Trabajo Práctico Integrador

Materia:

Programación

Tema elegido:

Algoritmo de búsqueda y ordenamiento

Alumno/os:

Lauk Karen, Lopez Leandro

Profesor/a:

Julieta Trapé

Fecha de Entrega:

9 de junio del 2025

Índice

Introducción

Al abordar la propuesta del trabajo integrador, se presentaron diversas temáticas para desarrollar una investigación aplicada utilizando el lenguaje Python. Luego de un análisis, decidimos centrarnos en el estudio de los algoritmos de búsqueda y ordenamiento, considerando su importancia fundamental en la informática y su presencia constante en el desarrollo de software.

La elección de este tema surgió a partir de qué evaluamos tanto el nivel de dificultad como la aplicabilidad de cada opción. Los algoritmos de búsqueda y ordenamiento resultaron ser particularmente atractivos por su utilidad práctica y su relevancia en la optimización del manejo de datos. Además, nos permiten comprender con mayor profundidad el análisis de la eficiencia algorítmica, un aspecto clave en la resolución de problemas informáticos reales.

Consideramos que trabajar con estos algoritmos representa una oportunidad para reforzar conceptos fundamentales de programación, al mismo tiempo que desarrollamos habilidades técnicas como el análisis de complejidad, el uso adecuado de estructuras de datos y la implementación eficiente de soluciones. Asimismo, Python ofrece un entorno accesible y versátil para llevar a cabo esta exploración de manera efectiva.

A través de este trabajo, buscamos no solo aplicar lo aprendido, sino también adquirir una visión crítica sobre la elección y el desempeño de distintos algoritmos según el contexto de uso, mediante la elaboración de un marco teórico, la implementación de un caso práctico, y la documentación completa del proceso en un repositorio Git.

Marco teórico

Búsqueda y Ordenación en informática

En programación informática, la búsqueda y la ordenación son técnicas esenciales para organizar y manipular datos. La búsqueda implica encontrar un elemento específico dentro de un conjunto de datos, mientras que la ordenación consiste en organizar los elementos de una colección siguiendo un criterio determinado.

- ➔ **La búsqueda** consiste en localizar un elemento específico dentro de un conjunto de datos, y puede realizarse en estructuras ordenadas o desordenadas.
- ➔ **La ordenación**, por su parte, implica organizar los elementos de una colección siguiendo un criterio preestablecido, como de menor a mayor o de mayor a menor.

Ambas técnicas desempeñan un papel vital en el rendimiento y la eficiencia de los programas informáticos, ya que permiten optimizar tiempos de acceso y procesamiento de información. Existen diversos algoritmos de búsqueda y ordenación, cada uno con sus propias ventajas y desventajas según el tamaño, tipo y complejidad de los datos.

1. **Algoritmos de Ordenamiento**

Los **algoritmos de ordenamiento** son procedimientos sistemáticos que permiten organizar un conjunto de datos en un orden específico. Estos algoritmos son fundamentales en informática, ya que muchas operaciones computacionales, como búsquedas rápidas o análisis estadísticos, requieren que los datos estén previamente ordenados.

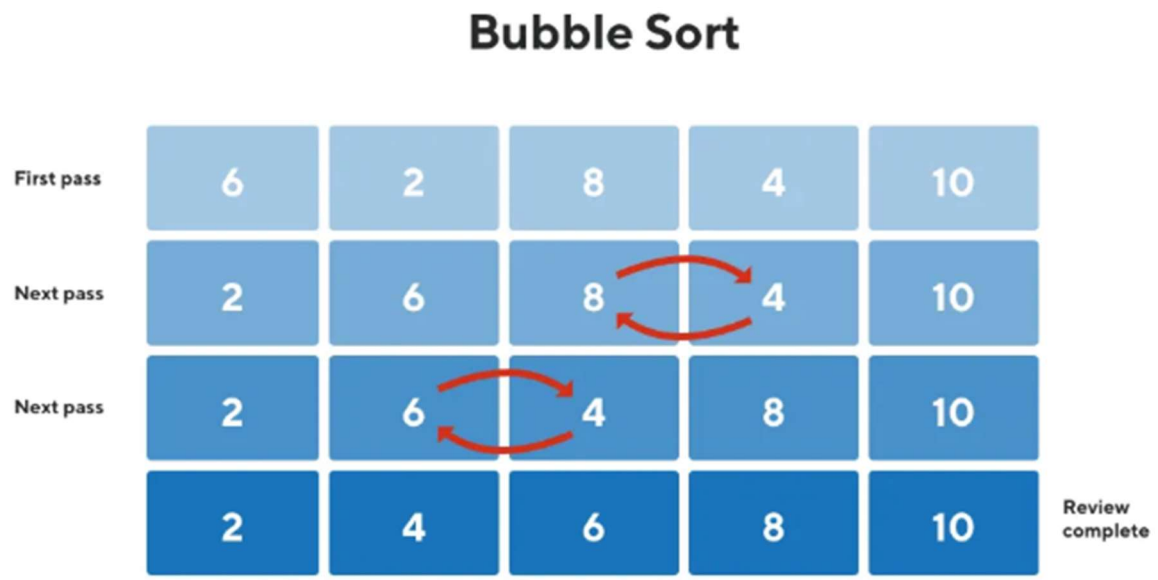
El ordenamiento eficiente de datos mejora considerablemente el tiempo de procesamiento y el rendimiento de los programas. A lo largo de la historia de la computación, se han desarrollado distintos algoritmos de ordenamiento, cada uno con características particulares y adecuado para contextos específicos.

Clasificación de los algoritmos de ordenamiento

1.1 Ordenamiento de burbujas (Bubble Sort)

El ordenamiento de burbuja es uno de los algoritmos de ordenamiento más simples y didácticos, ideal para comprender los principios básicos de algoritmos basados en comparación.

El algoritmo funciona comparando de manera repetitiva los elementos adyacentes de una lista y los intercambia si se encuentran en el orden incorrecto. Este proceso se repite desde el inicio hasta el final de la lista, tantas veces como sea necesario, hasta que la lista esté completamente ordenada.



En cada pasada, el elemento más grande (o más pequeño, según el criterio de ordenamiento) se "desplaza" hacia su posición final al final de la lista, como si fuese una burbuja que sube a la superficie, de ahí su nombre.

Código en Python

```python
def bubble_sort(lista):
    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
```

Ventajas y desventajas

Ventaja:

