Práctico 10

Complejidad y sorting

NOTA: “costo exacto” equivale a “cantidad de operaciones básicas (en forma exacta)

Ejercicio 1 (I)

a) Escribir un algoritmo recursivo de ordenación, que dada una secuencia de largo *n*, divida la secuencia en 2 partes de tamaño igual a la mitad de la secuencia original y ordene cada subsecuencia. Una vez ordenadas estas subsecuencias, los elementos se intercalan de forma tal que los elementos queden ordenados. Este algoritmo se denomina *MergeSort*.
b) Indicar cual o cuales entradas constituyen el peor caso. Calcular el costo exacto del algoritmo en el peor caso, suponiendo *n* potencia de 2.
c) Determinar la técnica de diseño de algoritmos dentro de la cual se puede considerar la solución dada.
d) Dibujar el árbol de decisión para $n = 3$.

Ejercicio 2 (I)

a) Escribir un algoritmo recursivo de ordenación, que dada una una secuencia de largo *n*, divida el conjunto en 2 partes colocando todos los elementos menores que uno elegido en el primer conjunto y los mayores en el segundo. Este algoritmo se denomina *QuickSort*.
b) Determinar la técnica de diseño de algoritmos dentro de la cual se puede considerar la solución dada.
c) Indicar cual o cuales entradas constituyen el peor caso. Calcular el costo exacto del algoritmo en el peor caso

Ejercicio 3 (I)

a) Escribir un algoritmo recursivo tal que busque un elemento en una secuencia ordenada de largo *n*. Este algoritmo debe comparar el valor dado con el elemento en la posición *n*/3. En caso de que el valor buscado no coincida, se debe buscar el mismo en uno de los siguientes subarreglos: [inicio, n/3], [n/3 + 1, fin].
En caso que el elemento a buscar no pertenezca a la secuencia, el algoritmo devuelve 0, 1 en caso contrario.
b) Calcular el costo exacto del algoritmo en el peor caso, suponiendo *n* potencia de 3/2.
c) Calcular el costo exacto en el caso promedio asumiendo que el elemento está en la secuencia y que todas las posiciones son equiprobables en referencia a contener el elemento buscado.

EJERCICIO 4 (I)

Es posible utilizar la estructura de datos *Heap* para ordenar un vector de n elementos.

a) Diseñar un algoritmo de ordenación basado en esta estructura de datos.
b) Estudiar el costo exacto de ejecución para el peor caso del armado de un *Heap* a partir de un arreglo inicialmente desordenado.
c) Dibujar el árbol de decisión para n = 3.

EJERCICIO 5 (R)

a) Escribir un algoritmo recursivo tal que busque un elemento en una secuencia ordenada de largo n. Este algoritmo debe comparar el valor dado con el elemento en la posición $n/3$ y el de la posición $2n/3$. En caso de que el valor buscado no coincida, se debe buscar el mismo en uno de los siguientes subarreglos: [inicio, n/3], [n/3 + 1, 2n/3], [2n/3 + 1, fin]. El tamaño del conjunto de búsqueda se reduce a 1/3 del tamaño del conjunto original.
   En caso que el elemento a buscar no pertenezca a la secuencia, el algoritmo devuelve 0, 1 en caso contrario.
b) Estudiar el costo exacto del algoritmo en el peor caso, suponiendo n potencia de 3.

EJERCICIO 6 (R)

Un método de ordenación es estable si al final del mismo, elementos de igual valor permanecen en el mismo orden en que se encontraban originalmente en el conjunto. Determinar cuáles de los siguientes algoritmos son estables:

a) InsertionSort
b) SelectionSort
c) MergeSort
d) QuickSort
e) HeapSort

Nota: explicar cuando considere necesario para cual implementación es estable y para cual no.

EJERCICIO 7 (C)

Se quiere construir un programa que dada una lista L de n números distintos y un entero m, lea m números distintos de alguna entrada y a cada lectura, indique si el número leído pertenece ó no a L. Se tienen dos opciones para poder indicar si el número leído pertenece ó no a L:

- hacer una búsqueda lineal para cada lectura.
- ordenar L, haciendo búsqueda binaria para cada lectura.

a) Suponiendo que $m = \lfloor \log_2(\log_2 n) \rfloor$, ¿cuál de las dos opciones es más eficiente?
b) Suponiendo que $m = \lfloor n^{1/2} \rfloor$, ¿cuál de las dos opciones es más eficiente?

En ambos casos se debe justificar la opción elegida.

