Unidad VI.

ORDENAMIENTO, BUSQUEDA E INTERCALACION

COMPETENCIA A DESARROLLAR:

El estudiante:

- Analiza el rendimiento del código generado en los procesos de ordenamientos y búsqueda aplicando concepto de complejidad computacional, algoritmos; seleccionando entre los algoritmos conocidos y aplicando Quicksort según el caso de estudio.

5.1 Complejidad computacional y algoritmos.

Habitualmente, un mismo problema puede tener numerosas soluciones que tienen distinta eficiencia (ej. rapidez de ejecución). Lo que se quiere en esta unidad es sentar las bases que permitan determinar qué algoritmo es mejor más eficiente dentro de una familia de algoritmos que resuelven el mismo problema.

La eficiencia de un algoritmo se relaciona con la cantidad de recursos que requiere para obtener una solución al problema (menor cantidad de recursos, mayor eficiencia).
Se asume que todo el algoritmo tratado va a ser eficaz, es decir, resuelven adecuadamente el problema para el que se han diseñado.

Definición 1:

Se define el costo o complejidad temporal de un algoritmo como el tiempo empleado por éste para ejecutarse y proporcionar un resultado a partir de los datos de entrada.

Definición 2:

Se define el costo o complejidad espacial de un algoritmo como cantidad de memoria requerida (suma total del espacio que ocupan las variables del algoritmo) antes, durante y después de su ejecución.

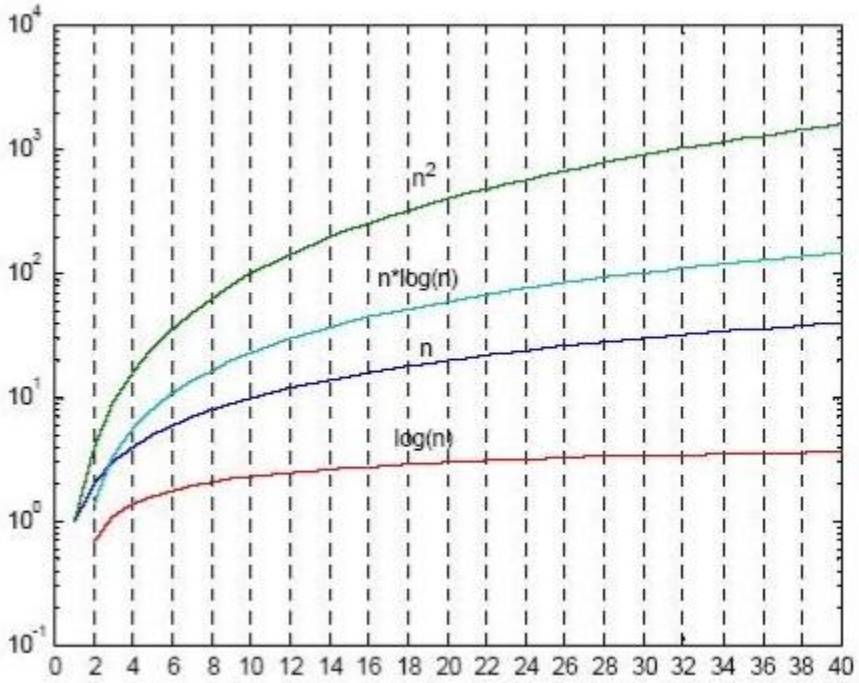
Lo que se pretende al analizar un algoritmo no es medir el costo temporal exacto (segundos) y la cantidad de memoria (bytes) que necesita para su ejecución, puesto que esto depende del proceso de programación y de ejecución de un programa y, por tanto, de la maquina o del compilador empleado. Lo que se requiere es obtener un valor estimado de estos valores mediante el cálculo del número de operaciones que será preciso realizar y del número de variables que será necesario emplear.

Para estudiar la complejidad computacional de los algoritmos se toma por cierto **el principio de invariancia** que dice: dos implementaciones distintas del mismo problema no difieren en su eficiencia más que en una constante multiplicativa; esto es, si dos implementaciones del mismo algoritmo necesitan t1 (n) y t2 (n) unidades de tiempo, donde n es el tamaño del vector de entrada, entonces existe un:

$$c>0 \text{ tal que } t1(n) < c*t2(n)$$

Hay dependencias en el tamaño n del volumen de datos a procesar las más frecuentes son:

TIEMPOS	DEPENDENCIA
Tiempo logarítmico	$c * \log(n)$
Tiempo lineal	$c * n$
Tiempo cuadrático	$c * n^2$
Tiempo polinomial	$k(c * n^k)$
Tiempo exponencial	c^n



Para realizar el cálculo de la eficiencia de un algoritmo se basa en contar el número de operaciones elementales que realiza. Por operación elemental se entiende operación cuyo tiempo de ejecución es constante y depende únicamente de la implementación como, por ejemplo, sumas, restas, productos, divisiones, módulo, operaciones lógicas, operaciones de comparación, etcétera.

Para representar la eficiencia de un algoritmo suele emplearse la notación "del orden de" (O (...)).

5.1.1 Complejidad computacional de las estructuras básicas de programación

En esta parte estudiaremos cuál es la complejidad computacional de las estructuras básicas que permiten construir algoritmos.

5.1.1.1 Sentencias sencillas

Nos referimos a las sentencias de asignación, comparación, operaciones lógicas y matemáticas. Este tipo de sentencias tiene un tiempo de ejecución constante que no depende del tamaño del conjunto de datos de entrada, siendo su complejidad O (1).

