Trabajo Práctico INTEGRADOR - PROGRAMACIÓN I

Búsqueda y Ordenamiento

Alumnos

Rodrigo Aguirre, Fernando Aguillón

Tecnicatura Universitaria en Programación - Universidad Tecnológica Nacional.

PROGRAMACIÓN I

Docente Titular

Nicolas Quirós

Docente Tutor

Florencia Gubiotti

09 de Junio de 2025

UNIVERSIDAD TECNOLOGICA NACIONAL

Introducción:

En este trabajo práctico se aborda el tema “Algoritmos de búsqueda y ordenamiento”, ambos siendo herramientas fundamentales en el ámbito de la programación. La elección se basó en su amplio uso en aplicaciones, inteligencia artificial, bases de datos, motores de búsqueda y análisis de datos entre otros. Entender estos algoritmos ayuda al programador a crear código eficiente y óptimo.

El foco principal de la búsqueda y el ordenamiento se centra en torno a la organización y recuperación de datos. Como ejemplo se puede ver a la hora de elegir un método de ordenamiento reduce significativamente el tiempo de ejecución de un programa, así como lo hace una búsqueda eficiente en un volumen grande de datos.

El objetivo principal de este trabajo es:

- Reproducir diferentes algoritmos de búsqueda y ordenamiento utilizados en Python.
- Afirmar los conocimientos teóricos a través de la aplicación práctica.
- Comparar los diferentes tipos de algoritmos en escenarios y conjuntos diferentes.

Marco Teórico

¿Qué es un algoritmo?

Un algoritmo es una secuencia finita y bien definida de pasos o instrucciones que permiten resolver un problema o realizar una tarea específica. Deben cumplir ciertas propiedades como finitez, precisión y efectividad. En el ámbito de la informática, los algoritmos son fundamentales para el procesamiento de datos y la toma de decisiones automatizadas.

Los algoritmos se pueden clasificar según su propósito o estrategia. Dos de los más relevantes en estructuras de datos son:

1. Algoritmos de Ordenamiento:
   Permiten reorganizar los elementos de una lista en un orden específico (ascendente o descendente). Son esenciales para mejorar la eficiencia de búsquedas y análisis de datos.
   - Ejemplos: Bubble Sort, Selection Sort, Insertion Sort, Merge Sort, QuickSort.
2. Algoritmos de Búsqueda:
   Se utilizan para localizar un valor dentro de una estructura de datos, como listas o árboles.
   - Ejemplos: Búsqueda Lineal, Búsqueda Binaria.

También pueden clasificarse por su estrategia de resolución:

1. Algoritmos Iterativos: usan bucles repetitivos (como for o while).
2. Algoritmos Recursivos: se llaman a sí mismos para resolver subproblemas más simples.

Algoritmos de Ordenamiento

Bubble Sort: compara elementos adyacentes y los intercambia si están en el orden incorrecto, repetidamente
Selection Sort: se basa en encontrar el elemento más pequeño (o más grande) en cada iteración y luego intercambiarlo con el elemento en la posición actual.
Insertion Sort: tomar un elemento y moverlo a su posición correcta en la sublista ordenada, repitiendo para el resto de la lista.
Quick Sort: selecciona un "pivote" y divide la lista en dos sublistas con los elementos menores y mayores, aplicando recursivamente este metodo a las sublistas.
Merge Sort: divide la lista en sublistas de elementos iguales hasta obtener listas de 1 elemento, las ordena y las fusiona en una lista ordenada.

Algoritmos de Búsqueda

Búsqueda Lineal (Secuencial): Recorre todos los elementos uno a uno hasta encontrar el buscado o agotar la lista.
Búsqueda Binaria: Requiere que los datos estén ordenados. Divide el conjunto en dos y descarta la mitad que no puede contener el elemento buscado, repitiendo el proceso.

¿Cómo se comparan entre sí?

La eficiencia de un algoritmo se mide principalmente en dos aspectos:

La complejidad temporal, que mide cuánto tiempo tarda un algoritmo en completarse según el tamaño de entrada N.

- Se expresa con la notación Big O (O-grande).
- Ejemplos:
  - O(n) → tiempo crece linealmente con la cantidad de datos.
  - O(n^2) → tiempo crece cuadráticamente (mucho más lento).
  - O(log n) → muy eficiente, crece lentamente incluso si los datos aumentan.

Vale aclarar que no se mide el tiempo real (en segundos), sino el número de operaciones básicas que realiza el algoritmo en función del tamaño de entrada.

Por otro lado, la complejidad espacial, que mide cuánta memoria adicional necesita el algoritmo para funcionar, también en función de la entrada N.

