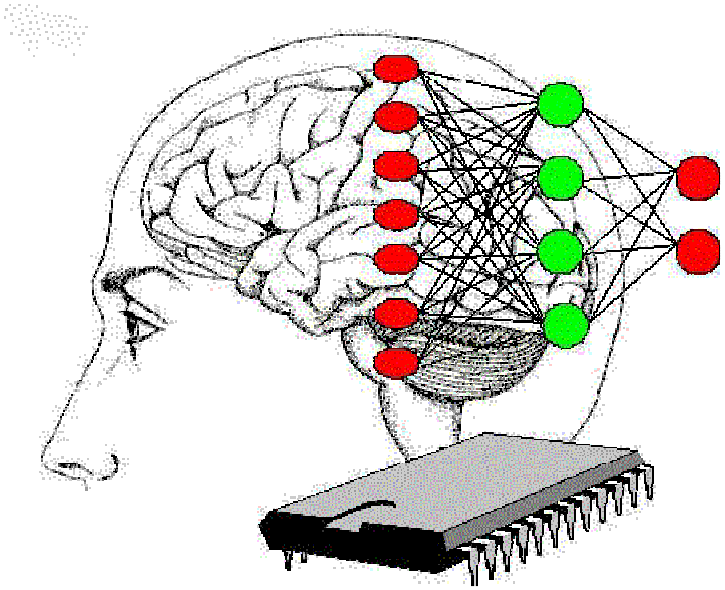


Algoritmos
y
Estructura de Datos I

TEORÍA DE LA COMPLEJIDAD ALGORÍTMICA

INTRODUCCIÓN

En la ciencia de la computación los algoritmos son más importantes que los LP[1] o que las computadoras; la solución de un problema haciendo uso de las computadoras requiere por una parte un algoritmo o método de resolución y por otra un programa o codificación del algoritmo en un LP. Ambos componentes tienen importancia; pero la del algoritmo es absolutamente indispensable; sabemos que un algoritmo es una secuencia de pasos para resolver un problema, sus características son:

1. Independiente: Del LP y de la máquina.
2. Definido: De pasos claros y concretos.
3. Finito: En el número de pasos que usará.
4. Preciso: Cada paso arroja un cálculo correcto.
5. Recibe datos: Debe poseer datos de entrada.
6. Arroja información: Debe arrojar información de salida.

Debemos saber que una solución es un conjunto único, pero no es el único conjunto de pasos que entregan la solución, existen muchas alternativas de solución y estas alternativas pueden diferir por:

- *Número de pasos*
- *Estructuras*

Ahora que sabemos que existen muchas alternativas de solución para un problema, debemos seleccionar el algoritmo más eficiente, el mejor conjunto de pasos, el que tarde menos en ejecutarse, que tenga menos líneas de código. Esta selección puede ser ejecutada a simple vista con sólo observar la cantidad de líneas del programa, pero cuando el programa crece se requiere una medición más exacta y apropiada, para esto se realizan ciertas operaciones matemáticas que establecen la eficiencia teórica del programa, al estudio de estos casos se denomina Complejidad Algorítmica.

[1] Lenguajes de Programación

COMPLEJIDAD ALGORÍTMICA

- Un algoritmo será mas eficiente comparado con otro, siempre que consuma menos recursos, como el tiempo y espacio de memoria necesarios para ejecutarlo.
- La eficiencia de un algoritmo puede ser cuantificada con las siguientes medidas de complejidad:

1. ***Complejidad Temporal o Tiempo de ejecución:*** Tiempo de cómputo necesario para ejecutar algún programa.
2. ***Complejidad Espacial:*** Memoria que utiliza un programa para su ejecución, La eficiencia en memoria de un algoritmo indica la cantidad de espacio requerido para ejecutar el algoritmo; es decir, el espacio en memoria que ocupan todas las variables propias al algoritmo. Para calcular la *memoria estática* de un algoritmo se suma la memoria que ocupan las variables declaradas en el algoritmo. Para el caso de la *memoria dinámica*, el cálculo no es tan simple ya que, este depende de cada ejecución del algoritmo.