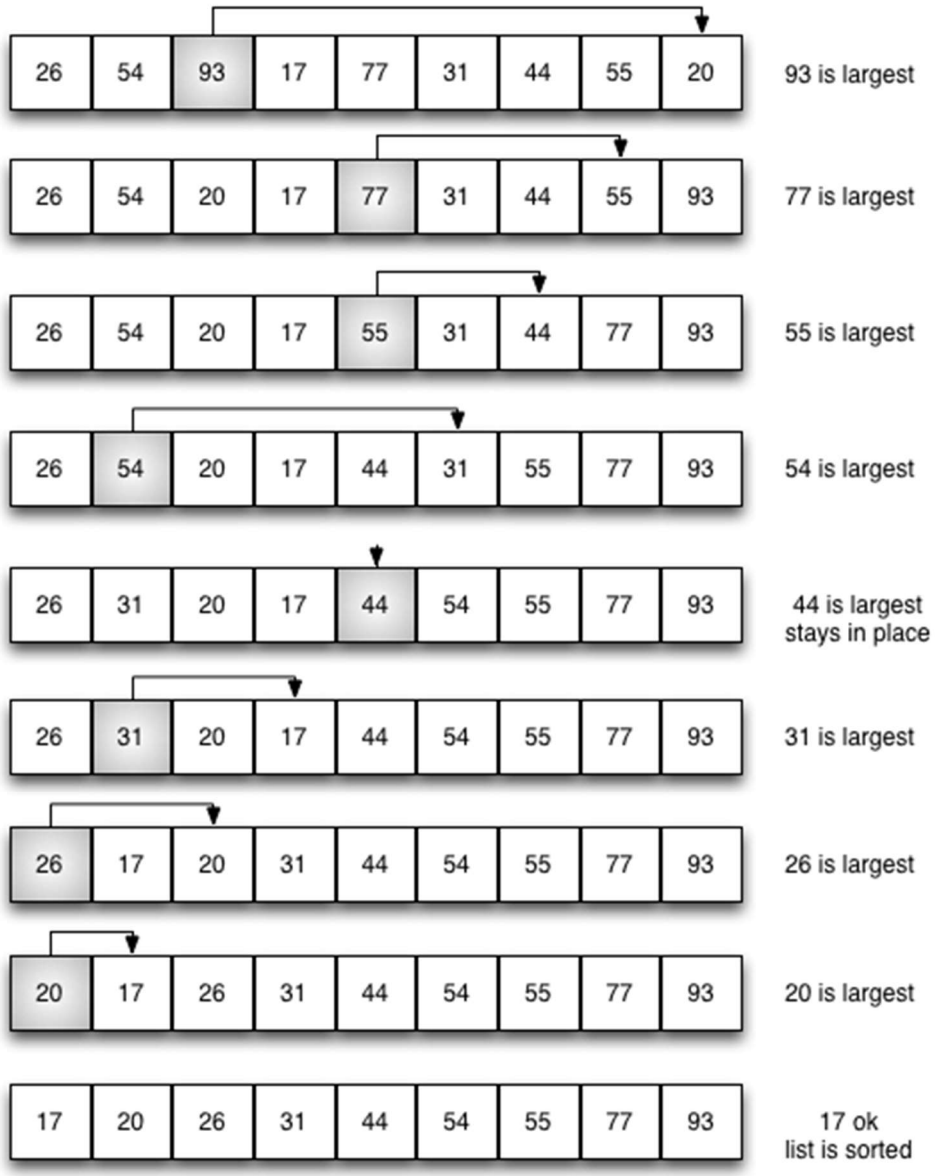
- Simplicidad: Fácil de entender e implementar.
- Implementación sencilla: No requiere estructuras de datos complejas.

Desventajas

- Lento para listas grandes: Su complejidad cuadrática lo vuelve poco práctico para volúmenes de datos grandes.
- No considera el orden parcial: Realiza el mismo número de comparaciones incluso si la lista ya está parcialmente ordenada.

1.2 Ordenamiento por selección (Selection Sort)

El ordenamiento por selección es un algoritmo simple y directo. En cada iteración, identifica el elemento más pequeño del segmento no ordenado de la lista y lo intercambia con el primer elemento no ordenado. Este proceso se repite avanzando una posición en cada paso, hasta que toda la lista queda ordenada.



Código en Python

```python
def selection_sort(lista):
    n = len(lista)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if lista[j] < lista[min_idx]:
                min_idx = j
        lista[i], lista[min_idx] = lista[min_idx], lista[i]
    return lista
```

Ventajas y Desventajas

Ventajas:

- Fácil de entender e implementar
- Número reducido de intercambios

Desventajas:

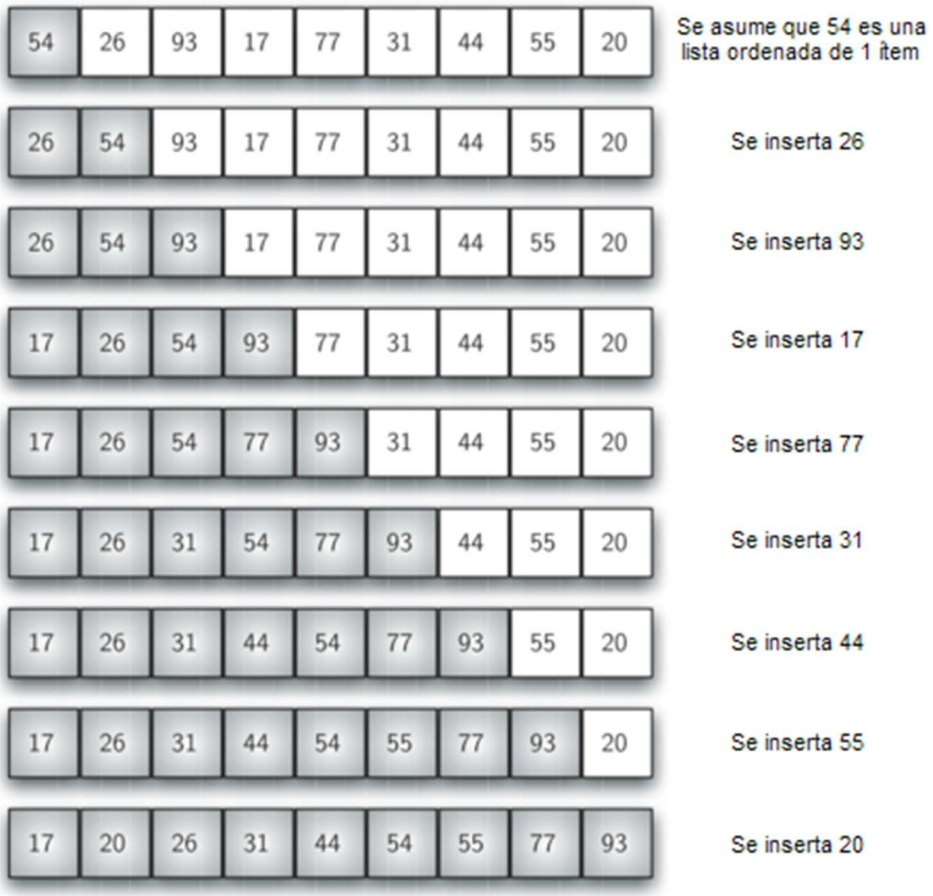
- Ineficiente para listas grandes
- No estable

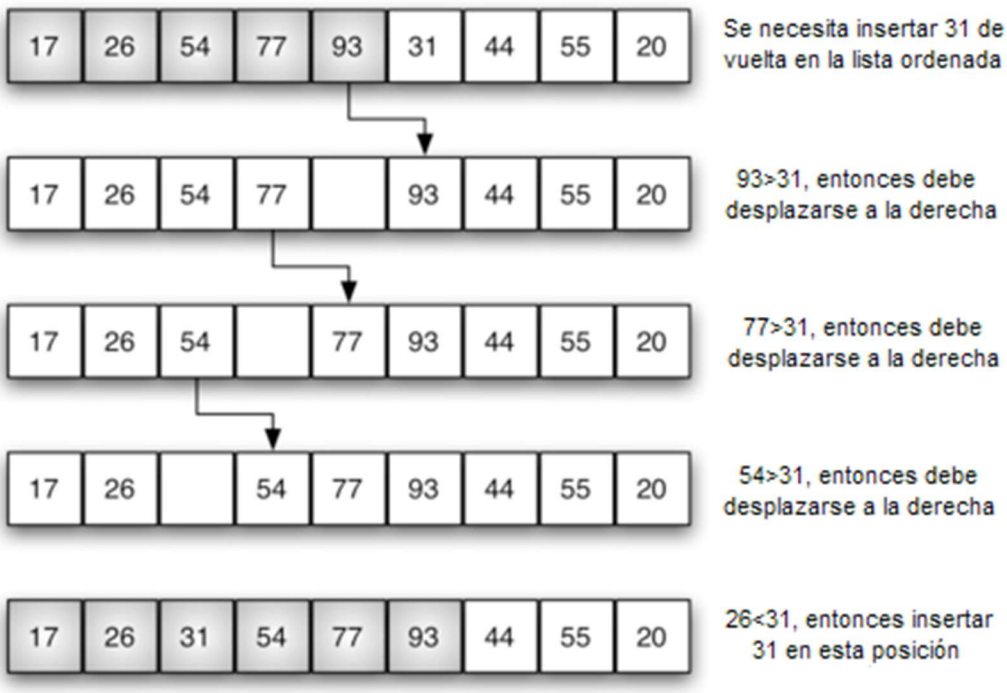
1.3 Ordenamiento por inserción

El ordenamiento por inserción es un algoritmo sencillo pero bastante eficiente en ciertos escenarios. Su funcionamiento se basa en la idea intuitiva de ordenar cartas en la mano: se toma una carta (elemento de la lista) y se la coloca en la posición adecuada respecto a las cartas ya ordenadas.