- También se expresa con notación Big O:
  - O(1) → uso constante de memoria, sin importar el tamaño de la entrada.
  - O(n) → necesita espacio proporcional a la cantidad de datos.
  - O(n log n) o más → algoritmos que generan estructuras auxiliares grandes.

Se suelen analizar tres casos típicos:

Caso	Descripción
Mejor caso	Condiciones ideales (ej. lista ya ordenada).
Peor caso	Condiciones más desfavorables (ej. lista invertida).
Caso promedio	Lo que ocurre en la mayoría de las situaciones.

Podemos resumir el rendimiento de los algoritmos analizados para el trabajo de la siguiente manera:

	MEJOR	PEOR	PROMEDIO	ESPACIO	USO RECOMENDADO
BURBUJA	O(n)	O(n^2)	O(n^2)	O(1)	LISTAS PEQUEÑAS O CASI ORDENADAS
SELECCIÓN	O(n^2)	O(n^2)	O(n^2)	O(1)	LISTAS PEQUEÑAS
INSERCION	O(n)	O(n^2)	O(n^2)	O(1)	LISTA PEQUEÑAS O CASI ORDENADAS
QUICKORT	O(n log n)	O(n^2)	O(n log n)	O(log n)	LISTAS GRANDES (EVITAR PEOR CASO)
MERGE SORT	O(n log n)	O(n log n)	O(n log n)	O(n)	LISTAS GRANDES.
BÚSQUEDA LINEAL	O(1)	O(n)	O(n)	O(1)	LISTAS PEQUEÑAS O NO ORENADAS.
BÚSQUEDA BINARIA	O(1)	O(log n)	O(log n)	O(1)	LISTAS GRANDES ORDENADAS.

¿Por qué son importantes?

Los algoritmos de búsqueda y ordenamiento son fundamentales porque son la base para manejar datos de manera eficiente.

Ventajas de los algoritmos de ordenamiento
Organizar datos: Ordenar datos facilita muchas tareas, como buscar, comparar o eliminar duplicados.

Eficiencia: Muchos algoritmos y estructuras de datos funcionan mejor o solo funcionan si los datos están ordenados (ejemplo: búsqueda binaria, árboles balanceados).

Optimización: Un buen algoritmo de ordenamiento reduce el tiempo de procesamiento, haciendo que programas y sistemas sean más rápidos y escalables.

Fundamento para algoritmos complejos: Ordenar es un paso clave en muchos algoritmos avanzados, como la compresión de datos o algoritmos de grafos.

Ventajas de los algoritmos de búsqueda
Acceso rápido a la información: La búsqueda permite encontrar datos específicos en grandes conjuntos sin revisar todo uno a uno.

Eficiencia: Usar algoritmos de búsqueda adecuados disminuye el tiempo y recursos necesarios para encontrar un dato.

Base para operaciones más complejas: Muchas operaciones, como filtros, joins en bases de datos o validaciones, dependen de búsquedas rápidas y eficientes.

¿En donde podemos encontrar algoritmos de búsqueda y ordenamiento?

1. Estructuras de Datos
2. Bases de Datos
3. Sistemas Operativos
4. Desarrollo Web y Software
5. Videojuegos e Inteligencia Artificial
6. Ciencia de Datos y Machine Learning
7. Compiladores y Lenguajes de Programación

Entre otros.

Caso Práctico

Para el caso práctico, pensamos en desarrollar una aplicación con Python orientada a un tipo de almacén o deposito de productos de computación. Se buscar lograr poder cargar en un diccionario artículos únicos con una id alfanumérica del estilo “A000” para el primer elemento por ejemplo, poder ordenar dichos artículos y realizar buscas rápidas aplicando los diferentes algoritmos vistos. Estas funcionalidades se modularizaron en diferentes funciones para mayor comprensión y reutilización del código.

En una segunda etapa se testean algunos de los algoritmos desarrollados en el trabajo con sets de datos ficticios de distintas dimensiones para comparar el rendimiento de cada uno.

