



TECNICATURA UNIVERSITARIA EN PROGRAMACIÓN A DISTANCIA

Trabajo Integrador - Propuesta de Investigación. Cátedra de Programación I

Búsqueda y ordenamiento

German Luis Dagatti
german.dagatti@tupad.utn.edu.ar
Nicolas Gabriel Demiryi
nicolas.demiryi@tupad.utn.edu.ar

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

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> python -u "c:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programaci
ón I\TP_Integrador_Programacion_I\scripts\busqueda_lineal.py"
El elemento 8 se encuentra en la posición 3.
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> []
```

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.

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> python -u "c:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I\scripts\tempCodeRunnerFile.py"
• El elemento 7 se encuentra en la posición 2.
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I>
```

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

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  PUERTOS
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> python -u "c:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I\scripts\tempCodeRunnerFile.py"
El elemento 5 se encuentra en la posición 3.
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> █
```

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:

- La función toma dos parámetros: `lista` (la lista de elementos) y `objetivo` (el elemento objetivo a encontrar).
- Se inicializan dos posiciones, `izquierda` y `derecha`, que representan los índices de inicio y fin de la lista, respectivamente.

2. Bucle Principal:

- El bucle `while` se ejecuta mientras `izquierda` sea menor o igual a `derecha`.
- En cada iteración, se calcula el índice medio (`medio`)
- .

3. Comparación y Actualización de Punteros:

- Si `lista[medio]` es igual al `objetivo`, se retorna el índice `medio`.
- Si `lista[medio]` es menor que `objetivo`, se actualiza `izquierda` a `medio + 1` para buscar en la mitad derecha.
- Si `lista[medio]` es mayor que `objetivo`, se actualiza `derecha` a `medio - 1` para buscar en la mitad izquierda.

4. Resultado:

- Si el bucle termina sin encontrar el `objetivo`, se retorna `-1`.

Ejemplo de Uso

```
import busqueda_binaria as bb

lista = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
objetivo = 7
resultado = bb.busqueda_binaria(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

El ordenamiento organiza los datos de acuerdo a un criterio, como de menor a mayor o alfabéticamente.

Los algoritmos de ordenamiento son importantes porque permiten organizar y estructurar datos de manera eficiente. Al ordenar los datos, se pueden realizar búsquedas, análisis y otras operaciones de manera más rápida y sencilla.

Algunos de los beneficios de utilizar algoritmos de ordenamiento incluyen:

- **Búsqueda más eficiente:** Una vez que los datos están ordenados, es mucho más fácil buscar un elemento específico. Esto se debe a que se puede utilizar la

búsqueda binaria, que es un algoritmo de búsqueda mucho más eficiente que la búsqueda lineal.

- **Análisis de datos más fácil:** Los datos ordenados pueden ser analizados más fácilmente para identificar patrones y tendencias. Por ejemplo, si se tienen datos sobre las ventas de una empresa, se pueden ordenar por producto, región o fecha para ver qué productos se venden mejor, en qué regiones se venden más productos o cómo cambian las ventas con el tiempo.

- **Operaciones más rápidas:** Muchas operaciones, como fusionar dos conjuntos de datos o eliminar elementos duplicados, se pueden realizar de manera más rápida y eficiente en datos ordenados.

Existen muchos algoritmos de ordenamiento diferentes, cada uno con sus propias ventajas y desventajas. Algunos de los algoritmos de ordenamiento más comunes incluyen:

1. Ordenamientos Simples
 - a. *Bubble Sort* (Ordenamiento burbuja): Compara elementos adyacentes y los intercambia si están en el orden incorrecto. Fácil de entender, pero ineficiente ($O(n^2)$).
 - b. *Insertion Sort* (Ordenamiento por inserción): Inserta cada elemento en su posición correcta en una lista ya ordenada. Bueno para listas pequeñas o casi ordenadas.
 - c. *Selection Sort* (Ordenamiento por selección): Selecciona el mínimo y lo coloca en su posición final. Simple, pero también lento ($O(n^2)$).
2. Ordenamientos Eficientes (Divide y vencerás)
 - a. *Merge Sort* (Ordenamiento por mezcla): Divide la lista en mitades, ordena cada mitad y luego las fusiona. Eficiencia: $O(n \log n)$ y estable. Usa recursividad.
 - b. *Quick Sort* (Ordenamiento rápido): Elige un pivote, divide la lista en menores y mayores, y ordena recursivamente. Muy eficiente en promedio: $O(n \log n)$ (pero $O(n^2)$ en el peor caso). No es estable por defecto.
3. Ordenamientos No Comparativos (Basados en propiedades del dato)
 - a. *Counting Sort* (Ordenamiento por conteo): Cuenta las ocurrencias de cada valor. Solo sirve para enteros en un rango pequeño. Tiempo: $O(n + k)$.

