PRÁCTICA 6

ANÁLISIS DE EFICIENCIA

1. ANÁLISIS DE ALGORITMOS

1) Dada una lista ordenada con representación de listas enlazadas, escriba un algoritmo para insertar un elemento (post condición: la lista sigue ordenada luego de insertar) y obtenga la complejidad computacional de dicho algoritmo. Calcule según mejor caso y peor caso.

2) Agrupe las siguientes funciones: dos funciones f y g pertenecen al mismo grupo si y solo si representan el mismo orden de complejidad.

$$
\begin{array}{ll}
f_1(n) = n & f_6(n) = 2^n \\
f_2(n) = n^3 & f_7(n) = logn \\
f_3(n) = nlogn & f_8 = n - n^3 + 9n^5 \\
f_4(n) = n^2 + logn & f_9(n) = n^2 \\
f_5(n) = 3n! & f_{10}(n) = (logn)^2
\end{array}
$$

3) Ordene las funciones del ejercicio 2) según su orden de crecimiento conforme crece n.

4) Se desea encontrar el elemento menor de una secuencia finita de enteros $x_1, x_2, ..., x_n$ usando el siguiente algoritmo:

```
funcion BuscarMinimo(X, n)
{
    min = X[0];
    for (j = 1; j < n; j++)
       if (X[j] < min)
          min = X[j];
    return min;
}
```

Realice el análisis temporal y espacial del algoritmo. Considere mejor caso y peor caso.

5) Mediante la notación asintótica, obtenga los tiempos de ejecución del peor caso supuesto para cada uno de los procedimientos siguientes (función de n):

a)

```
procedure prod_mat ( n: integer );
    var
        i, j, k: integer;
    begin
        for i := 1 to n do
            for j := 1 to n do begin
                C[i, j] := 0;
                for k := 1 to n do
                    C[i, j] := C[i, j] + A[i, k] * B[k, j]
            end
    end
```

b)

```
procedure misterio ( n: integer );
    var
        i, j, k: integer;
    begin
        for i := 1 to n - 1 do
            for j := i + 1 to n do
                for k := 1 to j do
                { alguna proposición que requiera tiempo O(1) }
    end
```

c)

```
procedure muy_impar ( n: integer );
        var
            i, j, x, y: integer;
        begin
            for i := 1 to n do
                if odd(i) then begin
                    for j := i to n do
                        x := x + 1;
                    for j := 1 to i do
                        y := y + 1
                end
        end
```

6) Explique con sus propias palabras, cómo calcular la complejidad de un algoritmo recursivo.

7) ¿Por qué no realizar el cálculo de la complejidad de los algoritmos recursivos de la misma forma que se hace con los iterativos?

8) El siguiente algoritmo cuenta el número de nodos en un árbol binario con todos los niveles llenos. Realice su análisis temporal.

```
funcion Cuenta(raiz)
{
  si (tipo(raiz) == hoja)
       retornar 1
     sino
        retornar 1 + Cuenta(raiz.izq) + Cuenta(raiz.der)
}
```

9) Diseñe un algoritmo recursivo para calcular la altura de un árbol binario. ¿Cuál es la complejidad temporal de su algoritmo?

10) Búsqueda binaria: dados un entero x y un arreglo A de *n* enteros que se encuentran ordenados en memoria, encontrar un i tal que A[i] == x o retornar 0 si x no se encuentra en el arreglo. A continuación se muestra el algoritmo que resuelve dicha búsqueda:

```
funcion busqueda_binaria(A, x, i, j)
{
   int medio

   si (i > j)          /* se cruzan indices: no existe x en A */
      retornar 0

   medio = (i+j) / 2
   si (A[medio] < x)   /* busco en la mitad superior */
      retornar busqueda_binaria(A, x, medio+1, j)
   else
      si (A[medio] > x) /* busco en la mitad inferior */
         retornar busqueda_binaria(A, x, I, medio-1)
      else
         return medio
}
```

El llamado a esta función se realiza asignando el tercer parámetro como 1 y el último como n. Realizar el análisis de complejidad computacional del algoritmo.

