Actividad de Aprendizaje
Semana 02

Indicaciones Generales:

1. El plagio se sanciona con la suspensión o expulsión del estudiante de la Universidad. Reglamento General de Estudios.
2. Sea cuidadoso con su redacción, la cual formará parte de su calificación (aspectos léxicos, sintácticos, semánticos).
3. Lea bien la pregunta o enunciado antes de proceder a su desarrollo, administre su tiempo eficazmente.
4. Se debe incluir en la entrega los archivos desarrolladores en C++ y su evidencia de ejecución a través de imágenes.

Actividad 1.

1. Uno de los algoritmos de ordenamiento más fáciles de entender es el que llamamos Max- sort, que funciona como sigue: hallamos la clave más grande, digamos `max`, en la porción no or- denada del arreglo (que en un principio es todo el arreglo) y luego intercambiamos `max` con el ele- mento que ocupa la última posición de la sección no ordenada. Ahora `max` se considera parte de la porción ordenada, que consiste en las claves más grandes al final del arreglo; ya no está en la sección no ordenada. Este procedimiento se repite hasta que todo el arreglo está ordenado.

    a. Escriba un algoritmo para Maxsort suponiendo que un arreglo *E* contiene *n* elementos a or-denar, con índices 0,…, *n* — 1.

    b. ¿Cuántas comparaciones de claves efectúa Maxsort en el peor caso? ¿En promedio?

2. A continuación vienen algunos ejercicios acerca de un método de ordenamiento llamado Ordenamiento de Burbuja (*Bubble Sort*), que opera efectuando varias pasadas por el arreglo, comparando pares de claves en posiciones adyacentes e intercambiando sus elementos si no están en orden. Es decir, se comparan las claves primera y segunda y se intercambian si la primera es mayor que la segunda; luego se comparan la (nueva) segunda clave y la tercera y se intercambian si es necesario, y así. Es fácil ver que la clave más grande se irá desplazando hacia el final del arreglo; en pasadas subsiguientes se hará caso omiso de ella. Si en una pasada no se intercambian ele mentos, el arreglo estará totalmente ordenado y el algoritmo puede parar. El algoritmo que sigue define con más precisión esta descripción informal del método.

Algoritmo 1 Ordenamiento de Burbuja

Entradas: *E*, un arreglo de elementos; y $n \geq 0$, el número de elementos.

Salidas: *E* con sus elementos en orden no decreciente según sus claves.

```
void bubbleSort(Elemento[] E, int n)

   int numPares; // el número de pares a comparar

   boolean huboCambio: // true si se efectuó un intercambio

   int j;


   numPares = n — 1;
   huboCambio = true;
   while (huboCambio)

      huboCambio = false;

      for (j = 0; j < numPares; j ++)
        if (E[j] > E[j + 1])

           Intercambiar E[j] y E[j+ 1].
           huboCambio = true;

        // Continuar ciclo for.

      numPares --;
   return;
```

El ejemplo de la figura 1 ilustra cómo funciona el Ordenamiento de Burbuja.

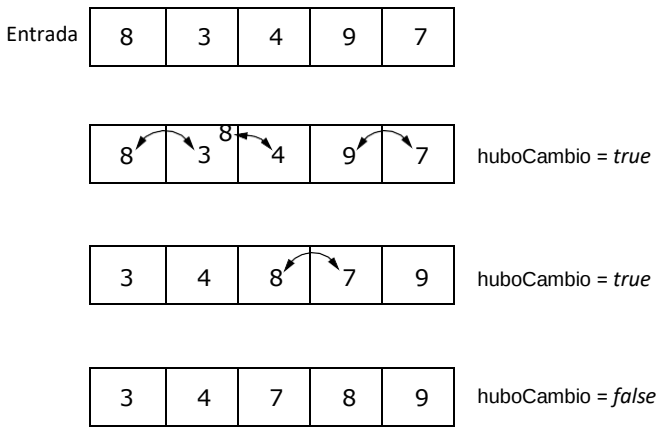

Figura 1 Ordenamiento de Burbuja

a. ¿Cuántas comparaciones de claves efectúa Ordenamiento de Burbuja en el peor caso? ¿Qué acomodo de claves constituye un peor caso?

b. ¿Qué acomodo de claves es un caso óptimo para Ordenamiento de Burbuja?, es decir, ¿con qué entradas efectúa el menor número de comparaciones?, ¿cuántas comparaciones efectúa en el mejor caso?

