Fundamentos de Estructuras de Datos
Taller No. 1
Arreglos - Métodos de Búsqueda y Ordenamiento

MSc. Carlos Andrés Sierra
Ingeniería de Sistemas
Corporación Universitaria Minuto de Dios

April 15, 2017

Para cada uno de los siguientes algoritmos, realice la implementación correspondiente en los lenguajes *C*, *Java*, y *Python*, y súbalos al GitHub que ha creado para los ejercicios del curso.

La implementación a realizar debe ser organizada, completamente documentada en el código, en donde para los algoritmos de ordenamiento se debe tener un solo archivo en cada lenguaje, y para los algoritmos de búsqueda otro archivo únicamente; cada algoritmo debe ser trabajado como una función. El usuario debe poder ingresar una serie de números separados por comas, y estos se entenderan como el conjunto de números del arreglo, y usando un menú en la consola debe poder seleccionar cual algoritmo utilizar.

1 Bubble Sort:

Este es uno de los algoritmos más simples de ordenamiento. Este algoritmo se basa en la comparación de elementos, particularmente entre parejas adyacentes, y si la pareja a comparar no está ordenada, simplemente se intercambian; el algoritmo concluye cuando al hacer todo el recorrido, y hacer todas las comparaciones entre vecinos adyacentes, no se requieren realizar más intercambios. Este algoritmo no es recomendable para arreglos con gran cantidad de datos, ya que tanto su caso promedio como su peor caso tienen un orden de crecimiento de $\Theta(n^2)$.

```
Data: A : Unsorted array of numbers
Result: A* : Sorted array of numbers
for i ← 0 to length(A) − 1 do
    swapped ← false
    for j ← 0 to length(A) − 1 do
        /* compare to adjacent elements */
        if array[j] > array[j + 1] then
            /* swap them */
            auxSwap ← array[j]
            array[j] ← array[j + 1]
            array[j + 1] ← swap
            swapped ← true
        end
    end
    /* if no number was swapped, the array is sorted now */
    if not swapped then
        break
    end
end
```

Algorithm 1: BubbleSort

2 Insertion Sort:

Algoritmo de ordenamiento basado en comparaciones en cada posición del arreglo. Este algoritmo funciona bajo una estrategia de programación dinámica, en donde se trabaja el arreglo como sub-listas, las cuales crecen en tamaño cada vez que están ordenadas; de esta forma, cada vez que se agregue un elemento a la lista, este solo deba ser ubicado una vez, debido a que el segmento de forma local está ordenado. Este algoritmo no se recomienda para grandes cantidades de datos a ordenar, ya que su orden de crecimiento está dado por $\Theta(n^2)$.

```
Data: A : Unsorted array of numbers
Result: A* : Sorted array of numbers
x → valueToInsert
index → holePosition
for i ← 1 to length(A) do
    /* select value to be moved */
    x ← A[i]
    index ← i
    /* locate hole position for the value to be inserted */
    while index > 0 and A[index - 1] > x do
        A[index] ← A[index - 1]
        index ← index - 1
    end
    /* insert the value at hole position */
    A[index] ← x
end
```

Algorithm 2: InsertionSort

3 Merge Sort:

Este es un algoritmo de ordenamiento que está basado en el paradigma de divide y vencerás. Este es considerado uno de los mejores algoritmos para ordenar elementos de un arreglo, puesto que en el peor de los casos el orden de crecimiento que tiene es de $\Theta(nlogn)$.La estrategia que maneja *Merge Sort* es simple: el arreglo se divide en dos partes por la mitad, y este proceso se repite hasta que se llegue a arreglos de tamaño 1; luego, cada una de las soluciones se combina de manera ordenada, obteniendo de manera emergente al final el arreglo total completamente ordenado.

```
Data: A : Unsorted array of numbers
Result: A* : Sorted array of numbers
if lenght(A) == 1 then
    /* array is already sorted */
    return A
else
    /* split in two parts */
    left_sub-array ← A[0] ... A[n / 2]
    right_sub-array ← A[(n / 2) + 1] ... A[n]
    /* sort each one of the parts */
    sortedL ← MergeSort( left_sub-array )
    sortedR ← MergeSort( right_sub-array )
    /* follow the stratefy divide and conquer */
    return Merge(sortedL, sortedR)
end
```