- b. *Radix Sort* (Ordenamiento por radix): Ordena dígito a dígito (de menor a mayor posición). Eficiente con números grandes si se combinan con Counting Sort.
 - c.
4. Integrado en Python: `sorted()` y `.sort()`. Python usa Timsort, una mezcla de Merge Sort e Insertion Sort. Eficiente: $O(n \log n)$ y estable.

```
lista = [4, 1, 3, 9, 2]
ordenada = sorted(lista) # No modifica la original
lista.sort()             # Modifica la lista original
```

Ordenamientos Simples

Ordenamiento por burbuja

El ordenamiento por burbuja (Bubble Sort) es uno de los algoritmos de ordenamiento más simples y fáciles de entender. Funciona comparando pares de elementos adyacentes en una lista e intercambiándolos si están en el orden incorrecto. Este proceso se repite hasta que toda la lista esté ordenada.

Ejemplo de Ordenamiento por burbuja en Python

```
def ordenamiento_burbuja(lista):
    n = len(lista)
    for i in range(n):
        # Para detectar si se realizaron intercambios en esta pasada
        intercambio = False
        for j in range(0, n - i - 1):
            # Comparar el elemento actual con el siguiente
            if lista[j] > lista[j + 1]:
                # Intercambiar los elementos
                lista[j], lista[j + 1] = lista[j + 1], lista[j]
                intercambio = True
        # Si no se realizaron intercambios, la lista ya está ordenada
        if not intercambio:
            break
    return lista

# Ejemplo de uso
if __name__ == "__main__":
    lista = [64, 34, 25, 12, 22, 11, 90]
    print("Lista original:", lista)
    lista_ordenada = ordenamiento_burbuja(lista)
```

```
print("Lista ordenada:", lista_ordenada)
```

Explicación Paso a Paso

1. Bucle Principal:

- El bucle for exterior itera n veces, donde n es la longitud de la lista. Cada iteración del bucle exterior asegura que el elemento más grande no ordenado se mueva a su posición correcta.

2. Bucle Interno:

- El bucle for interior itera desde el inicio de la lista hasta $n-i-1$. En cada iteración, se comparan dos elementos adyacentes y se intercambian si están en el orden incorrecto.

3. Intercambio:

- Se utiliza la variable intercambio para detectar si se realizaron intercambios en una pasada completa del bucle interior. Si no se realizaron intercambios, la lista ya está ordenada y se puede terminar el algoritmo antes de tiempo.

4. Optimización:

- Si en una pasada completa del bucle interior no se realizan intercambios, la lista ya está ordenada y se puede salir del bucle exterior.

Ejemplo de Uso

```
import ordenamiento_burbuja as ob

lista = [64, 34, 25, 12, 22, 11, 90]
print("Lista original:", lista)
lista_ordenada = ob.ordenamiento_burbuja(lista)
print("Lista ordenada:", lista_ordenada)
```

```
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> python -u "c:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I\scripts\Ordenamiento\ordenamiento_burbuja.py"
• Lista original: [64, 34, 25, 12, 22, 11, 90]
  Lista ordenada: [11, 12, 22, 25, 34, 64, 90]
○ PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I>
```

- **Lista Original:** [64, 34, 25, 12, 22, 11, 90]
- **Lista Ordenada:** [11, 12, 22, 25, 34, 64, 90]

Complejidad del Algoritmo

- **Complejidad Temporal:**
 - **Mejor Caso:** $O(n)$ si la lista ya está ordenada.

- **Caso Promedio:** $O(n^2)$ en la mayoría de los casos prácticos.
- **Peor Caso:** $O(n^2)$ cuando la lista está en orden inverso.
- **Complejidad Espacial:**
 - **In-place:** $O(1)$ ya que no requiere espacio adicional significativo.

Ventajas y Desventajas

Ventajas:

- **Sencillo de Implementar:** El algoritmo es muy fácil de entender y codificar.
- **Estable:** Mantiene el orden relativo de elementos duplicados.
- **In-place:** No requiere espacio adicional significativo.