3. La corrección de Ordenamiento de Burbuja (ejercicio 2) depende de varias cosas. Éstas son fáciles de verificar, y vale la pena hacerlo para reconocer de forma consciente las propiedades matemáticas que intervienen.

   a. Demuestre que, después de una pasada por el arreglo, el elemento más grande estará al final.

   b. Demuestre que, si ningún par de elementos consecutivos está en desorden, todo el arreglo está ordenado.

4. Podemos modificar Ordenamiento de Burbuja (ejercicio 2) para que no efectúe comparaciones innecesarias al final del arreglo tomando nota de dónde se efectuó el último intercambio dentro del ciclo `for`.

   a. Demuestre que si el último intercambio efectuado en alguna pasada se efectúa en las posiciones j-ésima y (j + 1)-ésima, entonces todos los elementos, del (j + 1)-ésimo hasta el (n — 1)-ésimo, están en su posición correcta. (Observe que esto es más categórico que decir simple- mente que estos elementos están en orden.)

   b. Modifique el algoritmo de modo que, si el último intercambio efectuado durante una pasada se efectúa en las posiciones j-ésima y (j + 1)-ésima, la siguiente pasada no examinará los elementos desde la (j + 1)-ésima posición hasta el final del arreglo.

   c. ¿Este cambio afecta el comportamiento de peor caso del algoritmo? Si lo hace, ¿cómo loafecta?

   ¿Se puede hacer algo similar a la mejora del ejercicio anterior para evitar comparaciones in- necesarias cuando las claves del principio del arreglo ya están en orden? En tal caso, escriba las modificaciones del algoritmo. Si no se puede, explique por qué.

5. Observamos que un peor caso de Ordenamiento por inserción es cuando las claves están en un principio en orden descendente. Describa al menos otros dos acomodos iniciales de las claves que también sean peores casos. Presente entradas para las que el número *exacto* de comparaciones de claves (no sólo el *orden asintótico*) sea el peor posible.

6. Sugiera un caso óptimo para Ordenamiento por inserción. Describa cómo estarían acomodados los elementos de la lista, e indique cuántas comparaciones de elementos de la lista se efectuarían en ese caso.

7. En este ejercicio se examina un algoritmo alterno para Partir, con código sencillo y ele- gante. Este método se debe a Lomuto; lo llamamos `partirL`. La idea, que se ilustra en la figura 2, consiste en reunir elementos pequeños a la izquierda de la vacante, elementos grandes in- mediatamente a la derecha de la vacante y elementos desconocidos (es decir, no examinados) a la extrema derecha del intervalo. En un principio, todos los elementos están en el grupo desconocido. Los elementos “pequeños” y “grandes” se determinan respecto a `pivote`. Cuando `partirL` encuentra un elemento pequeño en el grupo desconocido, lo coloca en la vacante y luego crea una nueva vacante una posición a la derecha transfiriendo un elemento grande desde ese lugar hastael extremo del intervalo “grande”.

```
int partirL(Elemento[] E, Clave pivote, int primero, int ultimo)
   int vacante, desconocido;


1. vacante = primero;
2. for(desconocido = primero + 1; desconocido <= ultimo; desconocido ++)
3.     if(E[desconocido] < pivote)
4.       E[vacante] = E[desconocido];
5.       E[desconocido] = E[vacante+1];
6.       vacante ++;
7. return vacante;
```

En cada iteración de su ciclo, `partirL` compara el siguiente elemento desconocido, que es `E[desconocido]`, con `pivote`. Por último, una vez que todos los elementos se han comparado con `pivote`, se devuelve `vacant` como `puntoPartir`.

a. Al principio de cada una de las líneas de la 2 a la 6, ¿cuáles son las fronteras de la región de claves pequeñas y de la región de claves grandes? Exprese su respuesta utilizando `descono cido` y otras variables de índice.

b. Al principio de la línea 7, ¿cuáles son las fronteras de la región de claves pequeñas y de la región de claves grandes? Exprese su respuesta *sin* usar `desconocido`.

c. ¿Cuántas comparaciones de claves efectúa `partirL` con un subintervalo de *E* que tiene *k* elementos? Si Quicksort usa `partirL` en lugar de `partir`, ¿qué impacto tiene ello sobre el número total de comparaciones de claves efectuadas en el peor caso?

