@BritoAlv @DavierSanBel

Índice

1. Cómo funciona el algoritmo **1**
- 1.1. Enunciado Particularizado . . . 1
- 1.2. El algoritmo . . . 1

2. Implementación **3**
- 2.1. Usando Swaps . . . 3
- 2.2. Usando Binary Search . . . 3

3. Complejidad del Algoritmo **5**
- 3.1. Swaps . . . 5
- 3.2. Binary Search . . . 6

4. Observaciones y Generalizaciones **8**
- 4.1. Qué hace falta para hacer un sort . . . 8

5. Referencias **9**

1. Cómo funciona el algoritmo

1.1. Enunciado Particularizado

Insertion Sort

Empecemsos con una versión **particularizada** de el problema: Dado un **array** de **int** ordenarlos de menor a mayor, usando el algoritmo de insertion sort.

1.2. El algoritmo

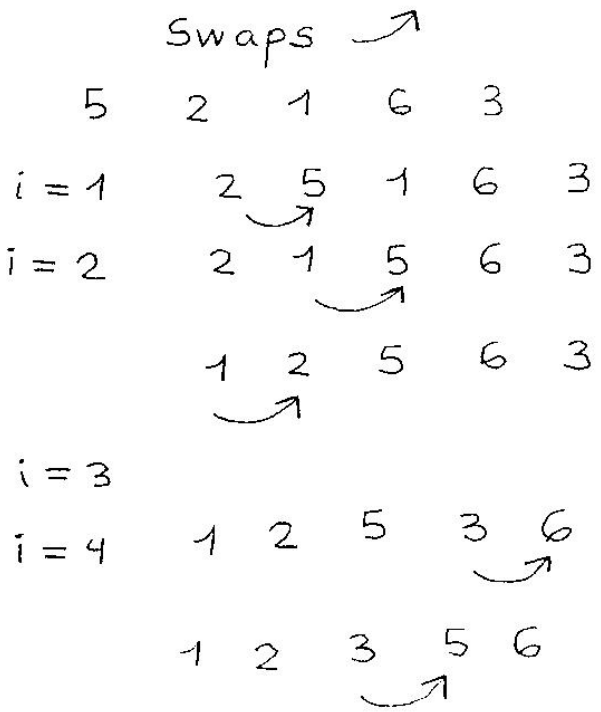
El algoritmo **recorre el array** desde la segunda posición ($i = 1$) hasta la última N, garantizando que al pasar por la posición i, el arreglo que va desde $[0, i]$ está **sorted**, notemos que inicialmente para $i = 0$ esto se cumple (porque es de longitud 1), el algoritmo mantiene esto invariante a medida que se va ejecutando y esto es su base.

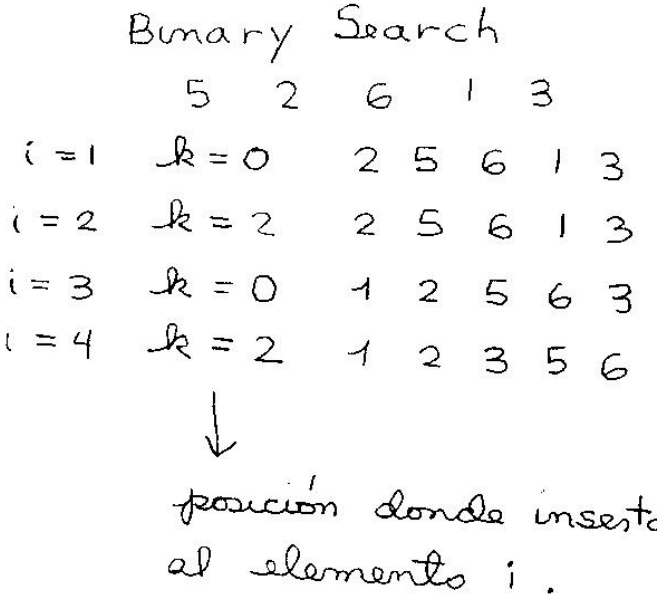
	5	2	1	6	3
i=1	2	5	1	6	3
i=2	1	2	5	6	3
i=3	1	2	5	6	3
i=4	1	2	3	5	6

	6	3	0	-1	-2
i=1	3	6	0	-1	-2
i=2	0	3	6	-1	-2
i=3	-1	0	3	6	-2
i=4	-2	-1	0	3	6

Existen dos formas de implementar el insertion sort. (¿qué ocurre dentro del for para **sort the array**?)