Desventajas:

- **Ineficiente para Listas Grandes:** Tiene una complejidad temporal de $O(n^2)$ en el peor y caso promedio, lo que lo hace ineficiente para listas grandes.
- **Peor Caso:** El peor caso ocurre cuando la lista está en orden inverso, lo que resulta en el máximo número de comparaciones y intercambios.

Uso Práctico

El ordenamiento por burbuja es especialmente útil en situaciones donde:

- La lista es pequeña.
- La lista está casi ordenada.
- Se prefiere una implementación simple y rápida.

Ordenamiento por Inserción

El ordenamiento por inserción (Insertion Sort) es un algoritmo de ordenamiento simple y eficiente para listas pequeñas o casi ordenadas. A continuación, se presenta una implementación en Python, junto con una explicación paso a paso.

Ejemplo de Ordenamiento por Inserción en Python

```
def ordenamiento_insercion(lista):  
    # Recorre la lista desde el segundo elemento hasta el final  
    for i in range(1, len(lista)):  
        indice = lista[i] # Elemento actual a insertar  
        j = i - 1 # Índice del elemento anterior  
  
        # Mover elementos mayores que el índice hacia la derecha  
        while j >= 0 and indice < lista[j]:
```

```
        lista[j + 1] = lista[j]
        j -= 1

    # Insertar índice en su posición correcta
    lista[j + 1] = indice
    return lista

# Ejemplo de uso
if __name__ == "__main__":
    lista = [12, 11, 13, 5, 6]
    print("Lista original:", lista)
    lista_ordenada = ordenamiento_insercion(lista)
    print("Lista ordenada:", lista_ordenada)
```

Explicación Paso a Paso

1. Bucle Principal:

- El bucle for recorre la lista desde el segundo elemento hasta el final. El primer elemento se considera ya ordenado.
- En cada iteración, se toma el elemento actual (índice) y se inserta en la parte ya ordenada de la lista.

2. Mover Elementos:

- El bucle while se utiliza para comparar el elemento actual (índice) con los elementos de la parte ya ordenada.
- Si un elemento de la parte ordenada es mayor que el índice, se desplaza hacia la derecha para hacer espacio para el índice.

3. Insertar la Clave:

- Una vez que se encuentra la posición correcta para el índice, se inserta en esa posición.

Ejemplo de Uso

```
import ordenamiento_insercion as oi

lista = [12, 11, 13, 5, 6]
print("Lista original:", lista)
lista_ordenada = oi.ordenamiento_insercion(lista)
```

```
print("Lista ordenada:", lista_ordenada)
```

```
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> python -u "c:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I\scripts\Ordenamiento\Ej_insercion.py"
Lista original: [12, 11, 13, 5, 6]
Lista ordenada: [5, 6, 11, 12, 13]
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I>
```

- **Lista Original:** [12, 11, 13, 5, 6]
- **Lista Ordenada:** [5, 6, 11, 12, 13]

Complejidad del Algoritmo

- **Complejidad Temporal:**
 - **Mejor Caso:** $O(n)$ cuando la lista ya está ordenada.
 - **Caso Promedio:** $O(n^2)$ para listas desordenadas.
 - **Peor Caso:** $O(n^2)$ cuando la lista está en orden inverso.
- **Complejidad Espacial:**
 - **In-place:** $O(1)$ ya que no requiere espacio adicional significativo.

Ventajas y Desventajas

Ventajas:

- **Sencillo de Implementar:** El algoritmo es fácil de entender y codificar.
- **Eficiente para Pequeñas Listas:** Funciona bien para listas pequeñas o casi ordenadas.
- **Estable:** Mantiene el orden relativo de elementos duplicados.
- **In-place:** No requiere espacio adicional significativo.

Desventajas:

- **Ineficiente para Grandes Listas:** Tiene una complejidad temporal de $O(n^2)$ en el peor caso, lo que lo hace ineficiente para listas grandes.

Uso Práctico

El ordenamiento por inserción es especialmente útil en situaciones donde:

- La lista es pequeña.
- La lista está casi ordenada.
- Se requiere un algoritmo estable.
- Se prefiere una implementación simple y eficiente en términos de espacio.