8. Suponga que el arreglo *E* contiene las claves 10, 9, 8, . . . , 2, 1, y se debe ordenar con Quicksort.

a. Muestre cómo estarían acomodadas las claves después de las dos primeras invocaciones del procedimiento `partir` del algoritmo 3. Indique cuántos traslados de elementos efectúa ca-da una de estas dos invocaciones de `partir`. A partir de este ejemplo, estime el número total de traslados de elementos que se efectuarían para ordenar *n* elementos que al principio están en orden decreciente.

b. Haga lo mismo con `partirL`, que se describió en el ejercicio anterior.

c. Cite algunas de las ventajas y desventajas relativas de los dos algoritmos para partir.

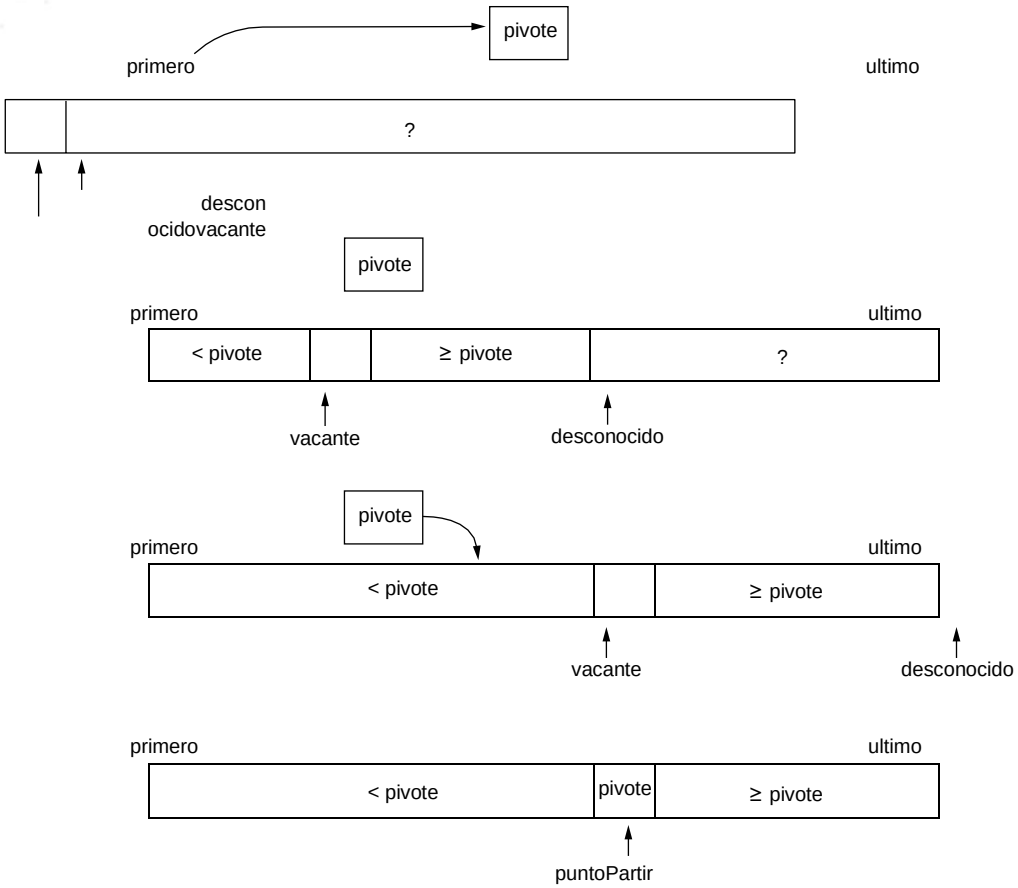

Figura 2 Cómo funciona PartirL: vistas inicial, intermedia y final

9. ¿Cuántas comparaciones de claves efectúa Mergesort si las claves ya están en orden cuando se inicia el ordenamiento?

10. Describimos Mergesort suponiendo que Fusionar desarrollaba su salida en un arreglo de trabajo y al terminar copiaba el contenido de ese arreglo en el arreglo de entrada.

    a. Deduzca una estrategia para cambiar del arreglo de entrada al de trabajo y viceversa de modo que se evite ese copiado extra. Es decir, en niveles alternos de la recursión, el arreglo deentrada original tiene los datos a fusionar o bien el arreglo de trabajo los tiene.

    b. Con la optimación anterior, ¿cuántos traslados de elementos efectúa Mergesort en promedio?Compare esa cifra con la de Quicksort (vea el ejercicio 2).