EJERCICIO 8 (R)

Dada una secuencia $a_1, a_2, \ldots, a_n$ de enteros distintos, se desea determinar el k-ésimo elemento, es decir, aquel elemento a_i en la secuencia tal que existen exactamente k-1 elementos menores que él en la secuencia.

Una forma de resolver el problema es ordenar la secuencia en forma ascendente, y tomar el elemento que queda en la posición k.

Otra forma posible de resolver el problema, es definir un algoritmo que determine el k-ésimo elemento sin ordenar totalmente la secuencia.

a) Definir un algoritmo basado en el *SelectionSort* para determinar el *k*-ésimo elemento.
b) Estudiar el costo exacto en el peor caso del algoritmo de la parte a).
c) Definir un algoritmo basado en el *HeapSort* para determinar el *k*-ésimo elemento.
d) Definir un algoritmo basado en el *QuickSort* para determinar el *k*-ésimo elemento.

EJERCICIO 9 (C)

Sea una lista de *n* elementos con 3 claves distintas: *cierto*, *falso* y *quizás*. Definir un algoritmo con costo en $\Theta(n)$ para reordenar la lista de modo que todos los elementos *falso* precedan a los elementos *quizás*, y que éstos precedan a los elementos *cierto*.

Nota: sólo se puede usar un tiempo constante adicional al recorrido inicial de la lista a ordenar. Calcule el costo exacto de su algoritmo de manera de mostrar que dicha función de costo efectivamente está en $\Theta(n)$

EJERCICIO 10 (C)

Una implementación particular del algoritmo *InsertionSort* es la dada en *IS1*:

```
void IS1 (element * a, int largo){
  int i, j;

  for (i = 0; i < largo - 2; i++)  {
    j = i + 1;

    while ((j > 0) && (a[j-1] > a[j])) {
      swap (a[j-1], a[j]);
      j--;
    }
  }
}
```

Definir una implementación *IS2* del *InsertionSort* que tenga menor cantidad de operaciones en el peor caso que *IS1*. Justifique su respuesta.

EJERCICIO 11 (I)

Muestre como queda el árbol de decisión para el siguiente algoritmo de ordenación con T = [A, B, C], n=3.

```
void ordenar(int *T, int n) {
     for (int i=1; i < n ; i++) {
           int x = T[i];
           int j = i - 1;
           while (j >= 0 && x < T[j]) {
                 T[j+1] = T[j];
                 j--;
           }
           T[j+1] = x;
     }
}
```

EJERCICIO 12 (R)

Suponga que tiene un árbol de decisión (AD) de un algoritmo de ordenación genérico. Siendo ***m*** la cantidad de nodos externos se sabe que la cantidad de nodos externos de AD es $\mathbf{m} \leq \mathbf{2^k}$, siendo ***k*** la profundidad del árbol.

i. Indique cual debe ser el menor valor posible para **m**. Justifique su respuesta.
ii. Se quiere encontrar la función $F_W(n)$ que acote inferiormente al costo de ejecución en el peor caso de los algoritmos de ordenación.

Demuestre que FW(n) = **n log n**

Explique detalladamente cada paso de la demostración.

EJERCICIO 13 (I)

Considere el tipo *Elemento* como predefinido con las operaciones usuales de comparación.
Dada una secuencia *A* de *N* elementos a ordenar, considere el algoritmo de ordenación de la Burbuja (BubbleSort). La estrategia se basa en comparar elementos consecutivos de la secuencia y desplazar el mayor hacia la derecha. Por ejemplo, usando notación de arreglos para la secuencia, se efectúa en primer lugar la comparación de A[1] con A[2], y suponiendo que A[1] < A[2] continúa con el siguiente paso: compara entonces A[2] con A[3], en caso contrario intercambia A[1] por A[2] para luego seguir con la comparación de A[2] con A[3](con los elementos ya cambiados) y así sucesivamente.
También puede utilizarse la misma estrategia pero de atrás hacia delante, de acuerdo al siguiente seudocódigo:

```
//A: entrada de elementos a ordenar
//N: tamaño de A

BubbleSort(arreglo A, int N)
      n_elem = N;
      mientras n_elem > 1 :
            para i de 1 a n_elem-1:
               si A[i] > A[i+1]:
                   intercambiar A[i] y A[i+1];
            n_elem--;
```

Considerando la entrada ***A = [a, b, c],*** dar el árbol de decisión del algoritmo indicando en cada paso el estado de la entrada. Explique el significado de los distintos nodos del árbol. En particular para este árbol indique la validez de lo representado por cada una de sus hojas.