Trabajo Integrador - Programación

Mónica Martín - C17

Walter Joel Martinez - C17

# Algoritmos de búsqueda y ordenamiento

## Introducción

Los algoritmos de búsqueda y ordenamiento son fundamentales para el procesamiento eficiente de datos. Estos algoritmos permiten organizar la información de forma que se facilite su consulta, análisis o modificación, optimizando así el rendimiento.

Los algoritmos de ordenamiento tienen como objetivo organizar una colección de datos(como números, letras o registros) según un criterio específico, generalmente de menor a mayor o alfabéticamente. Existen distintos tipos de ordenamientos, como Bubble Sort, Selection Sort, Insertion Sort y Quick Sort, cada uno con sus ventajas, desventajas y niveles de eficiencia según el tamaño y la naturaleza de los datos.

Por otro lado los algoritmos de búsqueda permiten localizar un elemento específico dentro de una estructura de datos. Los más comunes son la búsqueda lineal y la búsqueda binaria.

# Algoritmo de búsqueda

Un algoritmo de búsqueda es un conjunto de pasos o instrucciones que permite localizar un elemento dentro de una estructura de datos, como una lista o arreglo. Su objetivo es determinar si el elemento está presente y, en caso afirmativo, devolver su posición(índice).

Hay distintos tipos de algoritmos, pero los más comunes son:

### Búsqueda lineal

Recorre la lista elemento por elemento desde el principio hasta el final y no requiere que la lista esté ordenada

```python
def busqueda_lineal(lista, objetivo):
    for i in range(len(lista)):  # Recorremos la lista completa
        if lista[i] == objetivo:  # Si encontramos el valor buscado
            return i  # Retornamos el índice donde se encuentra
    return -1  # Si no se encuentra, devolvemos -1
```

Ventaja: Es simple y funciona con cualquier tipo de lista

Desventaja: Es lento si la lista es muy grande, porque puede necesitar recorrer toda la lista.

### Búsqueda binaria

Divide la lista ordenada a la mitad repetidamente para encontrar el valor. Compara el elemento del medio con el buscado y decide si debe buscar a la izquierda o derecha

```python
def busqueda_binaria(lista, objetivo):
    inicio = 0
    fin = len(lista) - 1

    while inicio <= fin:
        medio = (inicio + fin) // 2  # Calculamos el punto medio
        if lista[medio] == objetivo:
            return medio  # Valor encontrado
        elif lista[medio] < objetivo:
            inicio = medio + 1  # Buscar en la mitad derecha
        else:
            fin = medio - 1  # Buscar en la mitad izquierda

    return -1  # Si no se encuentra, devolvemos -1
```

Ventaja: Mucho más rápida que la búsqueda lineal (complejidad O(log n))

Desventaja: Requiere que la lista esté ordenada previamente

## Algoritmo de ordenamiento

Un algoritmo de ordenamiento es un conjunto de instrucciones que permite organizar los elementos de una lista o arreglo según un determinado criterio, generalmente de menor a mayor o de mayor a menor.

Hay distintos tipos de algoritmos de ordenamiento, cada uno con sus propias ventajas, desventajas y niveles de eficiencia dependiendo del tipo de datos o tamaño de la lista.

Para los ejemplos utilizamos listas de enteros y el ordenamiento de menor a mayor.

### Ordenamiento por selección – SelectionSort

Busca el elemento más pequeño del arreglo y lo pone en la posición correcta, repitiendo el proceso con el resto de los valores de la lista.

```python
def selection_sort(lista):
    n = len(lista)  # Obtenemos la longitud de la lista
    # Vamos a recorrer cada posición de la lista
    for i in range(n):
        # Suponemos que el elemento actual es el más pequeño
        minimo = i
        # Buscamos en el resto de la lista si hay un valor menor
        for j in range(i+1, n):
            # Si encontramos un elemento más pequeño que el actual mínimo, lo guardamos
            if lista[j] < lista[minimo]:
                minimo = j
        # Intercambiamos el elemento más pequeño encontrado con el actual en la posición i
        lista[i], lista[minimo] = lista[minimo], lista[i]
        # Así, al final de cada iteración, dejamos el menor en su lugar correcto
    return lista  # Devolvemos la lista ordenada
```

Ventaja: Es fácil de entender e implementar. No requiere memoria adicional.
Desventaja: Es lento en listas grandes y realiza muchas comparaciones innecesarias, incluso si ya está ordenada.

Ordenamiento de burbuja – BubleSort

Compara elementos adyacentes e intercambia si están en orden incorrecto. Al final de cada pasada, el mayor queda al final.

```python
def bubble_sort(lista):
    n = len(lista)  # Obtenemos la longitud de la lista
    # Repetimos el proceso n veces para asegurar que todos los elementos se comparen
    for i in range(n):
        # En cada pasada, recorremos desde el inicio hasta el último elemento no ordenado
        for j in range(0, n-i-1):
            # Comparamos el elemento actual con el siguiente
            if lista[j] > lista[j+1]:
                # Si están en el orden incorrecto, los intercambiamos
                lista[j], lista[j+1] = lista[j+1], lista[j]
                # Este intercambio hace que los valores grandes "suban" al final de la lista
    # Repetimos esto hasta que toda la lista quede ordenada
    return lista
```

Ventaja: Muy fácil de implementar y entender. Y útil para listas pequeñas o casi ordenadas.