1. Usando **swap**, un swap es intercambiar dos elementos consecutivos, por lo que si queremos hallar donde va el elemento i en el subarreglo de $[0, i-1]$ comparamos y hacemos swap hasta llegar a la posición buscada.
2. Usando **binary search** para hallar la posición k donde se va a insertar y después shift los elementos a la derecha de k.





2. Implementación

2.1. Usando Swaps

```
public static void swaps_0(int[] source)
/*
The thing is that inside the while there are three
constant operations.
*/
{
    for (int i = 1; i < source.Length; i++) // n-1 iterations, so O(n)
    {
        int j = i;
        // the array from position 0 to j-1 is sorted always,
        // so now insert source[i] in that array, keeping it sorted.
        while((j > 0) && (source[j-1] > source[j]) )
        // has to go from i-1 to 0, checking, in each check,
        // it compares and after that do the swap.
        {
            int temp = source[j-1];
            source[j-1] = source[j];
            source[j] = temp;
            j--;
        }
    }
}
```

2.2. Usando Binary Search

```
public static void binary_search_0(int[] source)
{
    for (int i = 1; i < source.Length; i++) // n-1 iterations, so O(n)
    {
        // the array from position 0 to j-1 is sorted always, so now
        //insert source[i] in that array, keeping it sorted.
        // binary search to insert source[i] in correct position.
        // find index where to insert, and after that shift indexes.
```

```
int l = 0;
int r = i-1;
int mid = 0;
while(r-l > 1)
// as long as l-r > 1 we ensure that l < r and that mid is
// inside
{
    mid = (l+r)/2;
    if( source[mid] == source[i])
    {
        l = mid;
        r = mid;
    }
    else if ( source[mid] > source[i])
    {
        r = mid-1;
    }
    else
    {
        l = mid+1;
    }
}
// we have either l = r or r = l+1 bases cases of recursivity.
int pos;
if (source[i] <= source[l])
{
    pos = l; //
}
else if(source[i] > source[r] )
{
    pos = r+1; //
}
else
{
    pos = r;
}
int temp = source[i];
```

```
            // shift process
            for (int j = i; j >= pos+1; j--)
            {
                source[j] = source[j-1];
            }
            source[pos] = temp;
        }
    }
```

3. Complejidad del Algoritmo

Para calcular la complejidad del algoritmo, debemos calcular cuantas operaciones primitivas realiza en términos de N, usando los siguientes datos:

1. Inicializar un objeto de tipo **int** es en tiempo constante.
2. Random access en un array es en tiempo constante.
3. Comparar dos enteros es en tiempo constante.

Nota 1: El proceso de shift los elementos es linear respecto a la cantidad de elementos.