El algoritmo divide conceptualmente la lista en dos secciones: una ordenada (al inicio, compuesta solo por el primer elemento) y otra desordenada (el resto de la lista). A medida que se avanza, se toma un elemento de la sección desordenada y se lo inserta en el lugar correspondiente dentro de la sección ordenada,

desplazando los elementos mayores si es necesario. Este proceso se repite hasta que todos los elementos han sido procesados, y la lista queda completamente ordenada.





Código en Python:

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

Ventajas y Desventajas

Ventajas:

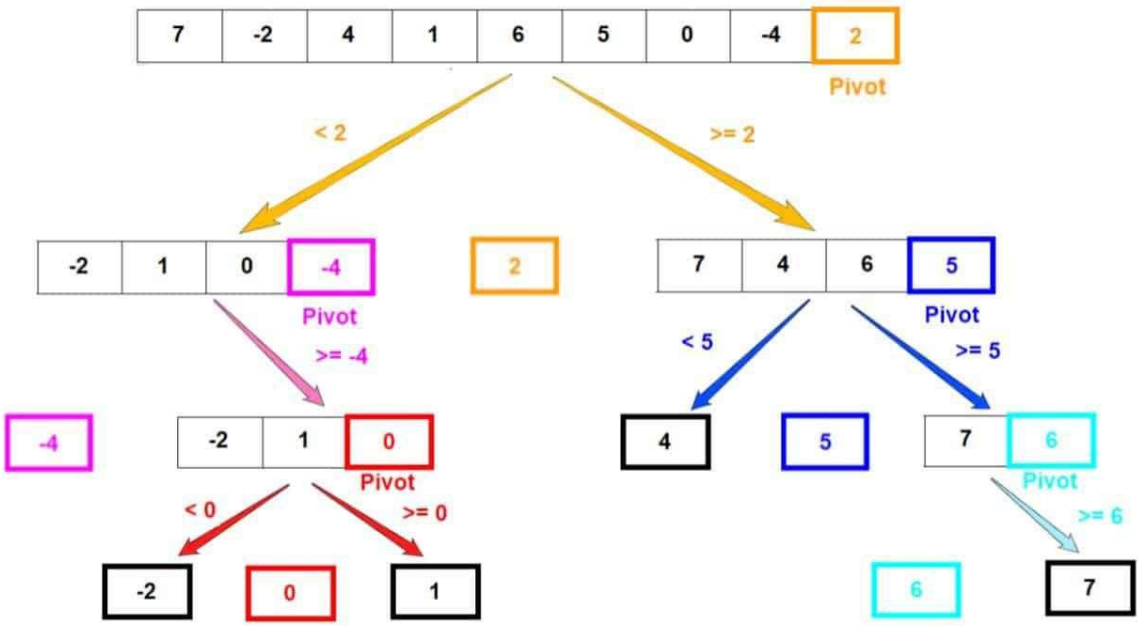
- Baja sobrecarga: Requiere menos comparaciones e intercambios, lo que mejora su eficiencia en listas pequeñas o parcialmente ordenadas.
- Simplicidad: fácil de entender e implementar.

Desventajas:

- Ineficiente en listas grandes: Su rendimiento disminuye considerablemente con el tamaño de la lista debido a su complejidad cuadrática
- Poca escalabilidad: No es adecuado para grandes volúmenes de datos, ya que su tiempo de ejecución crece rápidamente a medida que aumenta la cantidad de elementos.

1.5 Ordenación rápida (Quicksort)

Es uno de los algoritmos de ordenamiento más eficientes y ampliamente utilizados. Se basa en la estrategia de **divide y vencerás**: selecciona un elemento pivote y reorganiza la lista de manera que todos los elementos menores queden a su izquierda y los mayores a su derecha. Luego, aplica recursivamente este mismo proceso a las sublistas resultantes, hasta que toda la lista esté ordenada.



Código en Python:

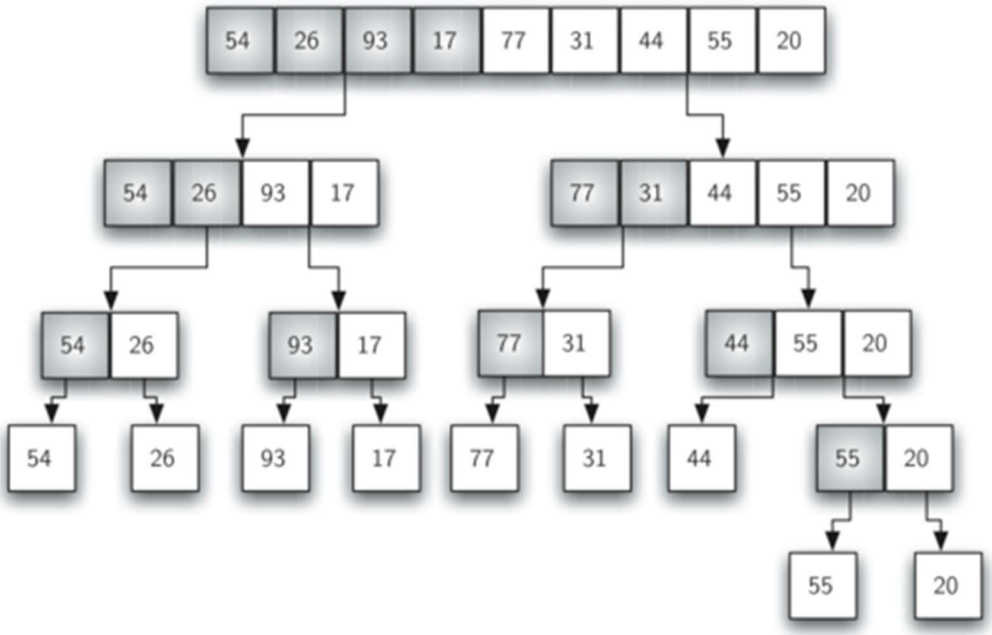
```
def quick_sort(lista):
    if len(lista) <= 1:
```