Ejemplo:

a=b; O(1)

5.1.1.2 Secuencia de sentencias

La complejidad de una serie de elementos de un programa es del orden de la suma de las complejidades individuales; el caso de una secuencia de sentencias sencillas la complejidad será: O (1)+ O (1)+...+ O (1)= k* O (1).

Ejemplo: se tiene las siguientes sentencias

Int a; O(1)

a=0;

a=20*10; complejidad es igual a: 3*O(1)

5.1.1.3 Sentencias de Selección (Condicionales)

La evaluación de la condición suele ser de O (1), complejidad que se debe sumar con la mayor complejidad computacional posible de las distintas ramas de ejecución, bien en la rama IF,o bien en la rama ELSE. En decisiones múltiples (ELSE IF, SWITCH CASE), se tomará la rama cuya complejidad computacional es superior.

Ejemplo:

```
if(a>b)
{
for(i=1;i<=10;i++){
}
}
else
{
A=10;
}
```

En este caso se tomaría la complejidad del if

5.1.1.3 Bucles

Los bucles son la estructura de control de flujo que suele determinar la complejidad computacional del algoritmo, ya que en ellos se realiza un mayor número de operaciones.

En los bucles con contador podemos distinguir dos casos: que el tamaño del conjunto de datos n forme parte de los límites del bucle o que no. Si la condición de salida del bucle es independiente de n entonces la repetición sólo introduce una constante multiplicativa:

for (i= 0; i < K; i++) { O (1)} la complejidad será K*O(1).

Cuando el número de iteraciones a realizar depende del tamaño de datos a procesar la complejidad computacional del bucle incrementará con el tamaño de los datos de entrada:

for (i= 0; i < n; i++) { O (1)} la complejidad será n* O (1)= O (n).

Cuando tenemos un ciclo anidado se tiene el siguiente análisis

```
for (i= 0; i < n; i++) {
      for (j= 0; j < n; j++) {
        ……..
      }
    }
```

En este caso el bucle exterior se realiza n veces, mientras que el interior se realiza 1, 2,3, ..., n veces respectivamente. La complejidad será: $n * n * O(1) = O(n^2)$

Ejemplo: se realizará un análisis de complejidad a los algoritmos:

5.2 Búsqueda secuencial

Consiste en comparar cada elemento del conjunto de búsqueda con el valor deseado hasta que éste sea encontrado o hasta que se termine de leer el conjunto

```
bool búsqueda(int v[], int tam){
        bool bandera=false;
        c=0;
        cout<<"\ndato a buscar?";
        cin>>dato;
        cout<<endl;
        while (c<tam && bandera==false) {
            if(nombre==vector[c]) {
             bandera=true;
            posi=c;
             }
         c=c+1;
        }
  return bandera;
}
```

La complejidad de este algoritmo medida en número de iteraciones en el mejor caso será 1, y será con aquella situación en la cual el elemento a buscar está en la primera posición del vector. En el peor caso la complejidad será tam y sucederá cuando el elemento buscar esté en la última posición del vector.

El orden de complejidad es lineal (O (tam)). Cada iteración necesita una asignación, una suma, tres comparaciones y un AND lógico.

Número de elementos examinados (peor caso)		
Tamaño del array	**Búsqueda binaria**	**Búsqueda secuencial**
1	1	1
10	4	10
1000	11	1000
5000	14	5000
100000	18	100000
1000000	21	1000000

5.3 Ordenamiento intercambio directo

El algoritmo se basa en el recorrido sucesivo de la lista a ordenar, comparando el elemento inferior de la lista con los restantes y efectuando intercambio de posiciones cuando el orden resultante de la comparación no sea el correcto.

```
void ordenamiento(int v[], int n){
  int aux=0,
  int I;
int j;
  for(i=0;i<=n-2;i++){
    for(j=i+1;j<=n-1;j++){
      if(v[i]>v[j]){
        aux=v[i];
        v[i]=v[j];
        v[j]=aux;
      }
    }
  }
}
```

El algoritmo consta de dos bucles anidados, está dominado por los dos bucles. De ahí, que el análisis del algoritmo en relación a la complejidad sea inmediato, siendo n el número de elementos, el primer bucle hace n-1 pasadas y el segundo n-i-1 comparaciones en cada

pasada (i es el índice del bucle externo, i = 0 .. n-2). El número total de comparaciones se obtiene desarrollando la sucesión matemática formada para los distintos valores de i:

$$n-1,\ n-2,\ n-3,\ \ldots\ ,\ 1$$

El número de comparaciones se obtiene sumando los términos de la sucesión:

$$\frac{(n-1)*n}{2}$$

y un número similar de intercambios en el peor de los casos. Entonces, el número de comparaciones y de intercambios en el peor de los casos es:

$$((n-1)*n)/2 = (n^2 - n)/2$$

El término dominante es n^2 por tanto la complejidad del ordenamiento por el método intercambio directo es $0(n^2)$.

5.4 Complejidad de algunos métodos de ordenamiento más utilizados

Algoritmo	Operaciones máximas
Burbuja	$O(n^2)$
Inserción	$O\ (n^2/4)$
Selección	$O\ (n^2)$
Shell	$O\ (n \log^2 n)$
Merge	$O\ (n \log n)$
Quick	$O\ (n^2)$ en peor de los casos y $O(n \log n)$ en el promedio de los casos.