Análisis de Algoritmos

UNAM, Facultad De Ciencias.

Profesora. María de Luz Gasca Soto
Ayudante. Enrique Ehecatl Hernández Ferreiro
Ayudante. Brenda Margarita Becerra Ruíz

Tarea 6: Ordenamientos I

Alumna:
Villafán Flores María Fernanda (Ciencias de la Computación).

---

03 de noviembre de 2022
Ciudad de México.

1. El Problema de Selección consiste en encontrar el k-ésimo elemento más pequeño de un conjunto de n datos, $k \leq n$.
   Considere los algoritmos de ordenamiento:
   (a) Bubble Sort; (b)Insertion Sort;
   (c) Selection Sort; (d) Shell Sort;
   (e) Local Insertion Sort.

   Pregunta: ¿ Cuáles de las estrategias usadas por los algoritmos anteriores, nos ayudan a resolver el Problema de Selección **sin** tener que ordenar toda la secuencia? Suponga k tal que $1 < k < n$.
   Justifique, para cada inciso, sus respuestas.

Respuesta.

a) ***No nos ayuda*** porque la estrategia del algoritmo nos dice que compara los elementos de dos en dos, donde en cada iteración, el límite superior se reduce en uno. Es decir, al momento de ir comparando dos elementos únicamente "acerca" a los menores al inicio de la secuencia (que contendrá a los elementos del conjunto de n datos ordenados); pero **NO** puede distinguir en cual de todas las comparaciones que realiza se encuentra el k-ésimo elemento más pequeño, hasta que ordena el conjunto y entonces, puede indicar que es el primero.

b) ***No nos ayuda*** porque la estrategia del algoritmo nos dice que tenemos una secuencia ordenada a la cual le vamos agregando nuevos elementos y una vez hecho esto, se ordenará nuevamente la secuencia. Es decir, conforme vamos añadiendo elementos a la secuencia ordenada, podemos meter un elemento tal que sea menor al elemento que teníamos como el menor antes de realizar la inserción. Por lo que el algoritmo **NO** puede distinguir al k-ésimo elemento más pequeño; hasta terminar de ordenar la secuencia y entonces, indicar que es el primero.

c) ***Sí nos ayuda*** porque la estrategia del algoritmo nos dice que localiza al mínimo elemento del conjunto y lo intercambia con el ubicado en la primera posición, luego busca al segundo más pequeño y lo intercambia con el de la segunda localidad y así sucesivamente con todos los elementos. Es decir, desde la primera iteración del algoritmo ya encontramos al k-ésimo elemento más pequeño **SIN** necesidad de ordenar el conjunto.

d) ***No nos ayuda*** porque la estrategia del algoritmo nos dice que dividimos la secuencia en h subsecuencias y ordenamos por inserción cada una de ellas, donde esto se hace sucesivamente disminuyendo en cada iteración el valor de h hasta llegar a $h = 1$. Es decir, mientras vamos intercambiando los elementos involucrados en cada subsecuencia, estamos "acercando" a los elementos menores al inicio de la secuencia original; pero no es hasta que $h = 1$ que el algoritmo puede distinguir al k-ésimo elemento más pequeño (cuando prácticamente la secuencia original ya está ordenada).

e) ***No nos ayuda*** porque la estrategia del algoritmo nos dice que la clave es mantener una lista biligada L siempre en orden, así como un apuntador u al último elemento insertado y

a partir de u, se inicia la búsqueda de la posición correcta para insertar el nuevo elemento. Es decir, conforme vamos insertando nuevos elementos a la lista biligada con respecto al apuntador u, podemos llegar a insertar un elemento tal que sea menor al elemento que teníamos como el menor antes de realizar la inserción. Por lo que el algoritmo **NO** puede distinguir al k-ésimo elemento más pequeño hasta terminar de ordenar la lista y entonces, indicar que es el primero.

■

2. Problema Φ: Suponga que tiene n intervalos cerrados sobre la recta real: $[a(i), b(i)]$, con $1 \leq i \leq n$.
   Encontrar la máxima k tal que existe un punto x que es *cubierto* por los k intervalos.
   (a) Proporcione un algoritmo que solucione el problema Φ.
   (b) Justifique que su propuesta de algoritmo es correcta.
   (c) Calcule, con detalle, la complejidad computacional de su propuesta
   (d) Proporcione un pseudo-código del algoritmo propuesto.

Sin respuesta.