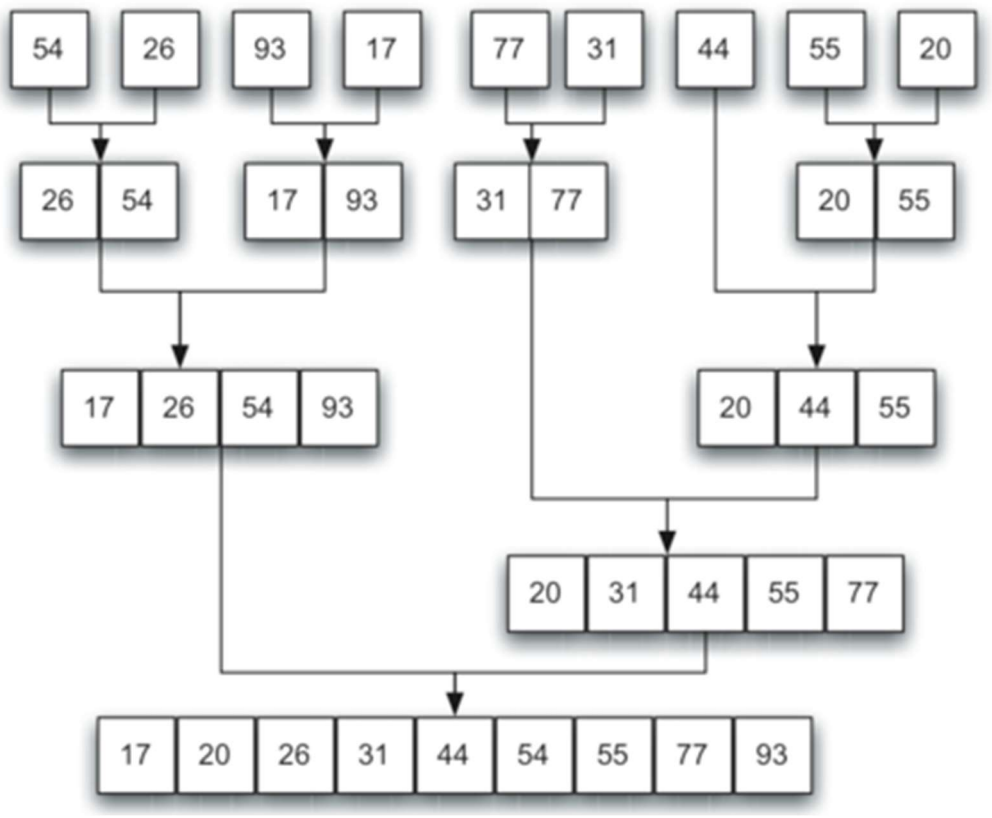
```
        return lista
    pivote = lista[0]
    menores = [x for x in lista[1:] if x <= pivote]
    mayores = [x for x in lista[1:] if x > pivote]
    return quick_sort(menores) + [pivote] + quick_sort(mayores)
```

1.6 Ordenamiento por mezcla (MergeSort)

Es un algoritmo eficiente que aplica la estrategia de **divide y vencerás**. Consiste en dividir recursivamente la lista en mitades hasta obtener sublistas de un solo elemento (que por definición están ordenadas). Luego, estas sublistas se **fusionan** de forma ordenada, comparando los elementos de cada par y combinándolos en una lista mayor también ordenada.

Este proceso continúa hasta que todas las sublistas se han combinado, resultando en una lista completamente ordenada.





Código en Python:

```python
def merge_sort(lista):
    if len(lista) <= 1:
        return lista
    mitad = len(lista) // 2
    izquierda = merge_sort(lista[:mitad])
    derecha = merge_sort(lista[mitad:])
    return merge(izquierda, derecha)
def merge(izq, der):
    resultado = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
```

```
            resultado.append(izq[i])
            i += 1
        else:
            resultado.append(der[j])
            j += 1
    resultado.extend(izq[i:])
    resultado.extend(der[j:])
    return resultado
```

Comparación de Ordenamientos

Ordenamiento	Naturaleza	Ventaja	Desventaja	Complejidad
Burbuja	Interno, iterativo	Simple de implementar, bueno para listas pequeñas	Consume bastante tiempo de computadora Requiere muchas lecturas/escrituras en memoria.	$O(n^2)$
Inserción	Interno, iterativo	Eficiente en listas pequeñas o casi ordenadas	Poco eficiente con listas grandes o muy desordenadas	$O(n^2)$
Selección	Interno, iterativo	Fácil de entender y programar	No es estable, y siempre hace el mismo número de comparaciones	$O(n^2)$
Rápido (QuickSort)	Interno, Recursivo	Muy rápido en la práctica, ampliamente usado	Implementación algo confusa. Realiza numerosas comparaciones e intercambios.	$O(n \log n)$
Por mezcla (MergeSort)	Interno, Recursivo	Muy eficiente y **estable**, ideal para listas grandes	Requiere memoria adicional	$O(n \log n)$

Conclusión

La elección del algoritmo de ordenamiento depende del tamaño y estado de la lista:

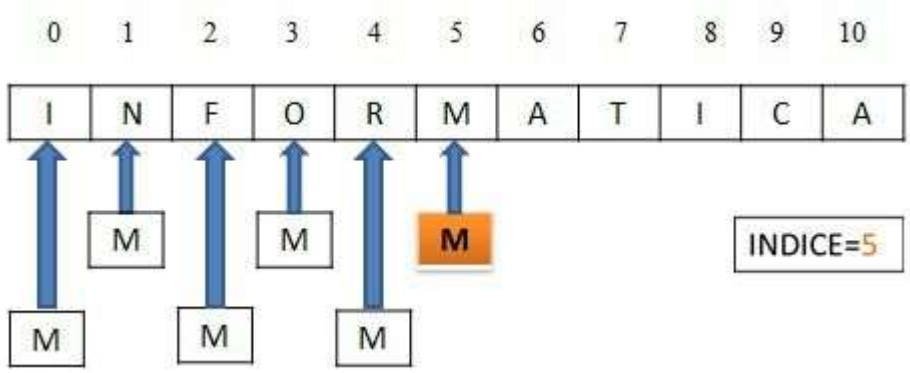
- En listas pequeñas o casi ordenadas, algoritmos como burbuja o inserción son suficientes y fáciles de implementar.
- Para listas grandes, se recomienda usar QuickSort o MergeSort:
    - QuickSort es muy eficiente, pero su rendimiento depende de una buena elección del pivote.
    - MergeSort es más estable y predecible, aunque requiere más memoria.

2. Algoritmo de búsqueda

Un algoritmo de búsueda es un procedimiento sistemático diseñado para localizar un valor objetivo dentro de una colección de datos, ya sea retornando la posición del elemento, el elemento mismo o indicando que no se encuentra. Los métodos más comunes son:

2.1 Búsqueda Lineal

Los algoritmos de búsqueda lineal implica recorrer una lista de elementos uno por uno hasta encontrar un elemento específico.



Código en Python:

```python
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i  # Devuelve la posición donde se encuentra
    return -1  # No se encontró
```

Ventajas y Desventajas

Ventajas:

- Sencillez: Búsqueda más simple y fácil de implementar. Solo requiere iterar a través de la lista de elementos uno por uno hasta encontrar el objetivo.
- Flexibilidad: Puede aplicarse a cualquier tipo de lista, independientemente de si está ordenada o no.

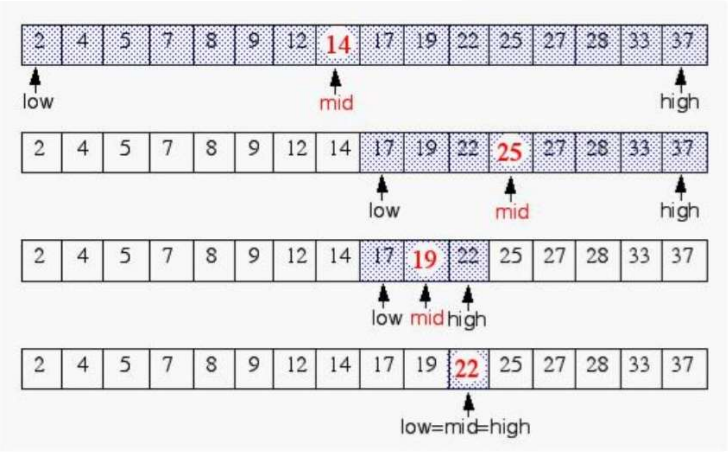
Desventajas:

- Ineficiencia en listas grandes: Debido a que compara cada elemento uno por uno, su tiempo de ejecución crece de manera lineal con el tamaño de la lista.
- No es adecuada para listas ordenadas: Aunque puede funcionar en listas no ordenadas, no es eficiente para listas ordenadas.

2.2 Búsqueda Binaria

El algoritmo de búsqueda binaria es un algoritmo muy eficiente que se aplica solo a listas ordenadas. Funciona dividiendo repetidamente la lista en dos mitades y comparando el elemento objetivo con el elemento del medio, esto reduce significativamente la cantidad de comparaciones necesarias.