- Este análisis se basa en las *Complejidades Temporales*, con este fin, para cada problema determinaremos una medida *N*, que llamaremos tamaño de la entrada o número de datos a procesar por el programa, intentaremos hallar respuestas en función de dicha *N*.
- El concepto exacto que cuantifica *N* dependerá de la naturaleza del problema, si hablamos de un array se puede ver a *N* como el rango del array, para una matriz, el número de elementos que la componen; para un grafo, podría ser el número de nodos o arcos que lo arman, no se puede establecer una regla para *N*, pues cada problema acarrea su propia lógica y complejidad.

TIEMPO DE EJECUCIÓN

- El tiempo de Ejecución de un programa se mide en función de *N*, lo que designaremos como *T(N).*
- Esta función se puede calcular físicamente ejecutando el programa acompañados de un reloj, o calcularse directamente sobre el código, contando las instrucciones a ser ejecutadas y multiplicando por el tiempo requerido por cada instrucción. Así, un trozo sencillo de código como:

```
S1;
for(x = 0; x < N; x++)
S2;
```

Demanda: $T(N) = t_1 + t_2 * N$

Donde t_1 es el tiempo que lleva ejecutar la serie S1 de sentencias, y t_2 es el que lleva la serie S2.

- Habitualmente todos los algoritmos contienen alguna sentencia condicional o selectiva, haciendo que las sentencias ejecutadas dependan de la condición lógica, esto hace que aparezca más de un valor para *T(N),* es por ello que debemos hablar de un rango de valores:

  $T_{min}(N) \leq T(N) \leq T_{max}(N)$

- Estos extremos son llamados "*el peor caso*" y "*el mejor caso*" y entre ambos se puede hallar "*el caso promedio*" o el más frecuente, siendo este el más difícil de estudiar; nos centraremos en el "*el peor caso*" por ser de fácil cálculo y se acerca a "*el caso promedio*", brindándonos una medida pesimista pero fiable.
- Toda función *T(N)* encierra referencias al parámetro *N,* y a una serie de constantes T_i dependientes de factores externos[2] al algoritmo. Se tratará de analizar los algoritmos dándoles autonomía frente a estos factores externos, buscando estimaciones generales ampliamente válidas, a pesar de ser demostraciones teóricas.

ASINTOTAS

- El análisis de la eficiencia algorítmica nos lleva a estudiar el comportamiento de los algoritmos frente a condiciones extremas. Matemáticamente hablando, cuando *N* tiende al infinito ∞, es un comportamiento asintótico.
- Sean *g(n)* diferentes funciones que determinan el uso de recursos, pudiendo existir infinidad de funciones *g*.
- Estas funciones *g* serán congregadas en familias, usando como criterio de agrupación su comportamiento asintótico, este comportamiento asintótico es similar cuando los argumentos toman valores muy grandes.
- Una familia de funciones que comparten un mismo comportamiento asintótico será llamada un *Orden de Complejidad*. Estas familias se designan con O( ).
- Para cada uno de estos conjuntos se suele identificar un miembro *f(n)* que se utiliza como representante de la familia, hablándose del conjunto de funciones *g* que son del orden de *f(n)*, denotándose como:

  $g \supset O(f(n))$, *g* esta incluido en *f(n)*

- Frecuentemente no es necesario conocer el comportamiento exacto, basta conocer una cota superior, es decir, alguna función que se comporte *'aún peor'*.
- El conjunto O(*f(n)*) es el de las funciones de orden de *f(n),* que se define como:

  $O(f(n)) = \{ g \mid \exists k \in \Re^+, \exists n_0 \in \aleph \text{ tales que}, \forall n \geq n_0, g(n) \leq kf(n) \}$

  O(*f(n)*) esta formado por aquellas funciones *g(n)* que crecen a un ritmo menor o igual que el de *f(n).*

[2] Memoria, velocidad de proceso, temperatura, etc.

De las funciones *g* que forman este conjunto O(*f(n)*) se dice que están dominadas asintóticamente por *f*, en el sentido de que para *N* suficientemente grande, y salvo una constante multiplicativa *k*, *f(n)* es una cota superior de *g(n)*.