Desventaja: Es muy ineficiente en listas grandes y hace muchos pasos innecesarios si no se optimiza.

## Ordenamiento por inserción – InsertionSort

Recorre la lista e inserta cada elemento en la posición correcta respecto a los elementos anteriores ya ordenados.

```python
def insertion_sort(lista):
    # Empezamos desde el segundo elemento (el primero ya se considera ordenado)
    for i in range(1, len(lista)):
        key = lista[i]  # Guardamos el valor actual que queremos insertar
        j = i - 1  # Comenzamos a comparar con los elementos anteriores
        # Mientras no lleguemos al inicio y el valor anterior sea mayor a key
        while j >= 0 and lista[j] > key:
            # Desplazamos los elementos una posición a la derecha
            lista[j + 1] = lista[j]
            j -= 1  # Seguimos retrocediendo para seguir comparando
        # Insertamos el valor actual (key) en su posición correcta
        lista[j + 1] = key
        # Así, la parte izquierda de la lista siempre queda ordenada
    return lista
```

Ventaja: Muy eficiente para listas pequeñas o que ya están parcialmente ordenadas.
Desventaja: No es adecuado para listas grandes y desordenadas. Puede hacer muchos desplazamientos.

## Ordenamiento rápido – QuickSort

Utiliza un elemento pivote para dividir la lista en dos partes: menores y mayores. Ordena cada parte de forma recursiva.

```python
def quick_sort(lista):
    # Si la lista tiene un solo elemento o está vacía, ya está ordenada
    if len(lista) <= 1:
        return lista

    # Elegimos el primer elemento como pivote
    pivot = lista[0]

    # Creamos una lista con los elementos menores o iguales al pivote
    listaMenores = [x for x in lista[1:] if x <= pivot]

    # Creamos otra lista con los elementos mayores al pivote
    listaMayores = [x for x in lista[1:] if x > pivot]

    # Aplicamos recursivamente quick_sort a las sublistas y combinamos todo
    return quick_sort(listaMenores) + [pivot] + quick_sort(listaMayores)
```

Ventaja: Muy rápido y eficiente en la mayoría de los casos, con complejidad promedio.
Desventaja: Si el pivote se elige mal, puede volverse muy lento. Además usa recursividad, lo cual puede consumir mucha memoria.

Conclusión Se comprendió la importancia de seleccionar el algoritmo adecuado según el tipo y tamaño de los datos. Dejando evidenciado que:

● Los algoritmos simples como Bubble Sort o Búsqueda Lineal son útiles para casos pequeños o educativos, pero poco eficientes en la práctica

● Algoritmos como QuickSort y Búsqueda Binaria ofrecen mejoras significativas en rendimiento, especialmente con grandes volúmenes de datos

● Comprender la complejidad temporal es clave para tomar decisiones informadas sobre qué algoritmo aplicar

● El uso de funciones integradas en Python, como sorted o in para búsquedas, ofrece soluciones prácticas y optimizadas, pero conocer los algoritmos detrás ayuda a entender mejor su funcionamiento y limitaciones Este análisis permitió no solo aplicar los algoritmos, sino también evaluar sus ventajas, desventajas y contextos de uso, fomentando una visión crítica y técnica sobre su implementación.

## Otro enfoque

En el ejercicio adjunto de recetas organizadas para la publicación de un libro de cocina, se utiliza el método sort ( para ordenar una lista de elementos en un orden específico ya sea ascendente o descendente). Es muy eficiente y puede ordenar listas grandes de manera rápida.

En Python este método, ordena los elementos de la lista en vez de devolverlos ordenados; a su vez usa el algoritmo Tim Sort (algoritmo híbrido que  es estable, es decir mantiene el orden de los elementos iguales; en cuanto a la eficiencia, tiene complejidad temporal promedio de 0(n log n) ideal para listas de gran tamaño) como algoritmo de ordenamiento subyacente. Esto significa que cuando se usa  el método sort en una lista, se está usando Tim Sort para ordenar los elementos.

```
def ordenar_recetas(recetas):
    recetas.sort(key=lambda x: x.nombre)
```

Tim sort divide la lista en sublistas pequeñas, llamadas “runs” luego combina las sublistas ordenadas, repite el proceso de dividir y combinar las sublistas hasta que la lista completa esté ordenada. Se adapta bien a diferentes tipos de datos y patrones de ordenamiento

Aquí, el método sort se aplica a la lista de recetas y utiliza una función lambda como clave para ordenar los elementos según el atributo nombre .

https://github.com/monmar630/INTEGRADOR-P1.git