■

3. Realice la siguiente modificacin al algoritmo Insertion Sort:
   Para buscar la posición del nuevo elemento, el que está en revisión, usar Busqueda Binaria, en vez de hacerlo secuencialmente.
   a) Determine el desempeño computacional del algoritmo modificado.
   b) ¿Mejora el desempeño computacional del proceso total? Justifique.

Respuesta.

a) Primero, veamos la versión modificada del algoritmo de *Búsqueda Binaria*:

- ***Precondiciones:***
  - La secuencia S es una secuencia ordenada y finita de números enteros, donde $|S| = n$.
  - Los elementos de S son comparables.
- ***Algoritmo:***

```
// PreC: X[a,b] es un arreglo ordenado, no vacio y finito
         de enteros.
// PostC: X no sufre cambios;
```

```
            regresa la posicion i, que es donde debe
            ser insertado el elemento z.

int BBinaria(data_array X; int z; int a; int b) {
    int mitad                                   // variable temporal

    while (a ≤ b) {
        mitad = a + (b − a) / 2;                // calcula mitad
        if (z == X[mitad]) {                    // posicion correcta encontrada
            return mitad + 1;
        } else if (z > X[mitad]) {              // busca a la derecha
            a = mitad + 1;
        } else {                                // busca a la izquierda
            b = mitad − 1;
        }
    }
    return a;
} // end BBinaria.
```

- ***Postcondiciones:***

  El índice i donde el elemento z debe ser insertado.

Ahora, veamos la versión modificada del algoritmo de *Insertion Sort*:

- ***Precondiciones:***
  - La secuencia S es una secuencia finita de números enteros, donde $|S| = n$.
  - Los elementos de S son comparables.
- ***Algoritmo:***

```
// PreC: A es un arreglo que contiene a una secuencia S
        con n elementos.
// PostC: A esta en orden ascendente.

insertionSort(array A) {
    int n = A.length;
    int i, j, z, temp;

    for (i = 1; i < n; i++) {
        j = i − 1;
        temp = A[i];
        z = BBinaria(A, temp, 0, j);
        while (j >= z) {
            A[j+1] = A[j];
            j−−;
```

```
            } end while.
            A[j+1] = temp;
        } end for.
        return A;
    } // end InsertionSort.
```

- ***Postcondiciones:***
  La secuencia S ordenada de forma ascendente.

Así, tenemos que el desempeño computacional de la versión modificada de *Búsqueda Binaria* es $O(log_2(n))$, puesto que en cada iteración reducimos la longitud del espacio de búsqueda del índice correcto para insertar al elemento justo a la mitad. Además, la modificación que se hizo al algoritmo no hace que aumente el desempeño computacional porque únicamente se modificaron los valores que almacenan las variables *a, b* y *mitad*; además de cambiar la variable que regresa el algoritmo (estos cambios toman tiempo $O(1)$ en realizarse).

Ahora, como tenemos que la versión modificada de *Insertion Sort* utiliza el algoritmo modificado de *Búsqueda Binaria*, significa que disminuimos el número de comparaciones que se realizan para insertar un solo elemento de $O(n)$ a $O(log_2(n))$. Sin embargo, dado que el número de "swaps" realizados es el mismo, podemos decir que esta versión modificada del algoritmo toma el mismo tiempo $O(n^2)$ que la versión original de *Insertion Sort*.

b) Tenemos que el desempeño computacional del proceso total es $O(n^2)$, por lo tanto, no mejora. Aunque en la práctica, esta versión modificada del algoritmo de *Insertion Sort* funciona más rápido ya que hace menos comparaciones al momento de insertar nuevos elementos debido a que utiliza *Búsqueda Binaria*.

■

4. Indique si lo siguientes algoritmos de ordenamiento son **estables** o no y justifique por qué:
   (a) Bubble Sort; (b)Insertion Sort;
   (c) Selection Sort; (d) Shell Sort;
   (e) Local Insertion Sort.

Respuesta.