Hallar elemento 22



Código en python

```python
def busqueda_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
```

Ventajas y Desventajas del Algoritmo de Búsqueda Binaria

Ventajas:

- Eficiencia de listas ordenadas: La principal ventaja de la búsqueda binaria es su eficiencia en listas ordenadas. Su tiempo de ejecución es de O(log n), lo que significa que disminuye rápidamente a medida que el tamaño de la lista aumenta.

- Menos comparaciones: Comparado con la búsqueda lineal, la búsqueda binaria realiza menos comparaciones en promedio, lo que lo hace más rápido para encontrar el objetivo.

Desventajas:

- Requiere una lista ordenada: La búsqueda binaria sólo es aplicable a listas ordenadas, Si la lista no está ordenada, se debe realizar una operación adicional para ordenarla antes de usar la búsqueda binaria.
- Mayor complejidad de implementación: Comparado con la búsqueda lineal, la búsqueda binaria es más compleja de implementar debido a su naturaleza recursivaLos algoritmos de búsqueda y ordenamiento constituyen una de las bases fundamentales de la programación y las ciencias de la computación. Su correcta implementación y elección influye directamente en el rendimiento de los programas, especialmente cuando se trabaja con grandes volúmenes de datos.

Comparación de búsquedas

	Lineal	Binaria
Orden del vector	No	Si
Definición	Recorre todos los elementos uno por uno hasta encontrar el objetivo o llegar al final.	Divide la lista ordenada a la mitad en cada paso, descartando la mitad irrelevante.
Simplicidad	Muy fácil de implementar.	Más compleja, especialmente en versión recursiva.
Tipo de recorrido	Secuencial, de principio a fin.	Divide y conquista, con cortes a la mitad.
Número de	Hasta n comparaciones en el peor	Hasta $\log_2(n)$ comparaciones en el

comparaciones	caso.	peor caso.
Complejidad temporal	O(n) – tiempo lineal.	O(log n) – tiempo logarítmico.
Robustez ante modificaciones	No requiere preprocesamiento.	Requiere orden previo; se vuelve ineficiente si se modifica constantemente la lista.

Conclusión

La búsqueda lineal es ideal para listas pequeñas o desordenadas, mientras que la búsqueda binaria es la opción más eficiente en listas ordenadas y grandes. La elección adecuada depende del tipo de datos, su ordenamiento y la frecuencia con la que se necesita realizar búsquedas.

Caso práctico

1. **Descripción del problema**

En este proyecto se desarrolló un programa en Python para simular un sistema básico de búsqueda de productos a partir de sus códigos numéricos. Para ello, se genera una lista aleatoria de 150 códigos entre 0 y 999. Luego, se utiliza el algoritmo de ordenamiento Burbuja para organizar la lista, y posteriormente se aplica una búsqueda binaria para localizar códigos específicos ingresados por el usuario.

El sistema realiza dos operaciones principales:

1. Ordena la lista de códigos utilizando el algoritmo de Burbuja (Bubble Sort).
2. Permite buscar códigos específicos en la lista ordenada aplicando búsqueda binaria.

Objetivos del programa

- Generar una lista aleatoria de 150 códigos de productos.
- Ordenar la lista utilizando un algoritmo comprensible y fácil de seguir.
- Permitir la búsqueda rápida de un código específico.
- Visualizar el tiempo de ordenamiento y búsqueda.

Herramientas implementadas

- **Biblioteca timeit**: Permite medir el tiempo exacto que tarda el programa en ordenar la lista y realizar las búsquedas.
- **Biblioteca random**: Se usa para generar los códigos aleatorios del 0 al 999.

Decisiones de Diseño

Decidimos usar el algoritmo de ordenamiento en Burbuja (Bubble Sort) para ordenar la lista de códigos, no solo porque nos pareció sencillo sino que también fácil de entender y aplicar. Aunque no es el más rápido para listas grandes, en este caso estamos trabajando con 150 códigos de productos, siendo una cantidad estándar o mediana. Esto hizo que el tiempo que tarda sea aceptable y nos permite ver cómo funciona el ordenamiento paso a paso.

Y la elección de binaria, nos pareció una búsqueda rápida que buscar uno por uno. Ya que su funcionamiento al dividir la lista en partes y descartando donde no está, haciendo que sea más eficiente, especialmente cuando la lista ya está ordenada.

Por eso, aunque el ordenamiento Burbuja no sea el más rápido, una vez que la lista está ordenada, la búsqueda binaria nos permite encontrar cualquier código de forma rápida y sin tener que revisar cada elemento.

2. **Código (Python)**

```
import timeit
import random
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]
    return arr

def busqueda_binaria(arr, objetivo):
    inicio = 0
    fin = len(arr) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if arr[medio] == objetivo:
            return True
        elif arr[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return False

codigos = [random.randint(0, 999) for _ in range(150)]

print("\nLista desordenada:")
print(codigos)

inicio_ordenamiento = timeit.default_timer()
codigos_ordenados = bubble_sort(codigos.copy())
fin_ordenamiento = timeit.default_timer()
tiempo_ordenamiento = fin_ordenamiento - inicio_ordenamiento

print("\nLista ordenada:")
print(codigos_ordenados)
print(f"Tiempo de ordenamiento (Bubble Sort): {tiempo_ordenamiento:.8f}
segundos\n")

while True:
    entrada = input("Ingrese el código de producto a buscar (o escriba
'salir' para terminar): ")
    if entrada.lower() == 'salir':
```

```
        print("Gracias por usar el sistema.")
        break
    if not entrada.isdigit():
        print("Por favor, ingrese un número válido.\n")
        continue

    codigo_a_buscar = int(entrada)

    inicio_busqueda = timeit.default_timer()
    encontrado = busqueda_binaria(codigos_ordenados, codigo_a_buscar)
    fin_busqueda = timeit.default_timer()
    tiempo_busqueda = fin_busqueda - inicio_busqueda

    print(f"¿El código {codigo_a_buscar} fue encontrado?: {'Sí' if
encontrado else 'No'}")
    print(f"Tiempo de búsqueda: {tiempo_busqueda:.8f} segundos\n")

            print("Por favor, ingrese un número válido.\n")
            continue
        codigo_a_buscar = int(entrada)
        inicio_busqueda = timeit.default_timer()
        encontrado = busqueda_binaria(codigos_ordenados,
codigo_a_buscar)
        fin_busqueda = timeit.default_timer()
        tiempo_busqueda = fin_busqueda - inicio_busqueda
        print(f"¿El código {codigo_a_buscar} fue encontrado?: {'Sí' if
encontrado else 'No'}")
        print(f"Tiempo de búsqueda: {tiempo_busqueda:.8f} segundos\n")
```

3. Validación del Funcionamiento

Se comprobó que el algoritmo de ordenamiento Burbuja funciona correctamente y es capaz de ordenar de forma adecuada la lista de productos, compuesta por códigos numéricos de uno a tres dígitos. Una vez ordenada la lista, se realizaron múltiples búsquedas, tanto con códigos que estaban presentes como con otros que no figuraban.

En todas las pruebas, el programa respondió de manera adecuada: los códigos existentes fueron encontrados sin inconvenientes, y los códigos ausentes fueron descartados rápidamente. Esto demuestra que el proceso de ordenamiento y la búsqueda binaria están operando correctamente y de forma eficiente.

Metodología Utilizada

Para llevar a cabo el programa, detallamos las siguientes etapas importantes:

1. **Generación de datos aleatorios**

Se utilizó la **biblioteca random** para crear una lista de 150 códigos numéricos entre 0 y 999. Esto nos permitió simular una lista de códigos numéricos, de manera automática.

