Estructuras de Datos y Algoritmos – IIC2133

I2

11 mayo 2015

1. Una observación que hicimos en clase sobre los algoritmos de **ordenación por comparación de elementos adyacentes**, p.ej., *insertionSort*( ), es que su debilidad (en términos del número de operaciones que ejecutan) radica en que sólo comparan e intercambian posiciones de elementos adyacentes. Así, si tuviéramos un algoritmo que usara la misma estrategia general de *insertionSort*( ), pero que comparara elementos que están a una cierta distancia > 1 entre ellos, podríamos esperar un mejor desempeño.

a) Considera el siguiente arreglo **a** y calcula cuántas comparaciones entre elementos hace *insertionSort*( ) para ordenarlo de menor a mayor; muestra que entiendes cómo funciona *insertionSort*( ):

a = [11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1]

insertionSort( ) coloca el segundo elemento ordenado con respecto al primero, luego el tercero ordenado con respecto a los dos primeros (ya ordenados entre ellos), luego el cuarto ordenado con respecto a los tres primeros (ya ordenados entre ellos), etc. En el caso del arreglo **a**, *insertionSort*( ) básicamente va moviendo cada elemento, 10, 9, …, 1, hasta la primera posición del arreglo. Para ello, el 10 es comparado una vez (con el 11), el 9 es comparado dos veces (con el 11 y con el 10), el 8 es comparado tres veces (con el 11, el 10 y el 9), y así sucesivamente; finalmente, el 1 es comparado 10 veces. Luego el total de comparaciones es 1 + 2 + 3 + ... + 10 = **55**.

b) Considera ahora el siguiente algoritmo de ordenación, *shellSort*( ), y calcula cuántas comparaciones entre elementos hace para ordenar el mismo arreglo **a** de menor a mayor. Muestra que entiendes cómo funciona este algoritmo; en particular, ¿qué relación tiene con *insertionSort*( )?

```
void shellSort(a)
   gaps[] = {5,3,1}
   for (t = 0; t < 3; t = t+1)
      gap = gaps[t]
      for (j = gap; j < a.length; j = j+1)
         tmp = a[j]
         k = j
         for (; k >= gap && tmp < a[k-gap]; k = k-gap)
            a[k] = a[k-gap]
         a[k] = tmp
```

[Primero, notemos que un algoritmo de ordenación por comparación muy afortunado podría ordenar el arreglo **a** haciendo sólo 5 comparaciones (y los consiguientes intercambios): 11 con 1, 10 con 2, 9 con 3, 8 con 4, y 7 con 5; este sería el mejor caso.]

En el caso de *shellSort*( ), notemos que las comparaciones entre elementos del arreglo se dan sólo en la comparación **`tmp < a[k-gap]`**; el algoritmo realiza **11** de estas comparaciones con resultado **`true`** y otras **17** con resultado **`false`**; en total, **28**.

Primero, realiza *insertionSort* entre elementos que están a distancia 5 entre ellos (según las posiciones que ocupan en **a**, no en cuanto a sus valores): el 6 con respecto al 11, el 5 c/r al 10, el 4 c/r al 9, el 3 c/r al 8, el 2 c/r al 7, el 1 c/r al 11, y el 1 c/r al 6.

Luego, realiza *insertionSort* entre elementos que están a distancia 3 (nuevamente, según sus posiciones en el arreglo): el 2 c/r al 5 y el 7 c/r al 10.

Finalmente, realiza *insertionSort* entre elementos que están a distancia 1: el 3 c/r al 4 y el 8 c/r al 9; estos son los dos únicos pares de valores que aún están "desordenados" al finalizar el paso anterior.

2. **a**) Describe el algoritmo de ordenación *mergeSort*( ) con un nivel de detalle similar al visto en clase, y **calcula su complejidad** en notación O( ).

Esto está en los apuntes de clase. La descripción puede ser en prosa, pero tiene que referirse explícitamente a cada uno de los grandes temas del algoritmo: las dos llamadas recursivas, c/u sobre una mitad del arreglo; y la mezcla posterior y, muy importante, cómo se hace ésta (bastaría con explicar el primer `while`, que es lo básico de la mezcla).

```
mergeSort(a, tmp, e, w)
   if ( e < w )
      m = (e+w)/2
      mergeSort(a, tmp, e, m)
      mergeSort(a, tmp, m+1, w)
      merge(a, tmp, e, m+1, w)

merge(a, tmp, e, m, w)
   p = e, k = e, q = m
   while ( p <= m-1 && q <= w )
      if ( a[p].key < a[q].key )
         tmp[k] = a[p]; k = k+1; p = p+1
      else
         tmp[k] = a[q]; k = k+1; q = q+1
   while ( p <= m-1 )
      tmp[k] = a[p]; k = k+1; p = p+1
   while ( q <= w )
      tmp[k] = a[q]; k = k+1; q = q+1
   for ( k = e; k <= w; k=k+1 )
      a[k] = tmp[k]
```

El análisis también está en los apuntes de clase. Esta es una posibilidad, pero hay otras.