Ejemplo: Se puede comprobar que la función $g(n) = 3n^3 + 2n^2$, es de $O(n^3)$

Aplicando la definición dada anteriormente:

$g(n) = 3n^3 + 2n^2$

$f(n) = n^3$

$n_0 = 0$

$k = 5$

Se tiene: $3n^3 + 2n^2 \leq 5n^3, \quad \forall\, n \geq 0$

n	***g( )***	***kf( )***
0	0	0
1	5	5
2	32	40
3	99	135

- El tiempo de ejecución es proporcional, se multiplica por una constante a alguno de los tiempos de ejecución anteriormente propuestos, además de la suma de algunos términos más pequeños. Así, un algoritmo cuyo tiempo de ejecución sea $T = 3n^2 + 6n$ se puede considerar proporcional a n^2. En este caso se puede asegurar que el algoritmo es del orden de n^2, y se escribe $O(n^2)$
- La notación O( ) ignora los factores constantes, desconoce si se hace una mejor o peor implementación del algoritmo, además de ser independiente de los datos de entrada del algoritmo. Es decir, la utilidad de aplicar esta notación a un algoritmo es encontrar el límite superior de su tiempo de ejecución '*el peor caso*'.

ÓRDENES DE COMPLEJIDAD

- La familia O(*f(n)*) define un *Orden de Complejidad*. Elegiremos como representante de este *Orden de Complejidad* a la función *f(n)* más sencilla perteneciente a esta familia.
- Las funciones de complejidad algorítmica más habituales en las cuales el único factor del que dependen es el tamaño de la muestra de entrada *n*, ordenadas de mayor a menor eficiencia son:

O(1)	Orden constante
O(*log n*)	Orden logarítmico
O(*n*)	Orden lineal
O(*n log n*)	Orden cuasi-lineal
O(n^2)	Orden cuadrático

$O(n^3)$	Orden cúbico
$O(n^a)$	Orden polinomio
$O(2^n)$	Orden exponencial
$O(n!)$	Orden factorial

- Se identifica una Jerarquía de Ordenes de Complejidad que coincide con el orden de la tabla mostrada; jerarquía en el sentido de que cada orden de complejidad inferior tiene a las superiores como subconjuntos.

  - O(1): Complejidad constante. Cuando las instrucciones se ejecutan una vez.

  - O(*log n*): Complejidad logarítmica. Esta suele aparecer en determinados algoritmos con iteración o recursión no estructural, ejemplo la búsqueda binaria.

  - O(*n*): Complejidad lineal. Es una complejidad buena y también muy usual. Aparece en la evaluación de bucles simples siempre que la complejidad de las instrucciones interiores sea constante.

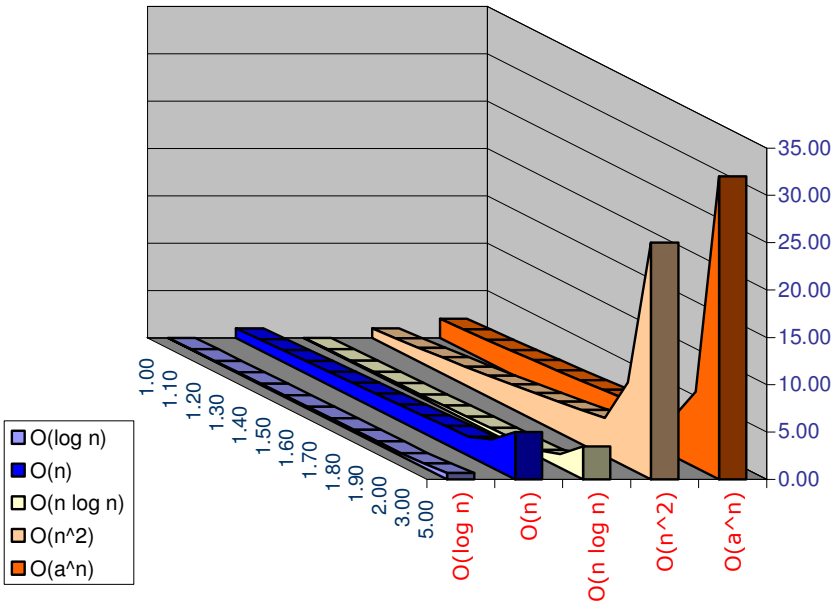
  - O(*n log n*): Complejidad cuasi-lineal. Se encuentra en algoritmos de tipo divide y vencerás como por ejemplo en el método de ordenación quicksort y se considera una buena complejidad. Si *n* se duplica, el tiempo de ejecución es ligeramente mayor del doble.