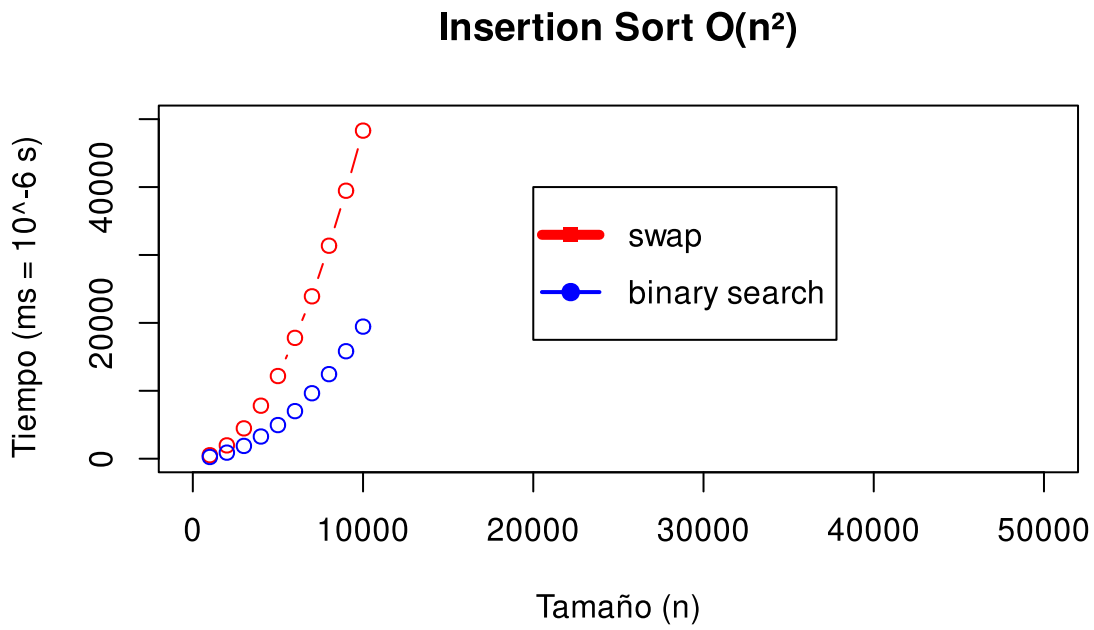
3.1. Swaps

Se inicializa i. Veamos para cada iteración i del **for** cuantas operaciones se realizan:

- se inicializa j.
- La mayor cantidad de veces que puede ser ejecutado el **while** es $j = i$ veces que es cuando siempre se cumple la segunda condición (todas las comparaciones dan true).
- Cada ejecución del **while** realiza 6 operaciones, acceder a `source[j-1]`, inicializar un **int** con ese valor, modificar los valores de `source[j-1]`, `source[j]`, acceder a `source[j]` y $j--$, como i va desde 1 a N, sería un total $\frac{6N(N+1)}{2}$ en el peor de los casos.

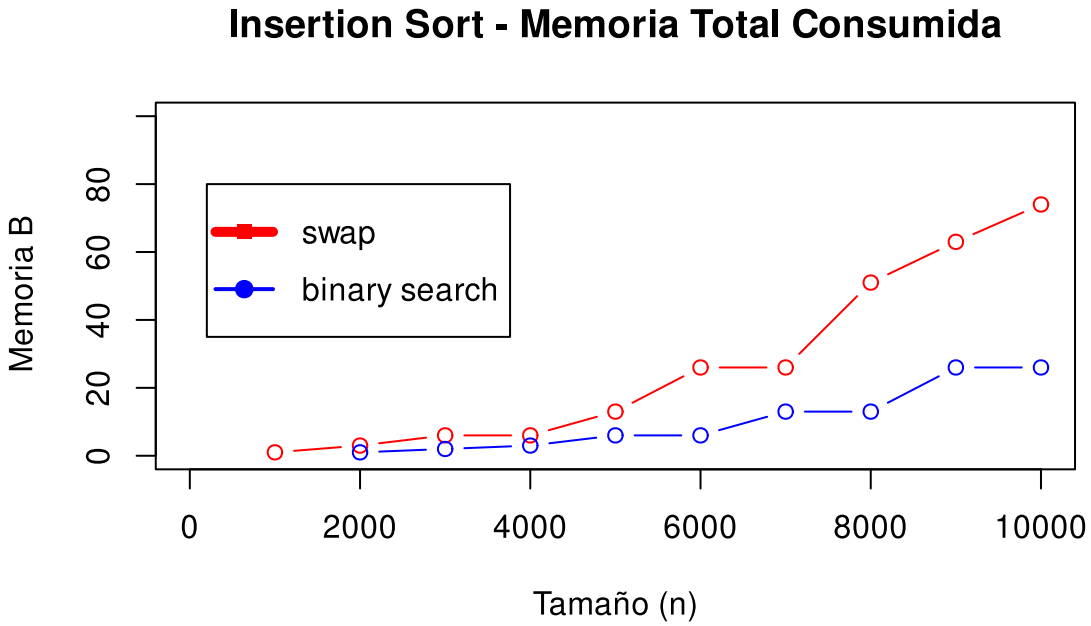
3.2. Binary Search

En el caso de binary search, sucede lo siguiente nuevamente contamos cuantas operaciones primitivas se realizan para cada i, en el **for**, la diferencia entre este método y el otro es que en este las comparaciones y actualizar el arreglo son dos procesos separados, **primero se halla la posición donde se va a insertar usando binary search y después se modifica el arreglo haciendo swaps**, esto reduce la cantidad de comparaciones a realizar, pero el **hecho de tener que realizar los mismos swaps mantiene la complejidad cuadrática del algoritmo**. Binary Search localiza la posición para cada i en $\log_2 i$ porque operaciones como **random access y comparar** son constantes, mientras que tiene que realizar a lo más i swaps (se mantiene como antes), por lo que antes teníamos i comparaciones y i swaps, ahora usando binary hay $\log_2 i$ comparaciones y i swaps. Como i domina sobre $\log_2 i$ la complejidad del algoritmo sigue siendo cuadrática.