2. **Ordenamiento de la lista**

Se aplicó el algoritmo de **ordenamiento en burbuja** (Bubble Sort) sobre la lista generada.

Al tratarse de una lista pequeña (150 elementos), el rendimiento sigue siendo aceptable.

3. **Búsqueda de elementos**

Una vez ordenada la lista, se implementó la **búsqueda binaria** para encontrar códigos específicos. El usuario puede ingresar un código a buscar y el programa devuelve si fue encontrado o no, junto con el tiempo que tardó en buscarlo.

4. **Medición de tiempos**

Se usó la **biblioteca timeit** para calcular con precisión cuánto tarda el sistema en:

- Ordenar la lista con ordenamiento burbuja.
- Buscar un código con búsqueda binaria.

Esto permitió tener una idea clara del rendimiento en un entorno controlado.

5. **Validación**

Se realizaron pruebas con distintos códigos: algunos presentes en la lista y otros que no.

En todos los casos, el sistema respondió correctamente, confirmando que tanto el ordenamiento como la búsqueda funcionan de manera esperada.

Resultados Obtenidos

Los resultados obtenidos fueron:

1. Lista de 150 códigos generados de manera aleatoria, mostrándola en pantalla de forma desordenada.

2. La aplicación del algoritmo de ordenamiento en burbuja, la lista fue ordenada de menor a mayor y mostrada en pantalla.
3. El tiempo que llevó ordenar la lista, fue prácticamente baja.
4. Durante la búsqueda, al ingresar diferentes códigos, el programa fue capaz de determinar rápidamente si el código estaba presente o no en la lista.
5. El tiempo de búsqueda con la técnica de búsqueda binaria fue muy bajo, lo que nos confirma su eficiencia en la localización de los elementos de la lista.
6. Se realizaron búsquedas de códigos que estaban en la lista y otros que no. El programa funcionó correctamente.

Estos resultados indican que el programa cumple con los objetivos propuestos: ordenar una lista desordenada y permitir búsquedas rápidas, mostrando también un buen manejo del tiempo de ejecución para ambos procesos.

Conclusiones

Importancia:

El desarrollo de este trabajo permitió comprender en profundidad la importancia de los algoritmos de búsqueda y ordenamiento en el campo de la programación. Estos algoritmos no solo son pilares fundamentales en el procesamiento de datos, sino que también son clave para mejorar la eficiencia y rendimiento de los programas informáticos. Su correcta elección impacta directamente en la rapidez y precisión de los procesos computacionales.

Optimización:

Desde el punto de vista de la optimización, se evidenció cómo distintas técnicas pueden adaptarse a contextos específicos: algoritmos simples como el de burbuja o inserción son útiles para listas pequeñas o parcialmente ordenadas, mientras que métodos más avanzados como QuickSort o MergeSort ofrecen un rendimiento significativamente superior en grandes volúmenes de datos. Del mismo modo, la búsqueda binaria se posiciona como una opción altamente eficiente frente a la lineal, siempre que los datos estén previamente ordenados.

Aprendizaje:

En cuanto al aprendizaje, el trabajo nos brindó la oportunidad de afianzar conceptos clave del lenguaje Python, comprender la lógica algorítmica detrás de cada método, y desarrollar habilidades prácticas mediante la implementación y análisis de casos reales. La experiencia también nos ayudó a adoptar una mirada crítica sobre la aplicabilidad y eficiencia de distintas soluciones, promoviendo un enfoque más reflexivo y fundamentado en futuras decisiones de programación.

Bibliografía

Investigación:
https://medium.com/@vyankateshpareek733/sorting-algorithms-11a9057d688f

https://www.fing.edu.uy/tecnoinf/mvd/cursos/prinprog/material/teo/prinprog-teorico11.pdf

https://www.youtube.com/watch?v=u1QuRbx-_x4&ab_channel=Tecnicatura

https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoPorInsercion.html

https://medium.com/@Emmitta/b%C3%BAsqueda-binaria-c6187323cd72

https://4geeks.com/es/lesson/algoritmos-de-ordenamiento-y-busqueda-en-python?utm_source=chatgpt.com

https://colab.research.google.com/drive/1KVqiJSzYLTPDFRwTYjN8CP7G4LPreD9J?usp=sharing#scrollTo=PMcYh35HR3x-

https://runestone.academy/ns/books/published/pythoned/SortSearch/ElOrdenamientoPorSeleccion.html

Anexo

Link del repositorio de git:

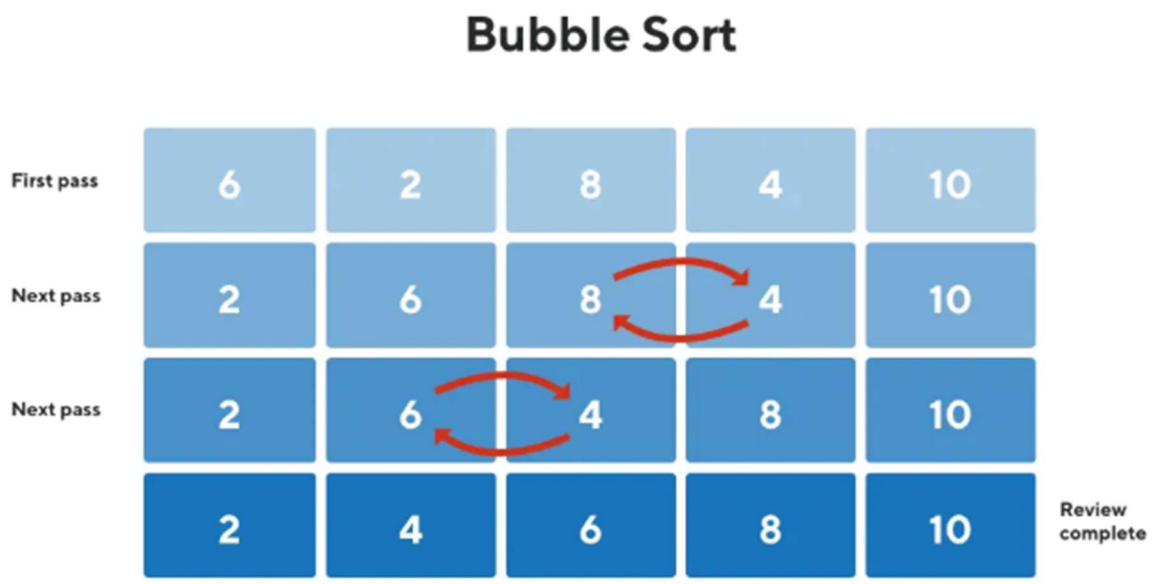
https://github.com/karenlauk/TPI.git

Link del video:

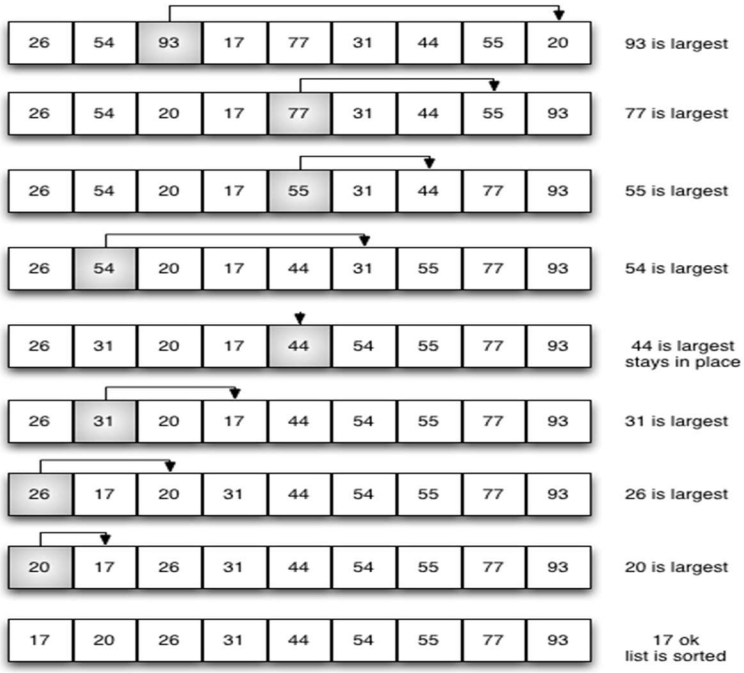
https://youtu.be/Cf0SpubPKA8

Capturas de pantalla:

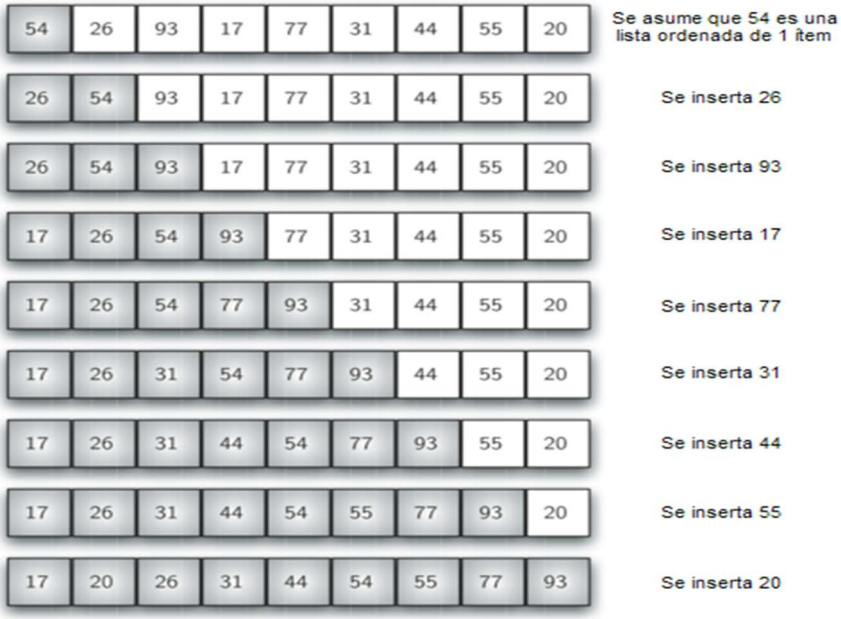
Ordenamiento de burbujas (Bubble Sort)

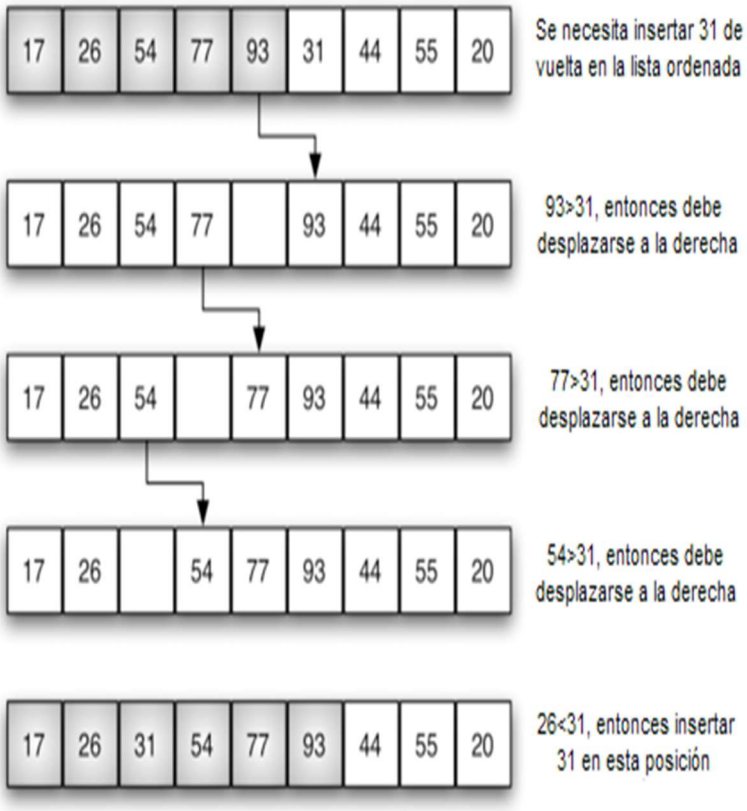


Ordenamiento por selección (Selection Sort)