  - $O(n^2)$: Complejidad cuadrática. Aparece en bucles o ciclos doblemente anidados. Si n se duplica, el tiempo de ejecución aumenta cuatro veces.

  - $O(n^3)$: Complejidad cúbica. Suele darse en bucles con triple anidación. Si *n* se duplica, el tiempo de ejecución se multiplica por ocho. Para un valor grande de n empieza a crecer dramáticamente.

  - $O(n^a)$: Complejidad polinómica ($a > 3$). Si *a* crece, la complejidad del programa es bastante mala.

  - $O(2^n)$: Complejidad exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Se dan en subprogramas recursivos que contengan dos o más llamadas internas. ☠



Algoritmos Polinomiales: Aquellos que son proporcionales a n^a. Son en general factibles o aplicables, los problemas basados en estos algoritmos son solucionables.

Algoritmos Exponenciales: Aquellos que son proporcionales a k^n, $\forall\, k \geq 2$. En general no son factibles salvo un tamaño de entrada n exageradamente pequeño, pero generalmente pertenecen a un universo de problemas de los cuales el cómputo se hace imposible. ☠

REGLAS DE LA NOTACIÓN ASINTÓTICA

1. REGLA DE LA SUMA

Si $T_1(n)$ y $T_2(n)$ son las funciones que expresan los tiempos de ejecución de dos fragmentos de un programa, y se acotan de forma que se tiene:

$T_1(n) = O(f_1(n))$ y $T_2(n) = O(f_2(n))$

Se puede decir que:

$T_1(n) + T_2(n) = O(max(f_1(n), f_2(n)))$

2. REGLA DEL PRODUCTO

Si $T_1(n)$ y $T_2(n)$ son las funciones que expresan los tiempos de ejecución de dos fragmentos de un programa, y se acotan de forma que se tiene:

$T_1(n) = O(f_1(n))$ y $T_2(n) = O(f_2(n))$

Se puede decir que:

$T_1(n)\, T_2(n) = O(f_1(n)\, f_2(n))$

IMPORTANCIA DE LA EFICIENCIA

- *¿Que utilidad tiene diseñar algoritmos eficientes si las computadoras procesan la información cada vez más rápido?*

  Bien; para demostrar la importancia de la elaboración de algoritmos eficientes, se plantea el siguiente problema:

  - Contamos con una computadora capaz de procesar datos en 10^{-4} seg. En esta computadora se ejecuta un algoritmo que lee registros de una base de datos, dicho algoritmo tiene una complejidad exponencial 2^n, *¿Cuánto tiempo se tardará en procesar una entrada* ***n*** *de datos?*

n	**TIEMPO**
10	≈ 1 décima de segundo
20	≈ 2 minutos
30	> 1 día
40	> 3 años
50	= 3 570 años
100	= 4,019,693,684,133,150 milenios

  - Ahora se tiene la misma computadora capaz de procesar datos en 10^{-4} seg. Pero se ejecuta un algoritmo que hace el mismo trabajo antes citado, pero este algoritmo tiene una complejidad cúbica n^3, *¿Cuánto tiempo se tardará en procesar una entrada* ***n*** *de datos?*

n	**TIEMPO**
10	= 1 décima de segundo
20	= 8 décimas de segundo
100	= 1.7 minutos
200	= 13.3 minutos
1000	≈ 1 día

- Se puede concluir, que solo un algoritmo eficiente, con un orden de complejidad bajo, puede tratar grandes volumen de datos, se razona que un algoritmo es:
  - Muy eficiente si su complejidad es de orden *log n*
  - Eficiente si su complejidad es de orden n^a

- Ineficiente si su complejidad es de orden 2^n

- Se considera que un problema es tratable si existe un algoritmo que lo resuelve con complejidad menor que 2^n, y que es intratable o desprovisto de solución en caso contrario.