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):
        # Encontrar el índice del elemento mínimo en la parte no
ordenada
        min = i
        for j in range(i + 1, n):
            if lista[j] < lista[min]:
                min = j

        # Intercambiar el elemento mínimo encontrado con el primer
elemento de la parte no ordenada
        lista[i], lista[min] = lista[min], lista[i]

    return lista

# Ejemplo de uso
if __name__ == "__main__":
    lista = [64, 25, 12, 22, 11]
    print("Lista original:", lista)
    lista_ordenada = ordenamiento_seleccion(lista)
    print("Lista ordenada:", lista_ordenada)
```

Explicación Paso a Paso

1. Bucle Principal:

- El bucle for exterior itera desde el inicio de la lista hasta el final. En cada iteración, se asume que el primer elemento de la parte no ordenada es el mínimo.

2. Búsqueda del Mínimo:

- El bucle for interior itera desde el elemento actual hasta el final de la lista para encontrar el elemento mínimo en la parte no ordenada.
- Si se encuentra un elemento más pequeño que el actual mínimo, se actualiza el índice del mínimo.

3. Intercambio:

- Una vez que se encuentra el elemento mínimo en la parte no ordenada, se intercambia con el primer elemento de la parte no ordenada.
- Esto asegura que el elemento más pequeño de la parte no ordenada se mueva a la parte ordenada.

4. Repetición:

- El proceso se repite hasta que toda la lista esté ordenada.

Ejemplo de Uso

```
import ordenamiento_seleccion as os

lista = [64, 25, 12, 22, 11]
print("Lista original:", lista)
lista_ordenada = os.ordenamiento_seleccion(lista)
print("Lista ordenada:", lista_ordenada)
```

```
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> python -u "c:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programación I\scripts\Ordenamiento\tempCodeRunnerFile.py"
Lista original: [64, 25, 12, 22, 11]
Lista ordenada: [11, 12, 22, 25, 64]
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> █
```

- **Lista Original:** [64, 25, 12, 22, 11]
- **Lista Ordenada:** [11, 12, 22, 25, 64]

Complejidad del Algoritmo

- **Complejidad Temporal:**
 - **Mejor Caso:** $O(n^2)$ incluso si la lista ya está ordenada.
 - **Caso Promedio:** $O(n^2)$ en la mayoría de los casos prácticos.
 - **Peor Caso:** $O(n^2)$ cuando la lista está en orden inverso.
- **Complejidad Espacial:**

- **In-place:** $O(1)$ ya que no requiere espacio adicional significativo.

Ventajas y Desventajas

Ventajas:

- **Sencillo de Implementar:** El algoritmo es muy fácil de entender y codificar.
- **In-place:** No requiere espacio adicional significativo.
- **Estable:** Mantiene el orden relativo de elementos duplicados (aunque no es inherentemente estable, se puede modificar para serlo).

Desventajas:

- **Ineficiente para Listas Grandes:** Tiene una complejidad temporal de $O(n^2)$ en todos los casos, lo que lo hace ineficiente para listas grandes.
- **Peor Caso:** El peor caso ocurre cuando la lista está en orden inverso, lo que resulta en el máximo número de comparaciones y intercambios.

Uso Práctico

El ordenamiento por selección es especialmente útil en situaciones donde:

- La lista es pequeña.
- Se prefiere una implementación simple y rápida.

Comparación entre ordenamiento por Burbujas, por Inserción y por Selección

A continuación, se presenta una comparación detallada de estos tres algoritmos en términos de complejidad temporal, complejidad espacial, estabilidad y uso práctico.

	Ordenamiento por Burbuja	Ordenamiento por Inserción	Ordenamiento por Selección
Complejidad Temporal	$O(n)$ mejor, $O(n^2)$ promedio y peor	$O(n)$ mejor, $O(n^2)$ promedio y peor	$O(n^2)$ en todos los casos
Complejidad Espacial	$O(1)$	$O(1)$	$O(1)$
Estable	Sí	Sí	No
Implementación	Muy sencillo	Sencillo	Sencillo
Uso práctico	Listas pequeñas o casi ordenadas	Listas pequeñas o casi ordenadas	Listas pequeñas

Ordenamientos Eficientes (Divide y vencerás)

Ordenamiento por mezcla

El ordenamiento por mezcla (Merge Sort) es un algoritmo de ordenamiento eficiente que utiliza la técnica de "divide y vencerás". Funciona dividiendo la lista en mitades, ordenando cada mitad y luego mezclando las dos mitades ordenadas para producir una lista completamente ordenada.

Ejemplo de Ordenamiento por mezcla en Python