Primero, recordemos que un algoritmo es ***estable*** si durante el proceso de ordenamiento se preserva el orden relativo entre los elementos de la entrada original.

a) *Bubble Sort es un algoritmo estable* ya que sólo se intercambian dos elementos consecutivos en la secuencia si están desordenados, es decir, si en una iteración del algoritmo un elemento a es menor a un elemento b y b se encuentra primero que a en la secuencia, entonces se hace "swap" entre ellos. En caso contrario, los elementos se quedan en su posición hasta la siguiente iteración del algoritmo (esto también significa que se mantendrá el orden relativo entre elementos iguales).

b) *Insertion Sort es un algoritmo estable* ya que durante el proceso de inserción de un elemento a que está en la subsecuencia desordenada a la subsecuencia ordenada de la secuencia original, primero se busca su posición correcta en la subsecuencia ordenada y se hace "swap" entre los elementos, por lo que sólo se lleva a cabo este intercambio de elementos si un elemento b que se encuentra en la subsecuencia ordenada, es mayor al elemento a que se quiere insertar (esto también significa que no hacemos "swap" entre elementos iguales).

c) *Selection Sort NO es un algoritmo estable* ya que funciona encontrando al elemento mínimo de la secuencia y luego insertándolo en su posición correcta al intercambiarlo con el elemento que está en la posición de este elemento mínimo, y es justo por este intercambio que es inestable.
Por ejemplo, digamos que tenemos dos elementos a, b en la secuencia S tales que $a < b$.

$$S : \left[\text{␣}, \text{␣}, \text{␣}, \ldots, a, \text{␣}, \ldots, \text{␣}, b, \ldots, \text{␣}, \text{␣}, \text{␣} \right]$$

En una iteración del algoritmo, hay un elemento c en la secuencia S que es menor al elemento a pero se encuentra después del elemento b en la secuencia.

$$S : \left[\text{␣}, \text{␣}, \text{␣}, \ldots, a, \text{␣}, \ldots, \text{␣}, b, \ldots, c, \text{␣}, \text{␣} \right]$$

En este punto, se hace "swap" entre los elementos a y c, por lo que ahora el elemento a está después del elemento b en la secuencia; aunque sabemos que $a < b$.
Por lo tanto, esto ilustra que el algoritmo no preserva el orden relativo de los elementos.

d) *Shell Sort NO es un algoritmo estable* ya que se van comparando elementos distantes, los cuales se intercambian si corresponde. Esto es, a medida que se aumentan los pasos, el tamaño de los saltos disminuye, de tal manera que un valor se moverá bastantes posiciones cerca de la que será su posición final (con sólo unas pocas comparaciones e intercambios). Notemos que generalmente para el tamaño de los saltos se utiliza $\frac{n}{2}$.
Por ejemplo, digamos que tenemos dos elementos a, b en la secuencia S tales que $a < b$.

$$S : \left[\text{␣}, \text{␣}, \text{␣}, \ldots, a, \text{␣}, \ldots, \text{␣}, b, \ldots, \text{␣}, \text{␣}, \text{␣} \right]$$

Entonces para este punto, cuando llevamos acabo los intercambios pertinentes entre los elementos que están involucrados en este tamaño de salto, puede que haya un elemento c en la secuencia S que sea menor al elemento a pero que se encuentre despúes del elemento b en la secuencia.

$$S : \left[\text{␣}, \text{␣}, \text{␣}, \ldots, a, \text{␣}, \ldots, \text{␣}, b, \ldots, c, \text{␣}, \text{␣} \right]$$

Cuando se reduzca el tamaño del salto en la siguiente iteración, se hace "swap" entre los elementos a y c, por lo que ahora el elemento a está después del elemento b en la secuencia; aunque sabemos que $a < b$.
Por lo tanto, esto ilustra que el algoritmo no preserva el orden relativo de los elementos conforme se va disminuyendo el tamaño de los saltos.

e) *Local Insertion Sort es un algoritmo estable* ya que trabaja con una lista biligada que siempre debe estar en orden y a la cual se van añadiendo nuevos elementos considerando la posición donde se realizó la última inserción. Es decir, esta característica hace que se preserve el orden relativo entre los elementos porque se busca la posición correcta para insertar un nuevo elemento a con respecto al apuntador del último elemento insertado b, donde si $a < b$ se busca a la izquierda y en caso contrario, a la derecha.
Por lo tanto, nunca habrá un elemento que sea mayor al último elemento insertado de su lado izquierdo en la lista.

■

5. **Opcional.**
Problema Γ: Dados n puntos en el plano, encontrar un poligono que tenga como vértices los n puntos dados.
(a) Proporcione un algoritmo que solucione el problema Γ.
(b) Justifique que su propuesta de algoritmo es correcta.
(c) Calcule, con detalle, la complejidad computacional de su propuesta
(d) Proporcione un pseudo-código del algoritmo propuesto.

Sin respuesta.

■