TRABAJO INTEGRADOR

ALGORITMOS DE BÚSQUEDA Y ORDENAMIENTO

ALUMNOS

JUAN SILVA - FABIAN SALAS

PROGRAMACIÓN I

Profesor: Nicolás Quirós

Fecha de entrega: 09/06/2025

ÍNDICE

Introducción:

En el desarrollo de software y el análisis de datos, la eficiencia con la que se procesan y organizan los datos es fundamental. Los algoritmos de búsqueda y ordenamiento son herramientas clave en este contexto, ya que permiten localizar información específica dentro de una estructura de datos y organizarla de manera que su acceso sea más rápido y eficiente. Estos algoritmos no solo tienen aplicaciones teóricas en la informática, sino que también son esenciales en tareas cotidianas como la gestión de bases de datos, la visualización de información y la optimización de procesos computacionales.

A través de este trabajo, se explorarán los principales algoritmos de búsqueda, como la búsqueda lineal y la búsqueda binaria, como así también el de ordenamiento por burbuja e inserción. Donde se realizaron análisis de sus principios de funcionamiento, su implementación en Python y su eficiencia en distintos escenarios.

Marco teórico

Ordenamiento

Un algoritmo de ordenamiento es un conjunto de pasos o instrucciones bien definidos que se utilizan para organizar un conjunto de datos en una secuencia específica. El objetivo principal es disponer los elementos de una colección (como números, palabras o registros) en un orden determinado, que puede ser ascendente, descendente, alfabético o según algún otro criterio definido. Esto se hace para facilitar tareas posteriores como la búsqueda, el análisis o la presentación de la información.

¿Qué es ordenar?

El ordenamiento de datos es una operación fundamental en la informática, ya que permite organizar la información de manera que su procesamiento sea más eficiente y comprensible. Al ordenar una colección de elementos, como números, palabras o registros, se facilita la ejecución de diversas tareas, tales como la búsqueda, la comparación, la visualización o el análisis de datos.

Por ejemplo, muchos algoritmos de búsqueda, como la búsqueda binaria, requieren que los datos estén ordenados previamente para funcionar correctamente. Además, en bases de datos o sistemas de información, mostrar los resultados de forma ordenada mejora la experiencia del usuario y permite encontrar patrones o detectar anomalías con mayor facilidad.

¿Por qué ordenamos?

En términos de eficiencia, trabajar con datos ordenados puede reducir significativamente el tiempo de ejecución de ciertos algoritmos, lo que es especialmente importante cuando se manejan grandes volúmenes de información. Por estas razones, aprender y aplicar algoritmos de ordenamiento es esencial en el diseño de soluciones informáticas eficaces

Tipo de ordenamientos

Burbuja

El **algoritmo de ordenamiento burbuja** compara elementos adyacentes de una lista y los intercambia si están en el orden incorrecto. Repite este proceso varias veces, haciendo que los elementos “más pesados” (mayores) vayan subiendo hacia el final, como burbujas en el agua.

Funcionamiento paso a paso:

1. Comparar el primer y segundo elemento.
2. Si están en el orden incorrecto, intercambiarlos.
3. Repetir con el segundo y tercer elemento, y así hasta el final.
4. Volver al inicio y repetir el proceso hasta que no haya más intercambios.

Complejidad temporal:

Mejor caso

- **Situación**: La lista ya está ordenada.
- **Número de comparaciones**: $n-1$ por pasada (con detección de que no hubo intercambios).
- **Número de intercambios**: 0.
- **Complejidad temporal**: $O(n)$ (con optimización que detecta si hubo intercambios).
- **Ejemplo**: `[1, 2, 3, 4, 5]`

En este caso, solo se necesita una pasada para confirmar que los elementos están en orden, si se usa una bandera para detectar si se realizaron intercambios.

Peor caso

- **Situación**: La lista está en orden inverso.
- **Número de comparaciones**: $n(n-1)/2$
- **Número de intercambios**: Máximo posible, también $n(n-1)/2$
- **Complejidad temporal**: $O(n^2)$
- **Ejemplo**: `[5, 4, 3, 2, 1]`

En este caso, cada elemento tiene que "burbujear" hasta su posición correcta, lo cual implica muchas comparaciones e intercambios.

Selección

El **algoritmo de ordenamiento por selección** recorre la lista buscando el elemento más pequeño y lo intercambia con el primer elemento. Luego repite el proceso con el resto de la lista (excluyendo los elementos ya ordenados), colocando el siguiente menor en la segunda posición, y así sucesivamente hasta que toda la lista esté ordenada.

Funcionamiento paso a paso:

1. Buscar el valor mínimo en la lista.
2. Intercambiarlo con el primer elemento.
3. Buscar el siguiente mínimo en la sublista restante.
4. Repetir hasta ordenar toda la lista.

