Algoritmos Fundamentales de Búsqueda y Ordenamiento: Análisis Teórico e Implementación en Python

Andres David Restrepo Hernandez

CEFIT

Estructuras de Datos y Algoritmos 3

Jorge Giraldo

Algoritmos Fundamentales de Búsqueda y Ordenamiento

La eficiencia en el manejo de datos es uno de los pilares de la computación moderna. Investigar los algoritmos fundamentales de búsqueda y ordenamiento permite comprender cómo se gestiona la información a bajo nivel. El objetivo de este trabajo es explicar y ejemplificar estos algoritmos utilizando el lenguaje de programación Python, analizando su funcionamiento lógico y su aplicación práctica.

1. Algoritmos de Búsqueda

Los algoritmos de búsqueda son procedimientos diseñados para localizar un elemento específico, conocido como clave o target, dentro de una estructura de datos (Cormen et al., 2009).

1.1 Búsqueda Lineal

La búsqueda lineal, también conocida como búsqueda secuencial, es el método más simple para encontrar un elemento. Consiste en recorrer la estructura de datos elemento por elemento desde el inicio hasta el final. Sirve principalmente para buscar datos en listas pequeñas o no ordenadas.

El funcionamiento paso a paso es sencillo: el algoritmo recibe una lista y un valor objetivo. Inicia un ciclo iterativo desde la primera posición (índice 0). En cada iteración, compara el elemento actual con el objetivo. Si coinciden, retorna la posición; si termina la lista sin éxito, indica que no se encontró.

Entre sus ventajas destaca su facilidad de implementación y que no requiere datos ordenados. Sin embargo, su desventaja es la ineficiencia en grandes volúmenes de datos, con una complejidad temporal de

$O(n)$.

Ejemplo en Python:

```python
def busqueda_lineal(lista, objetivo):
    # Recorro la lista obteniendo la posicion y el valor
    for indice, valor in enumerate(lista):
        # Compruebo si el valor actual es el que busco
        if valor == objetivo:
            return indice  # Lo encontre, devuelvo su posicion

    return -1  # Termine de recorrer y no estaba

# Pruebo mi funcion
mis_datos = [15, 2, 8, 10, 33, 1]
print(busqueda_lineal(mis_datos, 10))
```

1.2 Búsqueda Binaria

La búsqueda binaria es un algoritmo eficiente que requiere como condición necesaria que la lista esté ordenada. Aplica la técnica de "dividir y conquistar", reduciendo el espacio de búsqueda a la mitad en cada paso.

El proceso consiste en definir un puntero bajo y uno alto. Se calcula el punto medio; si el valor medio es el objetivo, se detiene. Si el objetivo es menor, se descarta la mitad derecha; si es mayor, se descarta la izquierda. Esto se repite hasta encontrar el valor.
Su principal ventaja es la velocidad, con una complejidad de
O(log n)

. Su desventaja es el requisito obligatorio de mantener la lista ordenada, lo cual tiene un costo computacional previo.

Ejemplo en Python:

```python
def busqueda_binaria(lista, objetivo):
    # Defino los punteros de inicio y fin
    bajo = 0
    alto = len(lista) - 1

    # Repito el ciclo mientras los punteros no se crucen
    while bajo <= alto:
        # Calculo el punto medio de la lista
        medio = (bajo + alto) // 2
        valor_medio = lista[medio]

        # Verifico si encontre el valor
        if valor_medio == objetivo:
            return medio
        # Si el valor es menor, busco en la mitad derecha
        elif valor_medio < objetivo:
            bajo = medio + 1
        # Si es mayor, busco en la mitad izquierda
        else:
            alto = medio - 1

    return -1  # El elemento no existe

# La lista debe estar ordenada para que esto funcione
datos_ordenados = [2, 5, 8, 12, 16, 23, 38]
print(busqueda_binaria(datos_ordenados, 23))
```

Algoritmos de Ordenamiento Básico

El ordenamiento organiza los elementos en una secuencia específica, facilitando operaciones futuras como la búsqueda (Goodrich et al., 2013).