ESTIMACIÓN DE LA COMPLEJIDAD EN ALGORITMOS NO RECURSIVOS

• ASIGNACIONES Y EXPRESIONES SIMPLES (=)

El tiempo de ejecución de toda instrucción de asignación simple, de la evaluación de una expresión formada por términos simples o de toda constante es ***O(1)***.

• SECUENCIAS DE INSTRUCCIONES (;)

El tiempo de ejecución de una secuencia de instrucciones es igual a la suma de sus tiempos de ejecución individuales. Para una secuencia de dos instrucciones S_1 y S_2 el tiempo de ejecución esta dado por la suma de los tiempos de ejecución de S_1 y S_2:

$$T(S_1 ; S_2) = T(S_1) + T(S_2)$$

- ✓ Aplicando la regla de la suma:

$$O(T(S_1 ; S_2)) = max(O(T(S_1), T(S_2)))$$

• INSTRUCCIONES CONDICIONALES (IF, SWITCH-CASE)

El tiempo de ejecución para una instrucción condicional *IF-THEN* es el tiempo necesario para evaluar la condición, más el requerido para el conjunto de instrucciones que se ejecutan cuando se cumple la condición lógica.

$$T(IF\text{-}THEN) = T(condición) + T(rama\ THEN)$$

- ✓ Aplicando la regla de la suma:

$$O(T(IF\text{-}THEN)) = max(O(T(condición), T(rama\ THEN)))$$

El tiempo de ejecución para una instrucción condicional de tipo *IF-THEN-ELSE* resulta de evaluar la condición, más el máximo valor del conjunto de instrucciones de las ramas *THEN* y *ELSE.*

$$T(IF\text{-}THEN\text{-}ELSE) = T(condición) + max(T(rama\ THEN), T(rama\ ELSE))$$

- ✓ Aplicando la regla de la suma, su orden esta dada por la siguiente expresión:

$$O(T(IF\text{-}THEN\text{-}ELSE)) = O(\ T(condición)) + max(O(T(rama\ THEN)), O(T(rama\ ELSE))\ )$$

✓ Aplicando la regla de la suma:

$$O(T(IF\text{-}THEN\text{-}ELSE)) = max(O(\ T(condición)), max(O(T(rama\ THEN)), O(T(rama\ ELSE))\ ))$$

El tiempo de ejecución de un condicional múltiple (*SWITCH-CASE*) es el tiempo necesario para evaluar la condición, más el mayor de los tiempos de las secuencias a ejecutar en cada valor condicional.

• INSTRUCCIONES DE ITERACIÓN (FOR, WHILE-DO, DO-WHILE)

El tiempo de ejecución de un bucle *FOR* es el producto del número de iteraciones por la complejidad de las instrucciones del cuerpo del mismo bucle.

Para los ciclos del tipo *WHILE-DO* y *DO-WHILE* se sigue la regla anterior, pero se considera la evaluación del número de iteraciones para el peor caso posible. Si existen ciclos anidados, se realiza el análisis de adentro hacia fuera, considerando el tiempo de ejecución de un ciclo interior y la suma del resto de proposiciones como el tiempo de ejecución de una iteración del ciclo exterior.

• LLAMADAS A PROCEDIMIENTOS

El tiempo de ejecución esta dado por, el tiempo requerido para ejecutar el cuerpo del procedimiento llamado. Si un procedimiento hace llamadas a otros procedimientos "*no recursivos*", es posible calcular el tiempo de ejecución de cada procedimiento llamado, uno a la vez, partiendo de aquellos que no llaman a ninguno.

- *Ejemplo:* función no recursiva que halla el factorial de un número *n* cualquiera

```
int factorial(int n)                                    O(1)
{
      int fact = 1;                                     O(1)
      for(int i = n; i > 0; i--)                        O(n)
      fact = fact * i;                                  O(1)
      return fact;                                      O(1)
}
```

Su complejidad es lineal O(n), debido a que se tiene un bucle *FOR* cuyo número de iteraciones es *n*.

