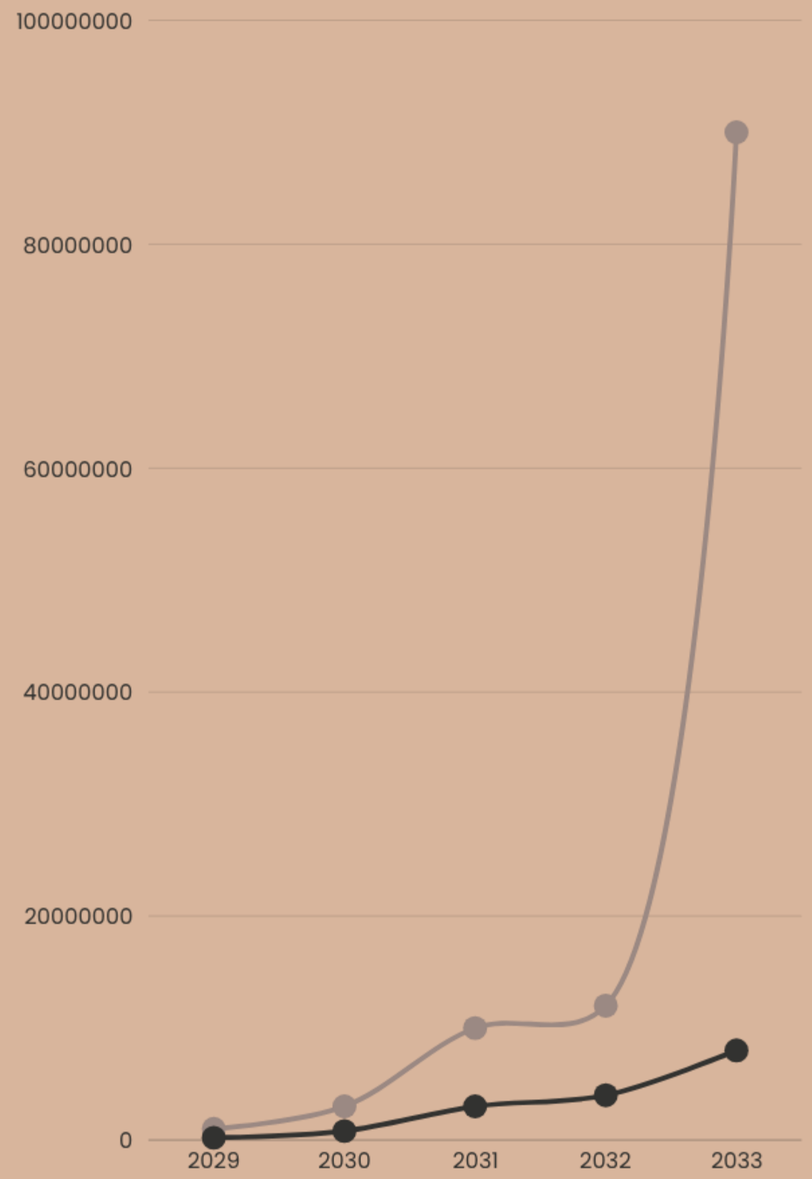
```python
def busqueda_binaria(lista, objetivo):
    """Búsqueda binaria (O(log n)) que devuelve (índice, tiempo)"""
    inicio = time.perf_counter()
    izquierda, derecha = 0, len(lista) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        valor_medio = lista[medio]

        if valor_medio == objetivo:
            fin = time.perf_counter()
            return medio, fin - inicio
        elif valor_medio < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1

    fin = time.perf_counter()
    return -1, fin - inicio
```



Detalles del algoritmo:

(Divide y vencerás)

- Calcula el índice medio de la lista.
- Compara el elemento en esa posición con el objetivo:
- Si son iguales: Retorna el índice (¡éxito!).
- Si el objetivo es mayor: Repite la búsqueda en la mitad superior.
- Si el objetivo es menor: Repite la búsqueda en la mitad inferior.



¿Por qué es importante aprender esto?

Conocimiento

Es importante conocer las alternativas para búsqueda y ordenamiento, ya que son esencial a la hora de programar y manejar datos.

Eficiencia

Conocerlo hará que podamos tomar mejores decisiones según el contexto, para así poder hacer nuestros programas más eficientes y óptimos en cuento a tiempo de ejecución.

¡Muchas gracias!