Comenzando con la aplicación, se creo una función para mostrar un menú al usuario donde podrá acceder a distintas funciones como agregar, eliminar y leer la lista de productos.

```python
def menu(productos):
    while True:
        print("\n--- Menú de Productos ---")
        print("1. Agregar producto")
        print("2. Eliminar producto")
        print("3. Imprimir lista de productos")
        print("4. Buscar producto")
        print("5. Salir")
        opcion = input("Seleccione una opción: ")

        if opcion == "1":
            agregar_producto(productos)
        elif opcion == "2":
            eliminar_producto(productos)
        elif opcion == "3":
            imprimir_productos(productos)
        elif opcion == "4":
            buscar_producto(productos)
        elif opcion == "5":
            print("Saliendo del programa.")
            break
        else:
            print("Opción no válida. Intente de nuevo.")
```

Si se desea agregar un producto se llamará a la función “agregar_producto” que pedirá un código alfanumérico el cual será validado con regex, y un nombre para el artículo. Luego se ordenarán los datos utilizando la función “merge_sort”, que como su nombre lo indica utiliza el algoritmo de ordenamiento Merge Sort.

```python
# Funcion para agregar producto
def agregar_producto(productos):
    while True:
        codigo = input("Ingrese el código del producto (formato: 1 letra + 3 números, ej: A123): ").upper()
        if re.fullmatch(r"[A-Z][0-9]{3}", codigo):
            break
        else:
            print("Código inválido. Debe ser una letra seguida de 3 números (ej: B456). Intente de nuevo.")

    nombre = input("Ingrese el nombre del producto: ")
    productos.append({"codigo": codigo, "nombre": nombre.title()})
    merge_sort(productos)
    print("Producto agregado correctamente.")
```

```python
def merge_sort(products):
    if len(products) > 1:
        mid = len(products) // 2
        left = products[:mid]
        right = products[mid:]

        merge_sort(left)
        merge_sort(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
            if left[i]['codigo'] < right[j]['codigo']:
                products[k] = left[i]
                i += 1
            else:
                products[k] = right[j]
                j += 1
            k += 1

        while i < len(left):
            products[k] = left[i]
            i += 1
            k += 1

        while j < len(right):
            products[k] = right[j]
            j += 1
            k += 1
```

Para buscar el producto deseado se pedirá la id y se procederá a buscar utilizando el algoritmo de Búsqueda Binaria.

```python
# Búsqueda binaria por código
def buscar_producto(productos):
    codigo = input("Ingrese el código del producto a buscar: ").upper()
    inicio = 0
    fin = len(productos) - 1

    while inicio <= fin:
        medio = (inicio + fin) // 2
        cod_actual = productos[medio]["codigo"]

        if cod_actual == codigo:
            print("Producto encontrado:")
            print(f"Código: {productos[medio]['codigo']}")
            print(f"Nombre: {productos[medio]['nombre']}")
            return productos[medio]
        elif codigo < cod_actual:
            fin = medio - 1
        else:
            inicio = medio + 1

    print("Producto no encontrado.")
    return None
```

Luego desarrollamos un script para generar una serie de datos abarcando del elemento A000 al Z999 que nos da aproximadamente 26.ooo elementos.

```
#GENERAMOS UN ID ALFANUERICO UNICO PARA CADA PRODUCTO CON LA ESTRUCTURA "A000" POR EJEMPLO PARA EL PRIMER VALOR
# Letras de la A a la Z
letras = [chr(c) for c in range(65,91)]
# Números de 000 a 999 con formato fijo de 3 dígitos
numeros = [f"{i:03}" for i in range(1000)]
# Producto cartesiano: todas las combinaciones letra + número
id = [letra + numero for letra, numero in itertools.product(letras, numeros)]

#Generamos lista de productos usando comprehension lists, recorriendo los ids generados, agregamos campo de produc
productos = [[i,
              random.choice(["Mouse", "Teclado", "Monitor", "PC"]),
              random.choice(["Zona Sur","Zona Norte","CABA","Interior"])]
              for i in id]
```

Luego tomamos subset de diferentes tamaños y los desordenamos para poder aplicar los algoritmos de ordenamiento y búsqueda.

```
#TOMAMOS SUBSETS DE LISTA PRINCIPAL
set_chico = productos[:100]
set_mediano = productos[:10000]
set_grande = productos[:100000]

#LOS DESORDENAMOS PARA LUEGO PROBAR LOS ALGORITMOS DE ORDENAMIENTO
random.shuffle(set_chico)
random.shuffle(set_mediano)
random.shuffle(set_grande)
```

Y por último comparamos como era la performance de Quicksort vs Bubble Sort a la hora del ordenamiento, utilizando la librería “time” viendo el periodo de tiempo transcurrido de ejecución. Pudimos visualizar que con sets de datos grandes N = 25.000, empezaba a notarse una gran diferencia performance en términos de tiempo (67segs):

```
QUICKSORT
Tiempo ejecucion con muestra grande: 0.1755661964416504
Tiempo ejecucion con muestra mediana: 0.04436659812927246
Tiempo ejecucion con muestra chica: 0.0
BUBBLE SORT
Tiempo ejecucion con muestra grande: 67.8942518234253
Tiempo ejecucion con muestra mediana: 11.211306095123291
Tiempo ejecucion con muestra chica: 0.000993967056274414
```