- *Ejemplo:* Si el bucle se repite un número fijo de veces, liberado de *n*, entonces el bucle introduce una constante que puede ser absorbida.

```
int y, z, k = 10;                                   O(1)
for(int i = 0; i < k; i++)                          k * O(1)
{
      y = y + i;                                    O(1)
      z = z + k;                                    O(1)
}
```

Su complejidad es constante; es decir, O(1), debido a que se tiene un bucle for independiente de *n*.

- *Ejemplo:* Dos bucles anidados dependientes de *n*.

```
for(int i = 0; i < n; i++)                          O(n)
{
      for(int z = 0; z < n; z++)                    O(n)
      {
            if(vector[z] > vector[z + 1])           O(1)
            {
            aux = vector[z];                        O(1)
            vector[z] = vector[z + 1];              O(1)
            vector[z + 1] = aux;                    O(1)
            }
      }
}
```

Tenemos O(n) * O(n) * O(1) = $O(n^2)$, complejidad cuadrática.

- *Ejemplo:* Dos bucles anidados, uno dependiente *n* y otro dependiente del bucle superior.

```
for(int i = 0; i < n; i++)                          O(n)
{
      for(int z = n; z < i; z--)                    O(n)
      {
            if(vector[z] < vector[z - 1])           O(1)
            {
            aux = vector[z];                        O(1)
            vector[z] = vector[z - 1];              O(1)
            vector[z - 1] = aux;                    O(1)
```

```
                        }
        }
}
```

Tenemos que el bucle exterior se ejecuta n veces ⇒ O(n), el bucle interno se ejecuta n + … + 3 + 2 + 1 veces respectivamente, o sea $(n * (n+1))/2$ ⇒ O(n).

$O(n) * O(n) * O(1) = O(n^2)$, complejidad cuadrática.

- *Ejemplo:* Bucles donde la evolución de la variable de control es ascendente no lineal.

```
int c = 1;                                              O(1)
while(c < n)                                            O(log n)
{
      if(vector[c] < vector[n])                         O(1)
      {
                  aux = vector[n];                      O(1)
                  vector[n] = vector[c];                O(1)
                  vector[c] = aux;                      O(1)
      }
                  c = c * 2;
}
```

Para este ejemplo al principio el valor de c es 1, al cabo de x iteraciones será 2^x ⇒ el número de iteraciones es tal que $n \leq 2^x$ donde x es el entero inmediato superior de *n*; x = $\log_2 n$ iteraciones, ⇒ para este caso es:

$O(1) * O(\log n) * O(1) = O(\log n)$, complejidad logarítmica.

- *Ejemplo:* Bucles donde la evolución de la variable de control es descendente no lineal.

```
int c = n;                                  O(1)
while(c > 1)                                O(log n)
{
            vector[c] = c;                  O(1)
            c = c / 2;                      O(1)
}
```

Para este ejemplo al principio el valor de c es igual a *n*, al cabo de x iteraciones será $n*2^{-x}$ $\Rightarrow$ el número de iteraciones es tal que $n*2^{-x} \leq n$; un razonamiento análogo nos lleva a $\log_2$ n iteraciones, $\Rightarrow$ para este caso es:

O(1) * O(log n) * O(1) = O(log n), complejidad logarítmica.

- *Ejemplo:* Bucles donde la evolución de la variable de control no es lineal, junto a bucles con evolución de variable lineal.

```
int c, x;                                   O(1)
for(int i= 0; i < n; i++)                   O(n)
{
c = i;                                      O(1)
      while(c > 0)                          O(log n)
      {
            x = x % c;                      O(1)
            c = c / 2;                      O(1)
      }
      x = x + 2;                            O(1)
}
```

Tenemos un bucle interno de orden O(log n) que se ejecuta O(log n) veces, luego el conjunto de ordenes es de orden:

O(1) * O(n) * O(log n) = O(n log n), complejidad cuasi-lineal.

