ORDENAMIENTO Y BUSQUEDA

Maria C. Torres Madroñero
Profesora asociada
Departamento de Ciencias de la Computación y la Información

Objetivos

- Implementar los algoritmos básicos de ordenamiento y búsqueda binaria
- Aplicar los algoritmos de ordenamiento y búsqueda a las estructuras de datos estudiadas

Recursos requeridos

- PC
- IDE para JAVA o Python – el estudiante deberá seleccionar un lenguaje de programación para el desarrollo de las prácticas de laboratorio

Actividades preliminares al laboratorio

- Lectura de la guía

Marco teórico

ORDENAMIENTO

El problema de ordenamiento se encuentra frecuentemente en múltiples aplicaciones. Formalmente, podemos definir el problema de ordenamiento como:

INPUT: una secuencia de n números $< a_1, a_2, \dots a_n >$
OUTPUT: una permutación de la secuencia de entrada $< a'_1, a'_2, \dots\ a'_n >$ tal que $a'_1 \leq a'_2 \leq\ a'_n$

En la literatura podemos encontrar varios algoritmos de ordenamiento, que pueden variar su complejidad computacional. Existen algoritmos de ordenamiento más eficientes que otros, esto se ha logrado mediante la aplicación de técnicas de diseño de algoritmos o el uso de estructuras de datos.

INSERCION

Este algoritmo es muy intuitivo, en el sentido que ordena como lo hacen muchas personas con una mano de cartas. El algoritmo parte de una secuencia vacía O, se toma uno de los elementos de la secuencia a ordenar y se agrega de primero en la secuencia O. Consecutivamente se toma uno de los elementos de la secuencia a ordenar y se INSERTA en la posición correcta en la secuencia O. Este es el algoritmo que utilizamos de forma intuitiva para organizar los datos de nuestro tablero de puntajes. El seudocódigo se presenta a continuación:

```
InsertionSort(A,n)
1. FOR (i=1, i<n, i++)
2.     temp = A[i]
3.     j=i
4.     WHILE (j>0 & A[j-1]>temp)
5.           A[j] = A[j-1]
6.           j--
7.     A[j] = temp
```

BURBUJA
Ordena la colección mediante la comparación consecutiva de elementos vecinos en la secuencia, los cuales son intercambiados si es necesario. El algoritmo recorre desde la primera posición toda la secuencia, intercambiando si es necesario elementos adyacentes. Este recorrido mueve el valor mas alto al final de la colección. Este proceso se repite n veces para lograr que todos los elementos queden ordenados.

```
BubbleSort(A,n)
1. FOR (i=0, i<n, i++)
2.    FOR(j=1, j<n-i, j++)
3.       IF A[j-1] > A[j]
4.          INTERCAMBIAR(A, j-1, j)
```

SELECCION
Ordena una colección colocando repetidamente un valor en su posición final. Primero, buscar el mínimo valor y lo intercambia con el elemento que esta de primero. Luego buscar el siguiente mínimo y lo intercambia con el segundo. Este proceso se repite secuencialmente hasta lograr el ordenamiento.

```
SelectionSort(A,n)
1. FOR (i=0, i<n, i++)
2.      minIndex = i
3.      FOR(j=i+1, j<n, j++)
4.         IF A[j] < A[minIndex]
5.            minIndex = j
6.      INTERCAMBIAR(A, i, minIndex)
```

MERGESORT
A diferencia de los algoritmos anteriores, MERGESORT fue diseñado a partir de una estrategia de diseño de algoritmos denominada DIVIDIR Y CONQUISTA. Esta estrategia tiene tres etapas: la primera divide el problema en pequeñas partes; la segunda soluciona de forma independiente cada problema; y la tercera, combina las soluciones para obtener la solución. Específicamente para el algoritmo de MERGESORT, el problema se divide en dos partes, de forma recursiva se aplica el mismo algoritmo a la primera mitad y a la segunda mitad de forma independiente, posteriormente se combina la solución de las dos mitades.

```
Merge_Sort(A,p,r)
 1. if p < r
 2.    q = ⌈(p + (r − p))/2⌉
 3.    Merge_Sort(A,p,q)
 4.    Merge_Sort(A,q+1,r)
 5.    MERGE(A,p,q,r)
```

BUSQUEDA

Por su parte, el problema de búsqueda se puede expresar como:

INPUT: un valor objetivo x y una secuencia de entrada
OUTPUT: ubicación del objetivo en la secuencia de entrada

El objetivo puede ser un valor exacto que se encuentre en la secuencia, o se puede generalizar a cualquier conjunto de datos, donde x represente una clave o atributo asociado al objetivo. Si estamos buscando en un arreglo la salida será el índice (posición en la que se encuentra) o si es una lista usualmente será el nodo.