2.1 Insertion Sort (Ordenamiento por Inserción)

Este algoritmo resuelve el problema de ordenar listas pequeñas construyendo la solución un elemento a la vez, similar a cómo se ordenan las cartas en una mano. Se toma un elemento de la parte desordenada y se compara con la parte ordenada, desplazando los elementos mayores a la derecha para hacer espacio e insertar el valor en su lugar correcto.

Ejemplo en Python:

```python
def insertion_sort(lista):
    # Empiezo a recorrer desde el segundo elemento
    for i in range(1, len(lista)):
        clave = lista[i]
        j = i - 1

        # Muevo los elementos que son mayores que mi clave
        # un espacio hacia adelante para hacer lugar
        while j >= 0 and lista[j] > clave:
            lista[j + 1] = lista[j]
            j -= 1

        # Pongo la clave en el hueco que quedo libre
        lista[j + 1] = clave

    return lista

numeros = [12, 11, 13, 5, 6]
print(insertion_sort(numeros))
```

2.2 Merge Sort

El Merge Sort es eficiente (O(n log n)) y utiliza el enfoque "divide y vencerás". Primero, divide recursivamente la lista en mitades hasta tener sublistas de un solo elemento. Luego, realiza la etapa de mezcla (merge), donde compara los elementos de las sublistas y los une ordenadamente en una nueva lista. Aunque es rápido, su limitación es que requiere memoria adicional para las listas temporales. Ejemplo en Python:

```python
def merge_sort(lista):
    # Caso base: si la lista tiene 0 o 1 elemento ya esta ordenada
    if len(lista) <= 1:
        return lista

    # Divido la lista en dos mitades
    medio = len(lista) // 2
    izquierda = merge_sort(lista[:medio])
    derecha = merge_sort(lista[medio:])

    return mezclar(izquierda, derecha)

def mezclar(lista_a, lista_b):
    ordenada = []
    i = 0
    j = 0

    # Comparo los elementos de ambas listas y agrego el menor
    while i < len(lista_a) and j < len(lista_b):
        if lista_a[i] < lista_b[j]:
            ordenada.append(lista_a[i])
            i += 1
        else:
            ordenada.append(lista_b[j])
            j += 1

    # Si sobran elementos en alguna lista, los agrego al final
    ordenada.extend(lista_a[i:])
    ordenada.extend(lista_b[j:])

    return ordenada

datos = [38, 27, 43, 3, 9, 82, 10]
print(merge_sort(datos))
```

2.3 Quick Sort

Quick Sort es un algoritmo que ordena "en el lugar" mediante particionamiento. Se escoge un elemento llamado pivote. El funcionamiento del particionamiento consiste en mover todos los elementos menores que el pivote a su izquierda y los mayores a su derecha. Luego se aplica recursivamente. La elección del pivote es crítica para su rendimiento (Bhargava, 2016).

Ejemplo en Python:

```python
def quick_sort(lista):
    # Verifico si la lista es pequena para retornar
    if len(lista) <= 1:
        return lista

    else:
        # Saco el ultimo elemento para usarlo como pivote
        pivote = lista.pop()

        # Creo dos listas nuevas
        mayores = []
        menores = []

        # Clasifico los numeros segun si son mayores o menores al pivote
        for item in lista:
            if item > pivote:
                mayores.append(item)
            else:
                menores.append(item)

        # Ordeno recursivamente y uno todo al final
        return quick_sort(menores) + [pivote] + quick_sort(mayores)

array = [10, 7, 8, 9, 1, 5]
print(quick_sort(array))
```

Referencias

Bhargava, A. (2016). Grokking Algorithms: An illustrated guide for programmers and other curious people. Manning Publications.

Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.

Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2013). Data Structures and Algorithms in Python. John Wiley & Sons.

Python Software Foundation. (2023). Python 3.11.0 Documentation. Recuperado de https://docs.python.org/3/