Algorithm 3: MergeSort

```
Data: A : Sorted array of numbers, B : Sorted array of numbers
Result: C : Sorted array of numbers that contains all elements of both A and B
l ← length(A) + length(B)
/* create C array */
C → Array of length l
indexA ← 0, indexB ← 0, indexC ← 0
while A and B have elements do
    if A[indexA] < B[indexB] then
        /* add element from A array */
        C[indexC] ← A[indexA]
        indexA ← indexA + 1
        indexC ← indexC + 1
    else
        /* add element from B array */
        C[indexC] ← B[indexB]
        indexB ← indexB + 1
        indexC ← indexC + 1
    end
end
/* one of A or B has still some elements */
while A has elements do
    C[indexC] ← A[indexA]
    indexA ← indexA + 1
    indexC ← indexC + 1
end
while B has elements do
    C[indexC] ← B[indexB]
    indexB ← indexB + 1
    indexC ← indexC + 1
end
return C
```

Algorithm 4: Merge

4 Quick Sort:

Este es uno de los algoritmos de ordenamiento más eficientes que existe (suele utilizarse con grandes conjuntos de datos, y su eficiencia tanto en casos promedio como en el peor caso es de $\Theta(nlogn)$), el cual consiste en una estrategia de divide y vencerás debido a que siempre parte el arreglo en pequeños sub-arreglos, y este proceso se repite de manera recursiva. Particularmente, en este algoritmo se usa la noción de pivote para definir la construcción de los sub-arreglos, en donde los valores más pequeños que el pivote van al primer sub-arreglo, y los mayores van al segundo sub-arreglo. Tradicionalmente, se selecciona el primer elemento del arreglo como el pivote, y de igual manera se seleciona en los sub-arreglos mientras se está haciendo la recursividad; valga aclarar que el pivote, luego de hacer el proceso de partición, ya se encuentra en el lugar que tendrá finalmente en el conjunto ordenado.

```
Data: A : Unsorted array of numbers
Result: A* : Sorted array of numbers
if length(A) == 1 then
    /* array A is already sorted */
    return A
else
    /* take first set element as a pivot */
    pivot ← A[0]
    for i ← 1 to length(A) do
        /* build both less and greater than pivot subarrays */
        if A[i] < pivot then
            less_subarray.add ← A[i]
        else
            greater_subarray.add ← A[i]
        end
        /* call recursion for each one of the subarrays, and concatenate the results */
        return QuickSort(less_subarray) + pivot + QuickSort(greater_subarray)
    end
end
```

Algorithm 5: QuickSort

5 Linear Search:

Este es un algoritmo bastante simple para realizar búsquedas en un conjunto de datos. En este tipo de búsqueda, se hace una exploración secuencial de todos los elementos comparando uno por uno con el valor que se desea buscar. Si se llega a encontrar el elemento, se retorna la ubicación (índice) en donde se encuentra el mismo; si no, se utiliza el marcado lógico -1 para indicar que el valor no se pudo encontrar en el conjunto. No es recomendado para buscar en grandes conjuntos de elementos, puesto que su orden de crecimiento es de $\Theta(n)$.

```
Data: A : Array of numbers, x : Value to be searched
Result: index : Index where the item is located
index ← -1
/* compare with all the elements of the array */
for i ← 0 to length(A) do
    if A[i] == value then
        index ← i
        break
    end
end
return index
```