```
def ordenamiento_mezcla(lista):  
    if len(lista) <= 1:  
        return lista # Una lista de un elemento o menos ya está  
ordenada  
  
    # Dividir la lista en dos mitades  
    medio = len(lista) // 2  
    izquierda = lista[:medio]  
    derecha = lista[medio:]  
  
    # Llamar recursivamente al ordenamiento_mezcla para ambas mitades  
    izquierda = ordenamiento_mezcla(izquierda)  
    derecha = ordenamiento_mezcla(derecha)  
  
    # Mezclar las dos mitades ordenadas  
    return mezclar(izquierda, derecha)  
  
def mezclar(izquierda, derecha):  
    resultado = []  
    i = j = 0  
  
    # Comparar elementos de ambas listas y añadir el menor al  
resultado  
    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  
    # Añadir los elementos restantes de la lista no procesada  
    resultado.extend(izquierda[i:])  
    resultado.extend(derecha[j:])  
    return resultado
```

```
j += 1

# Agregar los elementos restantes de la lista izquierda, si los
hay
while i < len(izquierda):
    resultado.append(izquierda[i])
    i += 1

# Agregar los elementos restantes de la lista derecha, si los hay
while j < len(derecha):
    resultado.append(derecha[j])
    j += 1

return resultado

# Ejemplo de uso
if __name__ == "__main__":
    lista = [38, 27, 43, 3, 9, 82, 10]
    print("Lista original:", lista)
    lista_ordenada = ordenamiento_mezcla(lista)
    print("Lista ordenada:", lista_ordenada)
```

Explicación paso a paso

1. Función ordenamiento_mezcla

- Esta es la función principal del algoritmo.
- Si la lista tiene un tamaño de 1 o menos, se considera que ya está ordenada y se devuelve tal cual.
- La lista se divide en dos mitades: izquierda y derecha.
- Se llama recursivamente a ordenamiento_mezcla para ordenar ambas mitades.
- Finalmente, se llama a la función mezclar para combinar las dos mitades ordenadas.

2. Función mezclar:

- Esta función toma dos listas ordenadas (izquierda y derecha) y las combina en una sola lista ordenada.

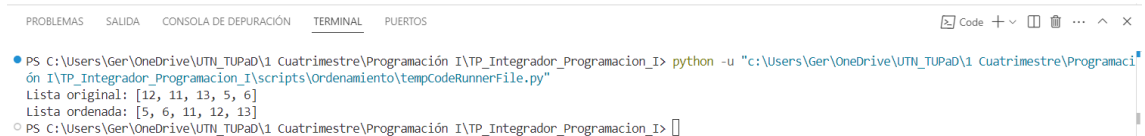
- Se inicializa una lista vacía resultado para almacenar la lista combinada.
- Se utilizan dos índices, i y j , para recorrer las listas izquierda y derecha, respectivamente.
- Se comparan los elementos de ambas listas y se añade el menor al resultado.
- Una vez que una de las listas se agota, se añaden los elementos restantes de la otra lista al resultado.

3. Ejemplo de Uso:

- Se define una lista de números desordenados.
- Se llama a `ordenamiento_mezcla` para ordenar la lista.
- Se imprime la lista original y la lista ordenada.
-

```
import ordenamiento_mezcla as om
```

```
lista = [12, 11, 13, 5, 6]
print("Lista original:", lista)
lista_ordenada = om.ordenamiento_mezcla(lista)
print("Lista ordenada:", lista_ordenada)
```