Sabemos que $T(1) = 0$ y $T(n) = 2T(n/2) + n$

Luego,

$T(n)/n = T(n/2)/(n/2) + 1$
$T(n/2)/(n/2) = T(n/4)/(n/4) + 1$
$T(n/4)/(n/4) = T(n/8)/(n/8) + 1$
... ...
$T(2)/2 = T(1)/1 + 1$

Si sumamos ambos lados del signo = y cancelamos los términos que aparecen a ambos lados, obtenemos

$T(n)/n = T(1)/1 + \log n$

b) Un paso fundamental de *mergeSort*( ) es mezclar dos subarreglos ya ordenados, de modo de formar un solo subarreglo ordenado con los elementos de ambos subarreglos. Esto requiere usar un arreglo auxiliar y finalmente copiar el contenido del arreglo auxiliar de vuelta al arreglo original (en las posiciones que correspondan). Sin embargo, si el contenido inicial de este arreglo original ya está ordenado, entonces no es necesario ejecutar la mezcla; **describe un cambio** (simple) que puedes hacer a tu algoritmo de (a) para tomar en cuenta esta situación.

De lo que se trata es de no hacer la mezcla cuando, después de las dos llamadas recursivas, tenemos que **`a[m] < a[m+1]`**; como en ese punto el subarreglo **`a[e]`**, ..., **`a[m]`** está ordenado y el subarreglo **`a[m+1]`**, ..., **`a[w]`** también está ordenado, entonces si **`a[m] < a[m+1]`** significa que todo el subarreglo **`a[e]`**, ..., **`a[w]`** está ordenado. Básicamente, hay que verificar que este *no sea el caso* justo antes de hacer la mezcla.

```
mergeSort(a, tmp, e, w)
   if ( e < w )
      m = (e+w)/2
      mergeSort(a, tmp, e, m)
      mergeSort(a, tmp, m+1, w)
      if ( a[m] >= a[m+1] )
         merge(a, tmp, e, m+1, w)
```

c) **Calcula la complejidad** en notación O( ) de tu algoritmo modificado de (b) *cuando se ejecuta sobre un arreglo que viene totalmente ordenado*.

Suponiendo que el arreglo tiene n elementos, la complejidad en este caso será **O(n)** (en lugar de O($n$log$n$)), que básicamente representa lo que cuesta reconocer que el arreglo está ordenado.

Lo que pasa, en esta versión levemente modificada de *mergeSort*, es que cuando las llamadas recursivas empizan a "volver", lo único que hacen con los elementos del arreglo es la comparación del nuevo **`if`**, y como estas siempre resultan falsas (ya que el arreglo está ordenado), entonces *cada instancia de una llamada recursiva* es **O(1)**.

¿Cuántas "instancias" hay? Las instancias más pequeñas (cuando la recursión se detiene) son cuando los subarreglos son de tamaño 1 (**`e == w`**); de éstas hay n, pero no involucran comparaciones entre elementos del arreglo. En el nivel inmediatamente anterior hay, aproximadamente, $n/2$; en el anterior, $n/4$; y así sucesivamente hasta el primer nivel, en que hay una llamada: $n/2 + n/4 + ... + 1 = n$.

3. **a**) Considera la siguiente afirmación respecto a un algoritmo de ordenación de strings del mismo largo:

> "Otra forma de establecer la demostración de que el algoritmo es correcto es pensar en el futuro: si los caracteres que aún no han sido examinados para un par de claves son idénticos, entonces cualquier diferencia entre las claves está restringida a los caracteres que ya han sido examinados, de modo que las claves han sido ordenadas apropiadamente y permanecerán así debido a la estabilidad. Si, por el contrario, los caracteres que no han sido examinados son diferentes, entonces los caracteres ya examinados no importan y una pasada posterior ordenará correctamente el par de claves basado en esas diferencias más significativas."

¿Exactamente a cuál algoritmo para ordenación de strings se refiere esta afirmación? **Justifica**.

