Trabajo Integrador: Algoritmos de Búsqueda y Ordenamiento

Joaquin Balaguer
TUPaD, Universidad Tecnológica Nacional
Programación I
Cinthia Rigoni
Oscar Londero
9 de junio de 2025

Algoritmos de Búsqueda y Ordenamiento

En este trabajo integrador nos proponemos ahondar en los conceptos de ordenamiento y búsqueda en el contexto de los algoritmos.

Marco Teórico

Algoritmos de Ordenamiento

Ordenar es el proceso de reorganizar una secuencia de objetos para ponerlos en un determinado orden lógico (Wayne & Sedgewick, 2011). Es ese proceso el que define a los algoritmos de ordenamiento.

Son muchos los motivos por los cuales estudiar y comprender estos algoritmos es importante, dos de ellos son:

- El ordenamiento sirve como base para la construcción de muchos otros algoritmos (de búsqueda, por ejemplo, como veremos más adelante). Entender cómo funcionan nos permite resolver más problemas y de manera más eficiente.
- Nociones importantes en el diseño de algoritmos aparecen en el estudio de los métodos de ordenamiento. Así, este tipo de algoritmos nos sirven como introducción a conceptos como “divide y vencerás” (división de un problema complejo en subproblemas más sencillos), estructuras de datos (formas de organizar los datos almacenados) o complejidad computacional (cantidad de tiempo que lleva ejecutar un algoritmo).

Selection Sort

Uno de los algoritmos de ordenamiento más sencillos es el llamado “Selection Sort”. Este funciona de la siguiente manera: intentando ordenar de menor a mayor, buscamos en el arreglo el elemento más pequeño y lo colocamos en la primera posición. Luego, buscamos el segundo elemento más pequeño y lo colocamos en la segunda posición. Repetimos este proceso hasta tener el arreglo entero ordenado. A medida que vamos iterando, nos va quedando una parte del arreglo ordenada:

comenzaremos a recorrer el arreglo a partir de la parte desordenada, ahorrándonos comparaciones innecesarias con los que se suponen son los elementos ya en orden. Para visualizarlo mejor, veamos una tabla que ilustra cómo sería el proceso de “Selection Sort” en un arreglo de 5 elementos:

Arreglo ordenado	Arreglo desordenado	Elemento más pequeño en el Arreglo desordenado
[]	[22, 43, 50, 9, 31]	9
[9]	[22, 43, 50, 31]	22
[9, 22]	[43, 50, 31]	31
[9, 22, 31]	[43, 50]	42
[9, 22, 31, 43]	[50]	50
[9, 22, 31, 43, 50]	[]	

Este algoritmo de ordenamiento es sencillo de aplicar y de entender, y que puede resultar una solución conveniente para listas o arreglos pequeños, donde el uso de algoritmos más complejos no estaría quizás justificado.

Analizar su complejidad es sencillo. Para encontrar el mínimo, debemos recorrer los n elementos de un arreglo y compararlos con el primero, para después hacer el intercambio. Esto nos llevaría un total de n - 1 comparaciones. Para encontrar el siguiente elemento más pequeño, debemos recorrer los n - 1 elementos restantes y hacer el intercambio. En este caso, esto nos llevaría un total de n - 2 comparaciones. Así se repetirá hasta que hayamos terminado de ordenar nuestro arreglo, resultando el total de comparaciones en:

$$(n-1) + (n-2) + \ldots + 1 = \sum_{i=1}^{n-1}$$

Lo que equivale a:

$$\sum_{i=1}^{n-1} i = \frac{(n-1)+1}{2}(n-1) = \frac{1}{2}n(n-1) = \frac{1}{2}(n^2 - n)$$

Identificando el término de mayor crecimiento, podemos concluir que el Selection Sort es un algoritmo con Big O de n^2 o O(n^2)

Conocer esto es importante para reafirmar lo que dijimos con anterioridad: se trata de un algoritmo que puede resultar eficiente para listas con pocos elementos. Debido a su complejidad, se ve rápidamente superado por algoritmos más complejos y sofisticados como los son QuickSort o HeapSort (con Big O de (n log n) en la mayoría de los casos).