11) Algoritmo recursivo para ordenación Mergesort (arreglo de *n* elementos): Si $n = 1$, la respuesta es dicho elemento, sino se divide el arreglo en dos mitades. Cada mitad es ordenada recursivamente, esto dá dos mitades ordenadas que pueden ser fusionadas por otra función. La operación de fusión (merge) fusiona dos arreglos ordenados y retorna un tercer arreglo ordenado. Como los arreglos están ordenados, esto se puede hacer en una pasada a través de los arreglos.

```
int merge_sort(int arr[],int low,int high)
{
    int mid;
    if(low<high)
    {
        mid=(low+high)/2;
        // llamdo recursivo con las 2 mitades
        merge_sort(arr,low,mid);
        merge_sort(arr,mid+1,high);
        // Fusion
        merge(arr,low,mid,high);
    }

    return 0;
}
```

A continuación se describe la función que realiza el merge de dos arrays:

```
int merge(int arr[],int l,int m,int h)
{
    int arr1[N],arr2[N];  // arreglos auxiliares

    int n1,n2,i,j,k;
    n1 = m - l + 1;
    n2 = h - m;

    for(i = 0;i < n1;i++)
       arr1[i] = arr[l+i];
    for(j = 0;j < n2;j++)
       arr2[j] = arr[m+j+1];

    arr1[i]=9999;  // marca el final del arreglo
    arr2[j]=9999; // marca el final del arreglo

    i = 0; j = 0;
    for(k = l; k <= h; k++)  //fusion de los dos arreglos
    {
        if(arr1[i] <= arr2[j])
           arr[k] = arr1[i++];
        else
           arr[k] = arr2[j++];
    }

    return 0;
}
```

Implemente y realice el cálculo de complejidad del algoritmo merge_sort.

12) Realice el cálculo de complejidad de los algoritmos iterativos para ordenación de arrays: Inserción, Selección, ShellSort, QuickSort. Compare la complejidad de los algoritmos de ordenación: Burbuja, Inserción, Selección, ShellSort, MergeSort, QuickSort y mergeSort.

13) Compare la complejidad computacional de los algoritmos de búsqueda: secuencial y binaria.

14) Se desea eliminar todos los números duplicados de una lista o vector (arreglo). Por ejemplo, si el arreglo tiene los valores:

4 7 11 4 9 5 11 7 3 5

ha de obtenerse

4 7 11 9 5 3

Escribir una función que elimine los elementos duplicados de un arreglo. Calcule su complejidad computacional.

15) Escribir una función que elimine los elementos duplicados de un vector ordenado. ¿Cuál es la complejidad computacional de esta función? Compare con la que tiene la función del ejercicio anterior.

16) Supongamos que se tiene una secuencia de n números que deben ser ordenados: 1) Si se utiliza el método ShellSort, ¿cuántas comparaciones y cuántos intercambios se requieren para ordenar la secuencia si: a) ya está ordenado? b) está en orden inverso? 2) Realizar los mismos cálculos si se utiliza el algoritmo quickSort.

17) Se desa realizar un programa que realice las siguientes tareas:
a) Generar, aleatoriamente, una lista de 999 números reales en el rango de 0 a 2000.
b) Ordenar en modo creciente por el método de la burbuja.
c) Ordenar en modo creciente por el método Shell.
d) Buscar si existe el número x (leído del teclado) en la lista. Aplicar la búsqueda binaria.
Ampliar el programa anterior de modo que pueda obtener y visualizar en el programa principal los siguientes tiempos:
t1: Tiempo empleado en ordenar la lista con cada uno de los métodos.
t2: Tiempo que se emplearía en ordenar la lista ya ordenada.
t3: Tiempo empleado en ordenar la lista en forma inversa.
Para tomar los tiempos de ejecución en CPU puede utilizar la rutina `gettimeofday()` de la librería `sys/time.h`

18) Los árboles binarios de búsqueda son llamado así por su eficiencia en las operaciones de búsqueda. Realice el análisis de complejidad computacional de las funciones de ABB: insertar un dato a un ABB, eliminar un dato de un ABB, buscar un elemento en un ABB, analice la complejidad computacional de los 3 recorridos vistos en clase.

19) Investigue las propiedades de los árboles AVL. Explique sus principales características y cómo las mismas repercuten en la eficiencia de las operaciones de búsqueda, inserción y eliminación. Explique cómo se realizan las operaciones de inserción y eliminación en un árbol AVL.