```
PS C:\Users\Ger\OneDrive\UTN_TUPad\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> python -u "c:\Users\Ger\OneDrive\UTN_TUPad\1 Cuatrimestre\Programación I\TP_Integrador_Programación I\scripts\Ordenamiento\tempCodeRunnerFile.py"
Lista original: [12, 11, 13, 5, 6]
Lista ordenada: [5, 6, 11, 12, 13]
PS C:\Users\Ger\OneDrive\UTN_TUPad\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I>
```

- **Lista Original:** [12, 11, 13, 5, 6]
- **Lista Ordenada:** [5, 6, 11, 12, 13]

Complejidad del Algoritmo

- **Tiempo de Ejecución:** El ordenamiento por mezcla tiene una complejidad temporal de $O(n \log n)$, donde n es el número de elementos en la lista. Esto lo hace muy eficiente para grandes conjuntos de datos.
- **Espacio de Ejecución:** Requiere $O(n)$ de espacio adicional debido a la necesidad de almacenar las sub-listas durante el proceso de mezcla.

El ordenamiento por mezcla es especialmente útil cuando se necesitan garantizar tiempos de ejecución consistentes y se dispone de suficiente memoria para almacenar las sub-listas.

Ventajas y Desventajas

Ventajas:

- **Eficiencia:** Tiene una complejidad temporal de $O(n \log n)$, lo que lo hace muy eficiente para grandes conjuntos de datos.
- **Estabilidad:** El ordenamiento por mezcla es un algoritmo estable, lo que significa que mantiene el orden relativo de elementos iguales.
- **Consistencia:** El tiempo de ejecución es consistente y no depende de la distribución inicial de los datos.

Desventajas:

- **Espacio Adicional:** Requiere $O(n)$ de espacio adicional, lo que puede ser un problema en entornos con memoria limitada.
- **Implementación Recursiva:** La implementación recursiva puede ser menos eficiente en términos de uso de la pila de llamadas, aunque esto generalmente no es un problema en la práctica.

En resumen, el Ordenamiento por mezcla es un algoritmo de ordenamiento muy eficiente y consistente, aunque requiere un espacio adicional significativo.

Ordenamiento rápido

El Ordenamiento rápido es un algoritmo de ordenamiento eficiente que utiliza la técnica de "divide y vencerás". Funciona seleccionando un elemento como pivote y particionando la lista en dos sub-listas: una con elementos menores que el pivote y otra con elementos mayores. Luego, se aplica recursivamente el mismo proceso a las sub-listas.

Ejemplo de Ordenamiento rápido en Python

```
def ordenamiento_rapido(lista):  
    if len(lista) <= 1:  
        return lista # Una lista de un elemento o menos ya está  
ordenada  
  
    # Seleccionar el pivote (en este caso, el último elemento)  
    pivote = lista[-1]  
    menores = []  
    mayores = []
```

```
iguales = []

# Particionar la lista
for x in lista:
    if x < pivote:
        menores.append(x)
    elif x > pivote:
        mayores.append(x)
    else:
        iguales.append(x)

# Aplicar Ordenamiento rápido recursivamente a las sub-listas
return ordenamiento_rapido(menores) + iguales +
ordenamiento_rapido(mayores)

# Ejemplo de uso
if __name__ == "__main__":
    lista = [3, 6, 8, 10, 1, 2, 1]
    print("Lista original:", lista)
    lista_ordenada = ordenamiento_rapido(lista)
    print("Lista ordenada:", lista_ordenada)
```

Explicación Paso a Paso

1. Caso Base:

- Si la lista tiene un tamaño de 1 o menos, se considera que ya está ordenada y se devuelve tal cual.

2. Selección del Pivote:

- En este ejemplo, se selecciona el último elemento de la lista como pivote. Otras estrategias incluyen seleccionar el primer elemento, el elemento medio o un pivote aleatorio.

3. Particionamiento:

- Se recorre la lista y se dividen los elementos en tres categorías:
 - menores: Elementos menores que el pivote.
 - iguales: Elementos iguales al pivote.

- mayores: Elementos mayores que el pivote.

4. Recursión:

- Se aplica el Ordenamiento rápido recursivamente a las sub-listas menores y mayores.
- Finalmente, se concatenan las listas ordenadas de menores, iguales y mayores para obtener la lista completamente ordenada.

Ejemplo de Uso

```
import ordenamiento_rapido as ora

lista = [3, 6, 8, 10, 1, 2, 1]
print("Lista original:", lista)
lista_ordenada = ora.ordenamiento_rapido(lista)
print("Lista ordenada:", lista_ordenada)
```