Algorithm 6: Lineal Search

6 Binary Search:

Este es un algoritmo de búsqueda bastante rápido, cuya complejida de tiempo es de orden $\Theta(logn)$; la Búsqueda Binaria es otra de las técnicas que se basa en el principio de Divide y Vencerás. Sin embargo, existe una restricción importante con la Búsqueda Binaria, y es que el conjunto de elementos sobre el cual se desea hacer la búsqueda debe estar ordenado (la Búsqueda Lineal, por ejemplo, si funciona sobre conjuntos desordenados de datos). La Búsqueda Binaria indaga por un valor en particular comparando con el elemento en la mitad del conjunto ordenado; si el valor y el elemento coinciden, se retorna el índice correspondiente. En casos contrarios se tienen dos opciones: (i) si el elemento de la mitad es más grande que el valor que se está buscando, el nuevo rango de búsqueda será el sub-arreglo a la izquierda de la mitad, (ii) por otra parte, si el elemento de la mitad es más pequeño que el valor que se está buscando, el nuevo rango de búsqueda será el sub-arreglo a la derecha de la mitad. Este proceso se repite hasta encontrar el elemento, o hasta que el rango de búsqueda se vuelva un sub-arreglo de tamaño cero.

```
Data: A : Sorted array of numbers, x : Value to be searched
Result: index : Index where the item is located
lowerBound ← 0
upperBound ← length(A) - 1
index ← -1
/* if upper bound is less than lower bound, there is not a factible solution */
while lowerBound < upperBound do
    middlePoint ← (lowerBound + upperBound) / 2
    if x == A[middlePoint] then
        /* element x has been found */
        index ← middlePoint
        break
    else
        if x < A[middlePoint] then
            upperBound ← middlePoint - 1
        else
            lowerBound ← middlePoint + 1
        end
    end
end
if lowerBound == upperBound and A[lowerBound] == x then
    index ← lowerBound
end
return index
```

Algorithm 7: BinarySearch

```
Data: A : Sorted array of numbers, x : Value to be searched, lowerBound : Lower bound of the
      search space, upperBound: Upper bound of the search space
Result: index : Index where the item is located
middlePoint ← (lowerBound + upperBound) / 2
/* if upper bound is equal than lower bound, there is just one factible solution */
if lowerBound == upperBound then
    if x == A[middlePoint] then
        /* element x has been found */
        return middlePoint
    else
        return-1
    end
else
    if x == A[middlePoint] then
        return middlePoint
    else
        if x < A[middlePoint] then
            return BinarySearch_Recursive(A, x, lowerBound, middlePoint)
        else
            return BinarySearch_Recursive(A, x, middlePoint + 1, upperBound)
        end
    end
end
```

Algorithm 8: BinarySearch_Recursive

7 Interpolation Search:

Esta técnica de búsqueda es una mejora al algoritmo tradicional de Búsqueda Binaria. Este algoritmo trabaja con una función lineal de interpolación que busca de forma directa el índice candidato del elemento a buscar; además de la restricción de conjunto de datos ordenado, este algoritmo es altamente eficiente cuando la distribución de los elementos en el conjunto es bastante equitativa (por ejemplo, series matemáticas). Si

se hace una analogía a un evento de la vida cotidiana, sería como buscar en un directorio, en donde en vez de buscar en la mitad, se busca directamente en el índice para aproximar de mejor manera la ubicación de la información.

```
Data: A: Sorted array of numbers, x : Value to be searched
Result: index : Index where the item is located
lowerBound ← 0
upperBound ← length(A) - 1
index ← -1
/* if upper bound is less than lower bound, there is not a factible solution */
while lowerBound < upperBound do
    /* calculate middle point as an interpolation */
    middlePoint ← lowerBound + ((higherBound - lowerBound) / (A[higherBound] -
    A[lowerBound])) * (x - A[lowerBound])
    if x == A[middlePoint] then
        /* element x has been found */
        index ← middlePoint
        break
    else
        if x < A[middlePoint] then
            upperBound ← middlePoint - 1
        else
            lowerBound ← middlePoint + 1
        end
    end
end
if lowerBound == upperBound and A[lowerBound] == x then
    index ← lowerBound
end
return index
```

Algorithm 9: InterpolationSearch

Ayuda: Utilice los códigos de ejemplo en **GitHub**.