En el caso de arreglos y listas, el algoritmo mas sencillo es una búsqueda lineal. La búsqueda lineal recorre toda la colección de principio a fin, y compara cada elemento de la colección con el valor objetivo:

```
find_in_Array(A[],x)
1. FOR(i=0, i<n, i++)
2.    if A[i]==x
3.       inx = i
4. RETURN inx

find_in_Lits(L,x)
1. temp=L.First()
2. FOR(i=0, i<L.size(), i++)
3.     if temp.getData()==x
4.        n = temp
5.     temp=temp.getNext()
6. RETURN n
```

En los arreglos, tenemos una alternativa mas eficiente para realizar la búsqueda. El algoritmo de búsqueda binaria soluciona el problema con una complejidad de O(log n) cuando el arreglo esta ordenado. El algoritmo consiste en comparar el valor objetivo con el valor de la mitad; si el valor es menor, se continua la búsqueda de forma recursiva en la mitad izquierda de arreglo; si el valor es mayor, se realiza la búsqueda recursiva en la mitad derecha:

```
BinarySearch(A[],x,iLeft,iRight)
1. IF iLeft>iRight
2.    RETURN -1
3. ELSE
4.    mid = (iLeft+iRight)/2
5.    IF x < A[mid]
6.       RETURN BinarySearch(A[], x, iLeft, mid-1)
7.    ELSEIF x >  A[mid]
8.       RETURN BinarySearch(A[], x, mid+1, iRight)
9.    ELSE
10.      RETURN mid
```

ACTIVIDADES

1. Implementar una clase ORDENADOR que permita aplicar los métodos de burbuja, selección, inserción y mergesort a un arreglo. La clase ORDENADOR se describe en el diagrama de clase:

ORDENADOR
- A: int[ ] - limit: int
+ Ordenador(int capacity) + inicializar() + ordenar_burbuja() + ordenar_seleccion()

+ ordenar_insercion() + ordenar_mergeSort() + mostrar() + búsqueda_binaria(): int

a) La clase debe permitir inicializar un arreglo de valores enteros aleatorios. El parámetro capacity del constructor indica el numero de elementos a contener en el arreglo.
b) El método inicializar() debe permitir volver a llenar el arreglo con valores enteros aleatorios.
c) Los métodos ordenar_burbuja(), ordenar_seleccion(), ordenar_insercion(), y ordenar_mergeSort() deben implementar los métodos estudiados en clase.
d) El método mostrar() debe permitir imprimir en pantalla el arreglo A.
e) El método búsqueda_binaria debe permitir buscar uno de los valores del arreglo y retornar el indice

2. Implementar una clase ORDENADOR_LISTA que permita aplicar uno de los métodos de ordenamiento a una lista simple:

ORDENADOR_LISTA
- L: List
+ Ordenador_Lista() + inicializar(int n) + ordenar() + mostrar()

a) La clase debe permitir inicializar una lista simple valores enteros aleatorios.
b) El método inicializar llena la lista con valores aleatorios. El parámetro n indica el numero de elementos a incluir en la lista simple.
c) El método ordenar() debe implementar uno de los métodos estudiados en clase para ordenar la lista de menor a mayor
d) El método mostrar() debe permitir imprimir en pantalla los valores enteros almacenados en la lista.

3. Implementar una clase ORDENADOR_AGENDA que permita aplicar uno de los métodos de ordenamiento a una lista doble que contenga USUARIOS:

ORDENADOR_AGENDA
- L: DoubleList
+ Ordenador_Agenda() + agregarUsuario(Usuario u) + ordenar() + mostrar()

a) El método agregarUsuario() debe permitir agregar Usuario a la lista. Cada usuario debe tener la información empleada en el Laboratorio 2.
b) El método ordenar() debe implementar uno de los métodos estudiados en clase para ordenar la lista doble de menor a mayor de acuerdo con el numero de la cedula.
c) El método mostrar() debe permitir imprimir en pantalla la información de los usuarios almacenados en la lista doble.

4. Validar el funcionamiento de sus clases con los siguientes problemas:
   a) Para la clase ORDENADOR
      Paso 1: Crear un objeto tipo ORDENADOR con una capacidad de 10: O = new ORDENADOR(10)
      Paso 2: Inicializar el arreglo con valores aleatorios O.inicializar()
      Paso 3: Mostrar los valores del arreglo O.mostrar()
      Paso 4: Aplicar el algoritmo de ordenamiento por burbuja O.ordenar_burbuja()
      Paso 5: Mostrar los valores del arreglo O.mostrar()
      Repetir pasos 2 a 5 para cada algoritmo de ordenamiento

Paso 6: Aplicar el método de búsqueda binaria par un valor del arreglo

b) Para la clase ORDENADOR_LISTA
   Paso 1: Crear un objetivo tipo ORDENADOR_LISTA: O = new ORDENADOR_LISTA()
   Paso 2: Inicializa la lista con 12 valores aleatorios O.inicializar(12)
   Paso 3. Aplica el algoritmo de ordenamiento seleccionado O.ordenar()
   Paso 4. Muestra los valores de la lista en pantalla O.mostrar()
c) Para la clase ORDENADOR_AGENDA
   Paso 1: Crear un objeto O = new Ordenador_Agenda()
   Paso 2: Agregar 6 usuarios con toda la información empleada en el Lab 2 (puede leer los datos de un archivo de texto)
   Paso 3. Aplica el algoritmo de ordenamiento seleccionado O.ordenar()
   Paso 4. Muestra los valores de la lista en pantalla O.mostrar()

Instrucciones de entrega

- La solución de los problemas debe desarrollarse en JAVA o Python. Los estudiantes tendrán la libertad de seleccionar el lenguaje de programación y plataforma para presentar la solución de los problemas.
- La solución debe emplear librerías nativas y se invita a los estudiantes a no usar código descargado de internet. Los laboratorios están diseñados para practicar los fundamentos teóricos; entre más código escriba el estudiante más fácil será su comprensión de los temas de clase.
- La solución se puede presentar en grupos de hasta 3 estudiantes.
- La solución de los problemas debe entregarse y sustentarse en el aula de clase o en la hora de asesoría a estudiantes. No se reciben soluciones por correo electrónico.