El gráfico demuestra que ambos métodos looks like a parabola, pero en el caso de binary insertion sort, crece más lento.

Analizemos el uso de memoria, de estos algoritmos. Primero veamos, la máxima memoria que va a consumir el insertion sort. Podemos afirmar que el **insertion sort** es un algoritmo **in-place**, o sea, a medida que el se va ejecutando la máxima memoria que va a consumir es constante, **no depende de el tamaño de la entrada** (sola-



mente la variables temporales) y no utiliza otras estructuras de datos auxiliares para su funcionamiento. **Este análisis es respecto a la máxima memoria que pudiera necesitar el algoritmo, que por ser constante eso lo hace in-place.**

Por otro lado pudieramos analizar el total de memoria consumida por el algoritmo para ejecutarse, en el caso de los **swaps:**

```
{
    int temp = source[j-1];
    source[j-1] = source[j];
    source[j] = temp;
    j--;
}
```

Remember, that for each j, the swap method creates an temp variable for `source[j-1]`, en el peor de los casos este proceso se realizará $\frac{N(N+1)}{2}$ veces **(esto es temporal para cada j, para cada i, por eso no afecta in-place)**, mientras que en **binary search** se crea esta variable y otras más como `left, mid, pos, right` pero se crean para cada i, porque en el caso de la binary search, lo que se hace es **shift** los valores, que implica crear una cantidad **constante** de variables auxiliares.

Conclusión:

Respecto a los swaps: la complejidad de este método es cuadrática, y es directamente proporcional a la cantidad de comparaciones que son necesarias realizar, cada comparación verdadera implica hacer un swap, y hacer otra comparación, de esto se deduce que el peor de los casos es un array ordenado de mayor a menor, y el mejor de los casos es un array ordenado donde la complejidad es linear.

Respecto a Binary Search: still cuadratic due to swaps, but reduce the number of comparisons, also uses less memory and separates the process of finding the index.

En particular insertion sort is useful when la entrada de datos es pequeña, less than 25, for example or in cases where the input data is almost sorted, (need less number of comparisons).

4. Observaciones y Generalizaciones

4.1. Qué hace falta para hacer un sort

The default thing that comes to mind when sorting, is an array of int, but to sort some set of objects we only need that such objects have a definition de lo que significa ser menor que, en `CSharp, Icomparable`, y una estructura donde los vamos a almacenar, tal que se pueda iterar (tenga definido cual es su primer elemento, su segundo, ...). Un binary-search-tree no es iterable, por ejemplo:

En el caso de un **array de int** teníamos las siguientes ventajas .

1. Inicializar un objeto de tipo **int** es en tiempo constante.
2. Random access en un array es en tiempo constante.
3. Comparar dos enteros es en tiempo constante.

Ahora veamos el problema desde un punto de vista más general:

Generalized Insertion Sort

Dado un `array` de objetos de tipo T, tales que tienen definido un orden total o que el método `A.CompareTo(B)` está definido para la clase T, ordenar estos elementos de menor a mayor usando el método Insertion Sort, devolviendo una estructura E tal que $E[i]$ es el í-esimo menor elemento.

Veamos como variar (perder o ganar ventajas) la estructura de datos y el tipo de datos afecta en la perfomance (time and memory) of the sort. **O sea, analizar los casos límites.**

1. Inicializar objetos de tipo `matriz cuadrada de enteros de tamaño` N no es constante, ni el uso en memoria, ya que tiene que inicializar N^2 enteros.
2. Si utilizamos una linked list no tenemos random access.
3. Comparar dos elementos de tipo `matriz cuadrada de enteros de tamaño` N no es constante, si definimos como método de comparación la que posea mayor determinante.

5. Referencias

- **Orden Total:** Un orden total es una relación binaria $R : A \times A$ que es reflexiva, antisimétrica, transitiva y $\forall x, y \in A$ se cumple que $xRy \vee yRx$. El método `A.CompareTo(B)` de C# especifica que el método debe satisfacer esas condiciones. `https://docs.microsoft.com/en-us/dotnet/api/system.icomparable.compareto?view=net-6.0`