EJERCICIOS DE APLICACIÓN

OBJETIVOS:

A. Cimentar los conocimientos sobre las reglas para evaluar el tiempo de ejecución algorítmico.

B. Calcular y analizar la Complejidad Espacial de algunos algoritmos.

1. ***Para los siguientes fragmentos de algoritmos codificados en C++ calcule la complejidad:***

A)

```
for(int i = 1; i <= n; i++){
      for(int j = 1; j <= n; j++){
            for(int k = 1; k <= n; k++)
                  x = x + 1;
      }
}
```

B)

```
if(n%2 == 0){
      for(i = 1; i <= n; i++)
            x = x + 1;
}
```

C)

```
int i = 1;
      while( i <= n){
            x = x + 1;
            y = x * y + i;
            i = i + 2;
      }
```

D)

```
int a, x = 1;
      while(x < n)
            x = 2 * x;
```

E)

```
int z = 1,x = 1, k = 100;
      for(x = 1; x < k; x++){
                  while(z < n){
                        x = 2 * x;
                        z = z**2;}}
```

2. ***Dados los siguientes algoritmos de ordenación codificados en C++, calcule el tiempo de ejecución de cada uno:***

A)

```
void ALGORITMO1(void){
int x, z, aux;
      for(x = 0; x < RANGO; x++){
            for(z = RANGO - 1; z >= 0; z--){
                  if(vector[z] < vector[z - 1]) {
                        aux = vector[z];
                        vector[z] = vector[z - 1];
                        vector[z - 1] = aux;
                        GET_CHANGE();
                  }
            }
      }
}
```

B)

```
void ALGORITMO2(void){
int i, j, min, aux;
      for(i = 0; i < RANGO; i++){
            min = i; j = i + 1;
            while (j < RANGO){
                  if (vector[j] < vector[i]){
                        min = j;
                        aux = vector[i];
                        vector[i] = vector[min];
                        vector[min] = aux;
                  }
            j++;
            }
      }
}
```

C)

```
void ALGORITMO3(void){
int z, x, aux; bool flag;
      for(x = 1; x < RANGO; x++){
      aux = vector[x];  z = x - 1;  flag = false;
            while(flag == false && z >= 0){
                  if(aux < vector[z]){
                        vector[z + 1] = vector[z];
                        z--;
                  }
                  else
                        flag = true;
            }
      vector[z + 1] = aux;
      }
}
```

D)

```
void ALGORITMO4(void){
int intervalo, k, j, i, aux;  intervalo = RANGO / 2;
while(intervalo > 0){
      for(i = intervalo - 1; i < RANGO; i++){
      j = i - intervalo;
            while(j >= 0){
                  k = j + intervalo;
                  if(vector[k] <= vector[j]){
                        aux = vector[j];
                        vector[j] = vector[k];
                        vector[k] = aux;
                  }
                  else
                        j = 0;
            j = j - intervalo;
            }
      }
      intervalo = intervalo / 2;
}
}
```

3. ***Las combinaciones son agrupaciones de objetos en las que no importa su orden. Tomando con ejemplo del reparto de cartas, normalmente no importa el orden en que se reciben éstas. El número de posibles combinaciones de la mano recibida por un jugador es igual al número de variaciones en que las cartas se podían haber repartido, dividido por el número de posibles formas de ordenar la mano. Por ejemplo, hay $V_{40,4}$ formas de repartir 4 cartas de una baraja de 40, y hay P_4 de ordenar dichas cartas. Por tanto, hay $V_{40,4}/P_4$ posibles combinaciones. En general, el número de combinaciones de n elementos tomados de m en m se escribe $C_{n,m}$, y su valor está dado por la siguiente fórmula:***

$$\begin{pmatrix} v_{n,m} \\ P_m \end{pmatrix} = \frac{n!}{m!(n-m)!}$$

Evalúe la complejidad de un algoritmo que calcula un número combinatorio, también implemente el algoritmo en C++.

```
PROCEDIMIENTO NumComb(n, m)
INICIO
    SI n < m ENTONCES
            Comb = 0
    SINO
    Result1 = Factorial(n)
    Result2 = Factorial(m)
    Result3 = Factorial(n - m)
    Comb = Result1 / (Result2 * Result3)
    FIN SI
FIN


FUNCION Factorial(n)
INICIO
        fact = 1
        SI n > 0 ENTONCES
        HACER desde i = 1 HASTA n
                fact = fact * i
        FIN HACER
        FIN SI
DEVOLVER fact
FIN
```