Respecto a los algoritmos de búsqueda apreciamos solo diferencias marginales utilizando una muestra grande de N = 260.000, y forzando al método Lineal a buscar un elemento ubicado más bien al final de una lista ordenada (worst case). La Búsqueda Binaria fue casi instantánea.

```
LINEAL
Tiempo ejecucion con muestra grande: 0.016812801361083984, elemento encontrado en 259000
BINARIA
Tiempo ejecucion con muestra grande: 0.0, elemento encontrado en 259000
```

Conclusiones

Los algoritmos de búsqueda y ordenamiento son pilares esenciales en el campo de la informática, ya que permiten organizar y acceder eficientemente a grandes volúmenes de datos. Su correcta implementación impacta directamente en el rendimiento de sistemas,

aplicaciones y procesos, desde tareas simples como ordenar una lista hasta operaciones complejas como la planificación de procesos o la búsqueda de rutas en inteligencia artificial. Comprender su funcionamiento y saber cuándo aplicar cada uno es clave para desarrollar soluciones informáticas eficientes, escalables y robustas.

Como mencionan Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest y Clifford Stein en su libro Introducción a los algoritmos:
“Tener una base sólida de conocimientos y técnicas algorítmicas es una de las características que distingue a los programadores realmente capacitados de los principiantes.”

Anexo:

Tipos de algoritmos de búsqueda:

- Búsqueda lineal: Revisa un elemento a la vez.

```python
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i
    return -1
```

Peor caso O(n)
Mejor caso O(1)

- Búsqueda binaria: Requiere la lista ordenada. Divide el conjunto en dos y descarta la mitad que no puede contener el elemento buscado, repitiendo el proceso.

```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
        elif lista[medio] < objetivo:
            izquierda = medio + 1
        else:
            derecha = medio - 1
    return -1
```

Peor caso O(log n)
Mejor caso O(1)

Tipos de algoritmos de ordenamiento:

- Ordenamiento burbuja: Intercambia elementos adyacentes cuando están en un orden incorrecto.

```python
def bubble_sort(arr):
    n = len(arr)
    for i in range(n):
        for j in range(0, n - i - 1):
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
```

Peor caso $O(n^2)$
Mejor caso $O(n)$

- Ordenamiento por inserción: Construye una lista ordenada insertando los elementos en la posición correcta.

```python
def insertion_sort(arr):
  for i in range(1, len(arr)):
    key = arr[i]
    j = i-1
    while j >=0 and key < arr[j] :
        arr[j+1] = arr[j]
        j -= 1
    arr[j+1] = key
```

Peor caso $O(n^2)$
Mejor caso $O(n)$

- Ordenamiento por selección: Iterativamente selecciona el elemento más pequeño y lo coloca primero en la lista.

```python
def selection_sort(arr):
    n = len(arr)
    for i in range(n):
        # Encontrar el índice del elemento mínimo
        min_index = i
        for j in range(i+1, n):
            if arr[j] < arr[min_index]:
                min_index = j
        # Intercambiar el elemento mínimo con el elemento actual
        arr[i], arr[min_index] = arr[min_index], arr[i]
```

Peor caso $O(n^2)$
Mejor caso $O(n^2)$ aun con la lista ordenada

- Quicksort: Se ordena los elementos más grandes y más chicos a partir de un elemento pívot. Luego a estos dos grupos se le vuelve a aplicar quicksort, y asi recursivamente.

```python
def quicksort(arr):
    if len(arr) <= 1:
        return arr
    else:
        pivot = arr[0]
        less = [x for x in arr[1:] if x <= pivot]
        greater = [x for x in arr[1:] if x > pivot]
        return quicksort(less) + [pivot] + quicksort(greater)
```

Peor caso $O(n^2)$
Mejor caso O(n log n)

- Mergesort: Similar a quicksort, este método divide por la mitad el arreglo.

```python
def merge_sort(arr):
    if len(arr) > 1:
        mid = len(arr) // 2
        left_half = arr[:mid]
        right_half = arr[mid:]

        merge_sort(left_half)
        merge_sort(right_half)

        i = j = k = 0

        while i < len(left_half) and j < len(right_half):
            if left_half[i] < right_half[j]:
                arr[k] = left_half[i]
                i += 1
            else:
                arr[k] = right_half[j]
                j += 1
            k += 1
        while i < len(left_half):
            arr[k] = left_half[i]
            i += 1
            k += 1
        while j < len(right_half):
            arr[k] = right_half[j]
            j += 1
            k += 1
```

Peor caso $O(n^2)$
Mejor caso O(n log n)