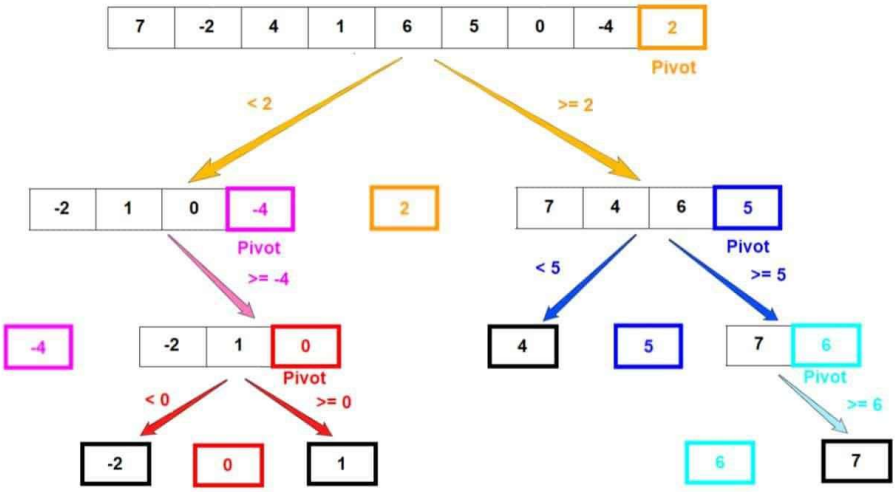


Ordenamiento por inserción

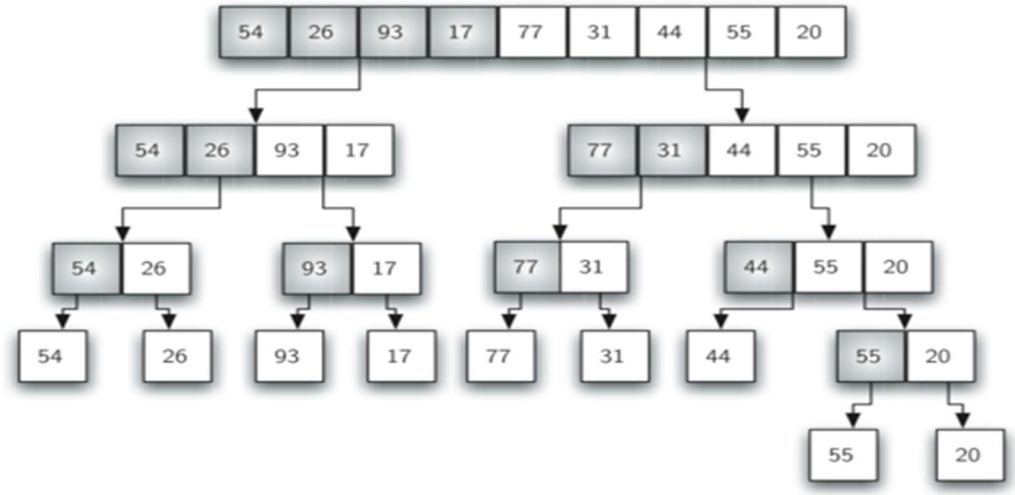


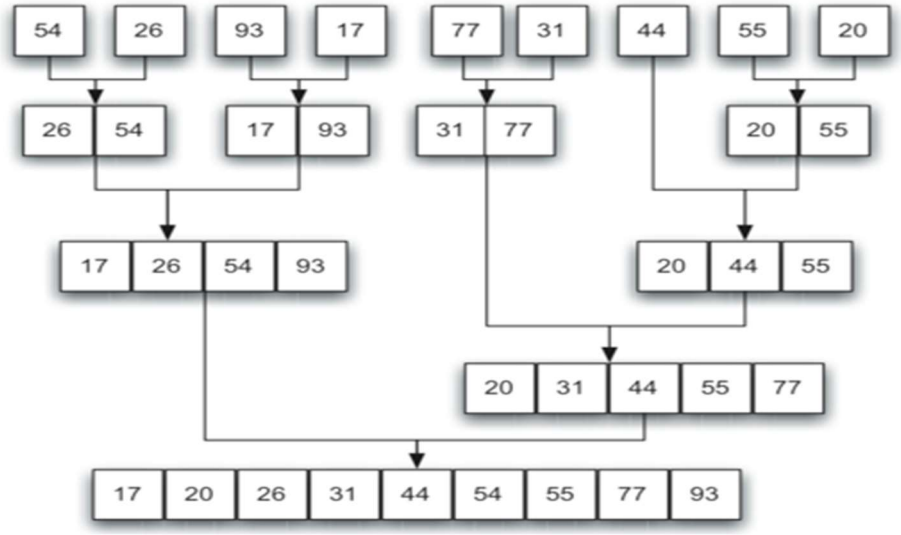


Ordenación rápida (Quicksort)

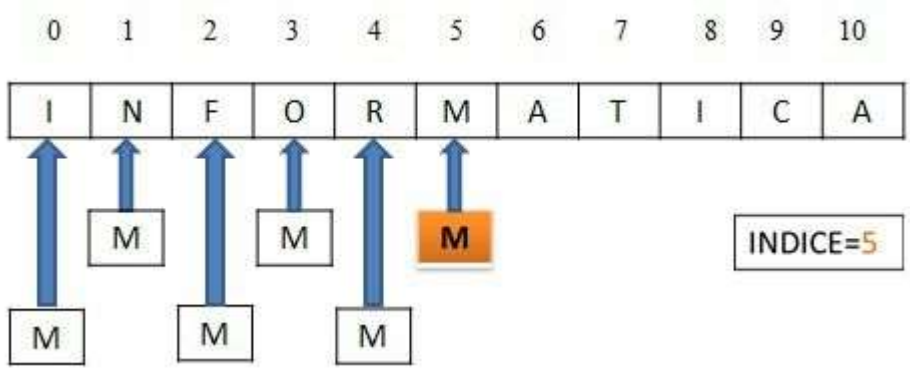


Ordenamiento por mezcla (MergeSort)



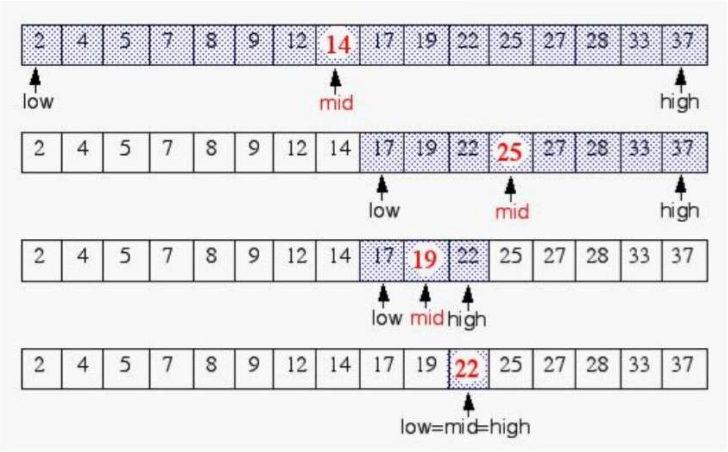


Búsqueda Lineal



Búsqueda Binaria

Hallar elemento 22



Códigos:

Ordenamiento de burbujas (Bubble Sort):

```
def bubble_sort(lista):

    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
```

Ordenamiento por selección (Selection Sort):

```python
def selection_sort(lista):
    n = len(lista)
    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if lista[j] < lista[min_idx]:
                min_idx = j
        lista[i], lista[min_idx] = lista[min_idx], lista[i]
    return lista
```

Ordenamiento por inserción:

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

Ordenación rápida (Quicksort):

```python
def quick_sort(lista):
    if len(lista) <= 1:
        return lista
    pivote = lista[0]
    menores = [x for x in lista[1:] if x <= pivote]
    mayores = [x for x in lista[1:] if x > pivote]
    return quick_sort(menores) + [pivote] + quick_sort(mayores)
```

Ordenamiento por mezcla (MergeSort):

```python
def merge_sort(lista):
    if len(lista) <= 1:
        return lista
    mitad = len(lista) // 2
    izquierda = merge_sort(lista[:mitad])
    derecha = merge_sort(lista[mitad:])
    return merge(izquierda, derecha)
def merge(izq, der):
    resultado = []
    i = j = 0
    while i < len(izq) and j < len(der):
        if izq[i] < der[j]:
            resultado.append(izq[i])
            i += 1
        else:
            resultado.append(der[j])
            j += 1
    resultado.extend(izq[i:])
    resultado.extend(der[j:])
    return resultado
```

Búsqueda Lineal:

```python
def busqueda_lineal(lista, objetivo):
```

```
    for i in range(len(lista)):
        if lista[i] == objetivo:
            return i  # Devuelve la posición donde se encuentra
    return -1  # No se encontró
```

Búsqueda Binaria:

```
def busqueda_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
```

Caso Práctico:

```
import timeit
import random
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]
    return arr

def busqueda_binaria(arr, objetivo):
    inicio = 0
    fin = len(arr) - 1
    while inicio <= fin:
        medio = (inicio + fin) // 2
        if arr[medio] == objetivo:
```

```
            return True
        elif arr[medio] < objetivo:
            inicio = medio + 1
        else:
            fin = medio - 1
    return False

codigos = [random.randint(0, 999) for _ in range(150)]

print("\nLista desordenada:")
print(codigos)

inicio_ordenamiento = timeit.default_timer()
codigos_ordenados = bubble_sort(codigos.copy())
fin_ordenamiento = timeit.default_timer()
tiempo_ordenamiento = fin_ordenamiento - inicio_ordenamiento

print("\nLista ordenada:")
print(codigos_ordenados)
print(f"Tiempo de ordenamiento (Bubble Sort): {tiempo_ordenamiento:.8f}
segundos\n")

while True:
    entrada = input("Ingrese el código de producto a buscar (o escriba
'salir' para terminar): ")
    if entrada.lower() == 'salir':
        print("Gracias por usar el sistema.")
        break
    if not entrada.isdigit():
        print("Por favor, ingrese un número válido.\n")
        continue

    codigo_a_buscar = int(entrada)

    inicio_busqueda = timeit.default_timer()
    encontrado = busqueda_binaria(codigos_ordenados, codigo_a_buscar)
    fin_busqueda = timeit.default_timer()
    tiempo_busqueda = fin_busqueda - inicio_busqueda

    print(f"¿El código {codigo_a_buscar} fue encontrado?: {'Sí' if
encontrado else 'No'}")
    print(f"Tiempo de búsqueda: {tiempo_busqueda:.8f} segundos\n")
```

```
            print("Por favor, ingrese un número válido.\n")
            continue
        codigo_a_buscar = int(entrada)
        inicio_busqueda = timeit.default_timer()
        encontrado = busqueda_binaria(codigos_ordenados,
codigo_a_buscar)
        fin_busqueda = timeit.default_timer()
        tiempo_busqueda = fin_busqueda - inicio_busqueda
        print(f"¿El código {codigo_a_buscar} fue encontrado?: {'Sí' if
encontrado else 'No'}")
        print(f"Tiempo de búsqueda: {tiempo_busqueda:.8f} segundos\n")
```