Aun así, Selection Sort no deja de ser una solución y un algoritmo sencillo, que nos sirve como introducción al ordenamiento.

Algoritmos de Búsqueda

La computación moderna nos ha dado acceso a mucha cantidad de información. Resulta fundamental, entonces, ser capaces de buscar de manera eficiente esos datos para poder procesarlos. No solo es importante tener almacenada la información, si no que también disponer de ella. Es aquí cuando los algoritmos de búsqueda cobran relevancia. Se tratan de algoritmos cuyo objetivo es encontrar un determinado elemento dentro de una estructura de datos, cualquiera sea.

Binary Search

El Binary Search o algoritmo de búsqueda binaria es un algoritmo rápido y eficiente para buscar en un conjunto de elementos ordenados. Para hacer la búsqueda de un elemento *q* en un arreglo *S* con *n* elementos, el algoritmo compara a *q* con el elemento en *S*[*n*/2] (justo en el medio). Si *q* es mayor, entonces deberá encontrarse en la parte superior del arreglo (de *S*[*n*/2] + 1 hasta *S*[*n*]); caso contrario, deberá

encontrarse en la parte inferior (de *S*[0] hasta *S*[*n*/2] - 1). Si resulta igual, entonces *q* está en el medio. Este proceso se repetirá de manera recursiva, descartando de a mitades hasta dar con *q*.

En código Python se veria tal que asi:

```python
def binary_search(arr, x):
    limite_inferior = 0
    limite_superior = len(arr) - 1
    mid = 0

    while limite_inferior <= limite_superior:

        mid = (limite_superior + limite_inferior) // 2

        # Si x es mayor, ignoramos la mitad izquierda
        if arr[mid] < x:
            limite_inferior = mid + 1

        # Si x es menor, ignoramos la mitad derecha
        elif arr[mid] > x:
            limite_superior = mid - 1

        # El elemento esta en el medio
        else:
            return mid

    # Si salimos del ciclo while, el elemento no se encontro
    return -1

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = int(input("Ingrese el elemento a buscar: "))
```

```
resultado = binary_search(arr, x)

if resultado != -1:
    print(f"El elemento esta presente en la posicion {str(resultado)}")
else:
    print("El elemento no esta presente en el arreglo")
```

Como vemos, con cada iteración actualizamos los valores limite_inferior y limite_superior, de acuerdo a si el valor de ‘x’ es mayor o menor en comparación con ‘mid’ (la posición mediana). De esa manera, la próxima vez que entremos al bucle, haremos la pregunta en la nueva mitad. El ciclo ‘while’ se interrumpe una vez hayamos encontrado el elemento o si el valor de limite_inferior supera al de limite_superior, señal que hemos terminado de recorrer el arreglo sin éxito.

El método de búsqueda binaria contrasta con el método de búsqueda linear. Con el primero, como hemos visto, dividimos el arreglo en subarreglos más pequeños hasta encontrar el elemento buscado. Con el segundo, recorremos posición por posición hasta encontrarlo. La búsqueda binaria es sin duda más eficiente, con una complejidad de O(log n) en comparación con una de O(n). Sin embargo, la búsqueda binaria requiere una condición previa: que la lista se encuentre ordenada.

Caso Práctico

En este trabajo integrador decidimos desarrollar dos programas en los que aplicamos el “Selection Sort” en uno y la “Binary Search” en el otro.

En el primer programa, “selection_sort.py”:

```
def selection_sort(arr):
    n = len(arr)
    for i in range(n - 1):
```

```
        # Suponemos como elemento mas pequeño al que se encuentra en la posicion
actual
        min_idx = i

        # Iteramos sobre la porcion desordenada del arreglo, buscando el elemento mas
pequeño
        for j in range(i + 1, n):
            if arr[j] < arr[min_idx]:

                # Actualizamos la posicion del indice si encontramos un elemento mas
pequeño
                min_idx = j

        # Movemos el elemento mas pequeño a su posicion
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

def mostrar_array(arr):
    for val in arr:
        print(val, end=" ")
    print()

arr = [64, 25, 12, 22, 11]

print("Arreglo original: ", end="")
mostrar_array(arr)

selection_sort(arr)

print("Arrgelo ordenado: ", end="")
mostrar_array(arr)
```