```
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> python -u "c:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programaci
ón I\TP_Integrador_Programacion_I\scripts\Ordenamiento\tempCodeRunnerFile.py"
Lista original: [3, 6, 8, 10, 1, 2, 1]
Lista ordenada: [1, 1, 2, 3, 6, 8, 10]
PS C:\Users\Ger\OneDrive\UTN_TUPaD\1 Cuatrimestre\Programación I\TP_Integrador_Programacion_I> █
```

- **Lista Original:** [3, 6, 8, 10, 1, 2, 1]
- **Lista Ordenada:** [1, 1, 2, 3, 6, 8, 10]

Complejidad del Algoritmo

- **Complejidad Temporal:**
 - **Mejor Caso:** $O(n \log n)$ cuando la partición es balanceada.
 - **Caso Promedio:** $O(n \log n)$ en la mayoría de los casos prácticos.
 - **Peor Caso:** $O(n^2)$ cuando la partición es desbalanceada (por ejemplo, si el pivote siempre es el mínimo o el máximo).
- **Complejidad Espacial:**
 - **Recursión:** $O(\log n)$ en promedio, $O(n)$ en el peor caso debido a la pila de llamadas recursivas.

Ventajas y Desventajas

Ventajas:

- **Eficiente en Promedio:** Tiene una complejidad temporal de $O(n \log n)$ en el caso promedio.

- **In-place:** No requiere espacio adicional significativo si se implementa correctamente.
- **Sencillo de Implementar:** Es relativamente fácil de entender y codificar.

Desventajas:

- **Peor Caso:** Puede degradarse a $O(n^2)$ si no se elige un buen pivote.
- **No Estable:** No mantiene el orden relativo de elementos duplicados.

Uso Práctico

El Quick Sort es un algoritmo muy utilizado en la práctica debido a su eficiencia en el caso promedio y su implementación sencilla. Sin embargo, en aplicaciones donde la estabilidad es crucial, se debe preferir un algoritmo estable como Merge Sort.

Comparación de las complejidades del Ordenamiento por Mezcla con el Rápido

El Ordenamiento por Mezcla y el Ordenamiento Rápido son dos algoritmos de ordenamiento muy populares que utilizan la técnica de "divide y vencerás". Aunque ambos tienen una complejidad temporal promedio de $O(n \log n)$, hay diferencias significativas en sus complejidades espaciales, comportamientos en el peor caso y otras características. A continuación, se presenta una comparación detallada de las complejidades de ambos algoritmos.

Característica	Ordenamiento por Mezcla	Ordenamiento Rápido
Complejidad Temporal	$O(n \log n)$ en todos los casos	$O(n \log n)$ promedio, $O(n^2)$ peor caso
Complejidad Espacial	$O(n)$	$O(\log n)$ promedio, $O(n)$ peor caso
Estabilidad	Sí	No
Implementación	Más compleja	Más sencilla
Uso Práctico	Bueno para datos grandes, external sorting	Bueno para datos en memoria, implementación sencilla

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.

La búsqueda lineal y la búsqueda binaria son dos algoritmos de búsqueda con diferentes características y requisitos. La búsqueda lineal es simple y no requiere que la lista esté ordenada, pero es ineficiente para listas grandes. La búsqueda binaria es mucho más eficiente para listas grandes, pero requiere que la lista esté ordenada. La elección del algoritmo depende del contexto específico de la aplicación, incluyendo el tamaño de la lista y si está ordenada.

El ordenamiento por burbuja es un algoritmo simple y fácil de implementar, pero no es adecuado para listas grandes debido a su complejidad cuadrática. El ordenamiento por inserción es un algoritmo simple y eficiente para listas pequeñas o casi ordenadas, pero no es adecuado para listas grandes debido a su complejidad cuadrática. El ordenamiento por selección, al igual que los otros dos, es un algoritmo simple y fácil de implementar, pero no es adecuado para listas grandes debido a su complejidad cuadrática.

Para listas grandes, se recomiendan algoritmos más eficientes como Ordenamiento por Mezcla u Ordenamiento Rápido

El Ordenamiento por Mezcla es un algoritmo excelente para aplicaciones donde la estabilidad y la consistencia en el tiempo de ejecución son cruciales, aunque requiere más memoria. El Ordenamiento Rápido es generalmente preferido en aplicaciones donde la memoria es un factor limitante y la implementación debe ser sencilla, siempre que se puedan manejar los casos de peor desempeño.

En la práctica, muchos algoritmos de ordenamiento híbridos (como Timsort en Python) combinan las ventajas de ambos algoritmos para aprovechar sus fortalezas y mitigar sus debilidades.

Referencias bibliográficas

- 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>
- Búsqueda Lineal vs. Búsqueda Binaria: Comparación y Contraste.
<https://informatecdigital.com/busqueda-lineal-vs-busqueda-binaria-comparacion-y-contraste>
- Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). *Algoritmos y estructuras de datos*. Pearson Education
-

j