Complejidad temporal:

Método de Selección

Mejor caso

- **Situación**: La lista ya está ordenada ascendentemente.
  Ejemplo: `[1, 2, 3, 4, 5]`

- **Comparaciones**:
  Siempre hacen $n(n-1)/2$ comparaciones, sin importar el orden.

  Ejemplo para n = 5 → 10 comparaciones.

- **Intercambios**:
  Aunque los elementos ya estén ordenados, se suelen realizar $n-1$ intercambios, uno por cada iteración (aunque algunos pueden ser innecesarios si se implementa una verificación).

- **Complejidad temporal**:
  $O(n^2)$ (No mejora en este caso porque no se evita ninguna comparación)

A diferencia de bubble sort, **selection sort no se optimiza** cuando los datos ya están ordenados.

Peor caso

- **Situación**: La lista está en orden completamente inverso.

  Ejemplo: `[5, 4, 3, 2, 1]`

- **Comparaciones**:
  También hace $n(n-1)/2$ comparaciones.
  Igual que en el mejor caso.

- **Intercambios**:
  Se realizan exactamente $n-1$ intercambios, moviendo el mínimo al inicio en cada iteración.

- **Complejidad temporal**:
  $O(n^2)$

El rendimiento es igual que en el mejor caso en términos de comparaciones, pero en este caso, **los intercambios son realmente necesarios**.

Comparación entre ambos Métodos

Criterio	Bubble Sort	Selection Sort
Funcionamiento básico	Compara e intercambia elementos adyacentes	Busca el mínimo y lo coloca en su posición correcta
Complejidad - Mejor caso	$O(n)$ (si se optimiza)	$O(n^2)$
Complejidad - Peor caso	$O(n^2)$	$O(n^2)$
Comparaciones	Hasta $\frac{n(n-1)}{2}$	Siempre $\frac{n(n-1)}{2}$
Intercambios	Muchos (puede hacer uno por cada comparación)	Pocos (a lo sumo $n-1$)
Eficiencia práctica	Lento y no escalable, pero puede detenerse antes	Más predecible, pero sin mejora en caso favorable
Uso común	Fines didácticos y listas muy pequeñas	Fines didácticos; no se usa en producción
Estabilidad	Sí (no cambia el orden relativo de elementos iguales)	No (puede cambiar el orden relativo)
Código más simple	Sí	Ligeramente más complejo

Ejemplo práctico

```
import time
import random

def bubble_sort(lista):
    """Ordena la lista usando el algoritmo burbuja."""
    n = len(lista)
    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]
    return lista

def selection_sort(lista):
    """Ordena la lista usando el algoritmo de selección."""
    n = len(lista)

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

```
            if lista[j] < lista[min_index]:
                min_index = j
            lista[i], lista[min_index] = lista[min_index], lista[i]
    return lista

# ------------------------
# Entrada del usuario
# ------------------------
entrada = input("Ingrese una lista de números separados por comas (ej: 5,2,9,1,4): ")
lista_usuario = [int(x.strip()) for x in entrada.split(',')]

# ------------------------
# Ejecutar ambos algoritmos
# ------------------------

# Burbuja
inicio_burbuja = time.time()
resultado_burbuja = bubble_sort(lista_usuario.copy())
fin_burbuja = time.time()

# Selección
inicio_seleccion = time.time()
resultado_seleccion = selection_sort(lista_usuario.copy())
fin_seleccion = time.time()

# ------------------------
# Mostrar resultados
# ------------------------
print("\n--- Resultados del ordenamiento ---")
print(f"Original:          {lista_usuario}")
print(f"Burbuja:           {resultado_burbuja} (Tiempo: {fin_burbuja - inicio_burbuja:.6f}
seg)")
print(f"Selección:         {resultado_seleccion} (Tiempo: {fin_seleccion - inicio_seleccion:.6f}
seg)")

# ------------------------
# Comparación con listas grandes
# ------------------------
print("\n--- Comparación de rendimiento con listas aleatorias ---")

lista_pequena = random.sample(range(100), 20)
lista_grande = random.sample(range(100000), 5000)

# Burbuja pequeña
inicio = time.time()
bubble_sort(lista_pequena.copy())
fin = time.time()

print(f"Burbuja - pequeña:    {fin - inicio:.6f} seg")

# Burbuja grande
inicio = time.time()
bubble_sort(lista_grande.copy())
fin = time.time()
print(f"Burbuja - grande:     {fin - inicio:.6f} seg")

# Selección pequeña
inicio = time.time()
selection_sort(lista_pequena.copy())
fin = time.time()
print(f"Selección - pequeña: {fin - inicio:.6f} seg")