Definimos una función “selection_sort()”, que recibe como parámetro un arreglo y lo ordena ascendentemente a través de bucles iterativos, aplicando el método del algoritmo homónimo.
Definimos también una función “mostrar_array()”, que recibe como parámetro un arreglo y lo muestra por pantalla.
En la función principal definimos un arreglo y lo mostramos por pantalla, invocando a “mostrar_array()”. Luego, invocamos “selection_sort()” para que lo ordene. Finalmente, lo volvemos a mostrar por pantalla, esta vez ya ordenado.
Salida por consola:

```
PS C:\Users\gauta> & C:/Users/gauta/AppData/Local/Microsoft/WindowsApps/python3.13.exe "d:/gauta/Documents/UTN TUP a distan
cia/(01) Primer año/(02) Programacion 1/TP Integrador Programacion I/selection_sort.py"
Arreglo original: 64 25 12 22 11
Arrgelo ordenado: 11 12 22 25 64
PS C:\Users\gauta>
```

En el segundo programa “binary_search.py”:

```python
def binary_search(arr, x):
    limite_inferior = 0
    limite_superior = len(arr) - 1
    mid = 0

    while limite_inferior <= limite_superior:

        mid = (limite_superior + limite_inferior) // 2

        # Si x es mayor, ignoramos la mitad izquierda
        if arr[mid] < x:
            limite_inferior = mid + 1

        # Si x es menor, ignoramos la mitad derecha
        elif arr[mid] > x:
            limite_superior = mid - 1
```

```
        # El elemento esta en el medio
        else:
            return mid

    # Si salimos del ciclo while, el elemento no se encontro
    return -1

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
x = int(input("Ingrese el elemento a buscar: "))

resultado = binary_search(arr, x)

if resultado != -1:
    print(f"El elemento esta presente en la posicion {str(resultado)}")
else:
    print("El elemento no esta presente en el arreglo")
```

Definimos una función “binary_search()” que recibe como parámetro un arreglo y el elemento a buscar. A través de la búsqueda binaria, determina la posición del elemento dentro del arreglo. En caso de encontrarlo, devuelve la posición; caso contrario, devuelve -1.

En la función principal, definimos nosotros un arreglo y le damos al usuario la opción de ingresar el elemento que desea buscar. Invocamos la función “binary_search()” y almacenamos el resultado en una variable “resultado”. De haber tenido éxito, el programa mostrará la posición en la que se encontró; de no haberlo tenido, se mostrará un mensaje “El elemento no está presente en el arreglo”.

Salida por consola al ingresar “2”:

```
PS C:\Users\gauta> & C:/Users/gauta/AppData/Local/Microsoft/WindowsApps/python3.13.exe "d:/gauta/Documents/UTN TUP a distan
cia/(01) Primer año/(02) Programacion 1/TP Integrador Programacion I/binary_search.py"
Ingrese el elemento a buscar: 2
El elemento esta presente en la posicion 1
PS C:\Users\gauta>
```

Metodologia Utilizada

La metodología utilizada fue del tipo cuantitativa. Desarrollamos dos pequeños programas en lenguaje Python, en los que definimos en uno un algoritmo de ordenamiento y en otro uno de búsqueda. Aplicamos un caso de prueba para evaluar los resultados y comprobar si eran los esperados.

Resultados Obtenidos

Logramos aplicar efectivamente un algoritmo de ordenamiento y otro de búsqueda. Trasladamos los conceptos de “Selection Sort” y “Binary Search” a código y los pusimos a prueba, obteniendo resultados satisfactorios.

Conclusiones

Comprender cómo funcionan los algoritmos de ordenamiento y de búsqueda aportan conocimientos esenciales para el diseño de algoritmos. A través de ejemplos concretos, cómo “Selection Sort” o “Binary Search”, hemos podido ahondar no solo en estos dos problemas fundamentales de la computación, cuya solución son la base para la solución de muchos otros problemas más complejos, si no que también en conceptos cruciales como lo es el de complejidad y eficiencia algorítmica.

Referencias

Kevin Wayne & Robert Sedgewick (2011, 24 de marzo). *Algorithms*. Addison-Wesley