Se refiere al algoritmo que en clase llamamos *radixSort*, que ordena los strings ordenándolos primero según el carácter menos significativo —el de más a la derecha—, luego según el siguiente carácter menos significativo —el que está inmediatamente a la izquierda del que está más a la derecha—, y así sucesivamente hasta finalmente ordenar los strings según el carácter más significativo —el de más a la izquierda. La ordenación de los strings según un carácter determinado debe hacerse usando un algoritmo de ordenación *estable*, p.ej., *countingSort*.

b) Otra forma de ordenar strings, especialmente cuando son de diferentes largos, es considerar los caracteres de izquierda a derecha y usar el siguiente método recursivo: Usamos *countingSort*( ) para ordenar los strings de acuerdo con el primer carácter; luego, recursivamente, ordenamos los strings que tienen un mismo primer carácter (excluyendo este primer carácter). Similarmente a *quickSort*( ), este algoritmo particiona el arreglo de strings en subarreglos que pueden ser ordenados independientemente para completar la tarea, sólo que particiona el arreglo en un subarreglo para cada posible valor del primer carácter, en lugar de las dos particiones de *quickSort*( ). **Escribe este algoritmo**.

```
sortString(a, e, w, k):
```
—ordena recursivamente el arreglo **a** *de strings, desde* **a[e]** *hasta* **a[w]**, *a partir del* **k**-*ésimo carácter*
—convierte cada carácter a un dígito entre 0 y 127 mediante la función (ficticia) **dgt**
—usa arreglos auxiliares **b** *y* **c**

```
   if e < w:
      for i = e, ..., w:
         p = dgt(a[i][k])
         b[p+2] = b[p+2]+1
      for r = 0, ..., 128:
         b[r+1] = b[r+1] + b[r]
      for i = e, ..., w:
         p = dgt(a[i][k])
         c[b[p+1]] = a[i]
         b[p+1] = b[p+1]+1
      for i = e, ..., w:
         a[i] = c[i-e]
      for r = 0, ..., 127:
         sortString(a, e+b[r], e+b[r+1]-1, k+1)
```

—los cuatro primeros **for** *implementan* **countingSort()**
—el quinto **for** *hace las llamadas recursivas sobre cada una de las particiones (strings con el mismo* **k**-*ésimo carácter)*

4. Considera el siguiente grafo no direccional, representado mediante sus listas de adyacencias:

[0]: 6 – 2 – 1 – 5	[1]: 0	[2]: 0	[3]: 5 – 4	[4]: 5 – 6 – 3
[5]: 3 – 4 – 0	[6]: 0 – 4	[7]: 8	[8]: 7	[9]: 11 – 10 – 12
[10]: 9	[11]: 9 – 12	[12]: 11 – 9		

Determina las **componentes conectadas** de este grafo, ejecutando el algoritmo basado en DFS estudiado en clase. En particular, muestra el orden en que se van produciendo cada una de las llamadas recursivas, si la primera llamada es DFS(0); marca los puntos de retorno de las llamadas en que se completa la detección de una componente conectada; y para cada componente conectada detectada lista sus vértices.

Respuesta: Próxima página

dfs(0)
    dfs(6)
        0 √ –*ya lo había descubierto*
        dfs(4)
            dfs(5)
                dfs(3)
                    5 √
                    4 √
                3 **fin** –*terminé de descubrir todo lo que podía desde* 3
                4 √
                0 √
            5 **fin**
            6 √
            3 √
        4 **fin**
    6 **fin**
    dfs(2)
        0 √
    2 **fin**
    dfs(1)
        0 √
    1 **fin**
    5 √
0 **fin –> *aquí se completa una componente conectada, formada por* 0, 1, 2, 3, 4, 5, 6**
dfs(7)
    dfs(8)
        7 √
    8 **fin**
7 **fin –> *aquí se completa otra componente conectada:* 7, 8**
dfs(9)
    dfs(11)
        9 √
        dfs(12)
            11 √
            9 √
        12 **fin**
    11 **fin**
    dfs(10)
        9 √
    10 **fin**
    12 √
9 **fin –> *aquí se completa la última componente conectada:* 9, 10, 11, 12**

5. Considera un grafo direccional acíclico. Dados dos vértices, v y w, de este grafo, describe un algoritmo para determinar el *primer ancestro común* de v y w. El **primer ancestro común** de v y w es un vértice que no tiene descendientes que también sean ancestros de v y w.

Una posibilidad: Sea G el DAG original. Primero, determinamos G^T, el grafo transpuesto de G: v y w tienen un ancestro común u en G si u es alcanzable (usando BFS) tanto desde v como desde w en G^T. Si u y u' son ancestros comunes de v y w en G, entonces el *primer* ancestro común será u si u' es alcanzable desde u en G^T (y viceversa). Más en general, si hay un conjunto de vértices que son ancestros comunes de v y w en G, entonces considera el subgrafo de G^T formado por esos vértices y por las aristas que los unen, y realiza una ordenación topológica de este subgrafo: la raíz resultante es el primer ancestro común.