# Selección grande
```

```
inicio = time.time()
selection_sort(lista_grande.copy())
fin = time.time()
print(f"Selección - grande:  {fin - inicio:.6f} seg")
```

Búsqueda:

La búsqueda en programación es el proceso de encontrar uno o más elementos específicos dentro de un conjunto de datos. Su objetivo es determinar si un elemento particular está presente y, si lo está, su ubicación dentro del conjunto.

Existen diferentes algoritmos de búsqueda, como:

- Búsqueda lineal (o secuencial): Recorre el conjunto de datos elemento por elemento hasta encontrar el deseado o llegar al final.

- Búsqueda binaria: Sólo funciona en conjuntos de datos previamente ordenados. Divide repetidamente el conjunto a la mitad hasta encontrar el elemento. Es mucho más eficiente que la búsqueda lineal para grandes volúmenes de datos.

Búsqueda Lineal (o Secuencial)

Es el algoritmo de búsqueda más simple, que recorre cada elemento del conjunto de datos de forma secuencial hasta encontrar el elemento deseado. Es fácil de implementar, pero puede ser lento para conjuntos de datos grandes.

- **Descripción:** Recorre todos los elementos de una lista hasta encontrar el valor buscado.

- **Complejidad:** `O(n)`

- **Ventajas:** Simple, funciona en listas no ordenadas.

- **Desventajas:** Lenta en listas grandes.

Búsqueda Binaria

Es un algoritmo de búsqueda eficiente que funciona en conjuntos de datos ordenados. Divide el conjunto de datos en dos mitades y busca el elemento deseado en la mitad correspondiente. Repite este proceso hasta encontrar el elemento o determinar que no está en el conjunto de datos.

- **Descripción:** Divide la lista ordenada en mitades y descarta la mitad donde no puede estar el valor.
- **Requisito:** La lista debe estar ordenada.
- **Complejidad:** `O(log n)`
- **Ventajas:** Muy rápido en listas grandes.
- **Desventajas:** Requiere orden previo.

Ejemplo práctico

```
import time
import random

def búsqueda binaria(lista, objetivo):
    izquierda = 0
    derecha = len(lista) - 1

    while izquierda <= derecha:
        medio = (izquierda + derecha) // 2
        if lista[medio] == objetivo:
            return medio
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1

    return -1

def búsqueda lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1


# Ingreso de datos
entrada = input("Ingrese una lista de números separados por comas (ej: 5,2,9,1,4): ")
lista = [int(x.strip()) for x in entrada.split(',')]

objetivo = int(input("Ingrese el número que desea buscar: "))
```

```
# Búsqueda Lineal
indice_lineal = búsqueda_lineal(lista, objetivo)
if indice_lineal != -1:
    printf"[Lineal] El número {objetivo} se encuentra en la posición {indice_lineal}")
else:
    printf"[Lineal] El número {objetivo} no se encuentra en la lista.")


# Búsqueda Binaria (requiere lista ordenada)
lista_ordenada = sorted(lista)
indice_binaria = búsqueda_binaria(lista_ordenada, objetivo)

if indice_binaria != -1:
    printf"[Binaria] El número {objetivo} se encuentra en la posición {indice_binaria} (en
la lista ordenada)")
else:
    printf"[Binaria] El número {objetivo} no se encuentra en la lista ordenada.")

# Medir tiempo de búsqueda lineal
inicio_lineal = time.time()
resultado_lineal = búsqueda_lineal(lista, objetivo)
fin_linear = time.time()
tiempo_lineal = fin_ lineal - inicio_lineal

lista_ordenada = sorted(lista)  # Lista ordenada para búsqueda binaria
# Medir tiempo de búsqueda binaria
inicio_binaria = time.time()
resultado_binaria = búsqueda_binaria(lista_ordenada, objetivo)
fin_binaria = time.time()
tiempo_binaria = fin_binaria - inicio_binaria

# Resultados
printf"Búsqueda Lineal: Resultado = {resultado_lineal}, Tiempo = {tiempo_lineal:.6f}
segundos")

printf"Búsqueda Binaria: Resultado = {resultado_binaria}, Tiempo = {tiempo_binaria:.6f}
segundos")
```

CONCLUSIÓN

Los algoritmos de ordenamiento y búsqueda son fundamentales en el campo de la informática, ya que permiten organizar y localizar datos de manera eficiente. El método de la burbuja y el de selección, aunque sencillos de implementar y útiles para fines educativos, resultan poco eficientes en grandes volúmenes de datos debido a su complejidad temporal $O(n^2)$. En contraste, las técnicas de búsqueda, como la búsqueda lineal y la binaria, ofrecen alternativas con distintos niveles de eficiencia: la búsqueda lineal es simple pero poco óptima en listas grandes, mientras que la búsqueda binaria, mucho más rápida ($O(\log n)$), requiere que los datos estén previamente ordenados. Comprender las características, ventajas y limitaciones de cada uno de estos algoritmos es esencial para seleccionar la solución más adecuada según el contexto y la naturaleza del problema a resolver.

Bibliografía

- Dr. Arno Formella Departamento de Informática Universidad de Vigo https://formella.webs.uvigo.es/doc/porto.pdf

- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms.

-Python a fondo “Oscar Ramirez Jiménez”

-https://programacion.net/