Programación II
Práctica 01: Complejidad algorítmica

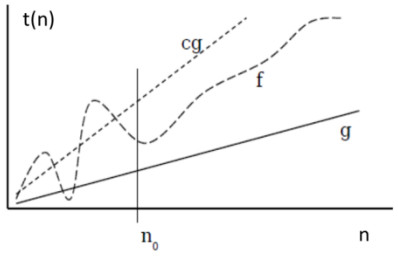
Notación O → definición

$f \in O(g) \Leftrightarrow \exists n_0, c \in \mathbb{N}^+$ tal que, $\forall n \geq n_0$

$f(n) \leq c.g(n)$

$n_0 > 0, c > 0 \cdot tales \cdot que \cdot (\forall n \geq n_0)\, f(n) \leq cg(n)\}$

Donde n es el tamaño de la instancia del problema con la que se ejecuta el algoritmo.



Ejercicio 1: Dadas los siguientes algoritmos se pide utilizar la definición anterior para demostrar su complejidad expresada con la notación O. El análisis se debe realizar para el peor de caso. Explicar de forma precisa cuál es el peor caso de cada uno y quién es la variable de complejidad **n**.

a)
```
static int cuantasApariciones(int[] vec, int x) {
        int cont=0;
        for (int i=0; i<vec. length; i++ )
                if (vec[i] == x)
                        cont++;
        return cont;}
```

b)
```
static int mayoresPromedio(double[] vec) {
        double prom=0;  int cont=0;

        for (int i=0; i<vec.length; i++ )
                prom=prom+vec[i];

        prom=prom/vec.length;

        for(int i=0; i<vec.length; i++ )
                if(vec[i]>prom)
                        cont++;
        return cont;}
```

c)
```
static boolean repetidosEnTodasLasFilas(int[] vec) {
        boolean tieneIgual, todasTienen=true;

        for (int i=0; i<vec. length; i++) {
                tieneIgual=false;
                for (int j=i+1; j<vec. length; j++)
                        tieneIgual=tieneIgual||vec[i]==vec[j];
                todasTienen = todasTienen && tieneIgual;
        }
        return todasTienen;}
```

d)
```
static void cambiaVector(int[] vec) {
        //siempre vec!=null
        if(vec. length % 2==0)
                for (int i=0; i<vec. length; i= i + 2)
                        vec[i] = vec[i] *2;
        else
                for (int i=1; i<vec. length; i= i * 2)
                        vec[i] = vec[i] * (-1);
}
```

Ejercicio 2: Utilizando la definición de complejidad, encontrar n_0 y c para justificar el orden de complejidad resultante de las siguientes expresiones de f(n) utilizando la notación O, donde n es la variable de complejidad en cada caso.

a) $n^2 – n^2 + 100$
b) $n^3 + 2n^2 + 10$
c) $n^{3/2} + n^{1/2} + 100$
d) $3n^2 – n \log n + 10$
e) $\sqrt{n} + \log n + 1000$
f) $n^n + n^{10} + 10$

Respuesta: (Se debe demostrar)
a) $O(1)$
b) $O(n^3)$
c) $O(n^{3/2})$
d) $O(n \log n)$
e) $O(n^{1/2})$
f) $O(n^n)$

Ejercicio 3: Burbujeo (Por definición de O)

a) ¿Cuál es el orden de complejidad del siguiente algoritmo? Demostrar por definición y explicar el peor caso.

```
void ordenarPorBurbujeo(int a[]) {
    for (int i = 1; i <  a.length; i++) {
        for (int j = 0; j < a.length-1; j++)  {
            if (a[j] > a[j+1])
                swap(a[j], a[j+1]);
        }
    }
}
```

```
Void swap( int n, int m){
        int aux = n;
        n = m;
        m = aux;
}
```

b) ¿Cambiará el orden de complejidad encontrada si cambiamos j < a.length-1 por **j < a.length-i**, en el segundo "for"? Demostrarlo para este caso.

Ejercicio 4: Búsqueda Binaria (Por definición de O)

Demostrar por definición el orden de complejidad del algoritmo de búsqueda binaria.

Reglas y Álgebra de Órdenes

1) $O(1) \leq O(\log n) \leq O(n) \leq O(n^k) \leq O(k^n) \leq O(n!) \leq O(n^n)$

2) $O(f) + O(g) = O(f + g) = O(\max\{f, g\})$

3) $O(f)\ O(g) = O(f\ g)$

4) $O(k) = O(1)$ para todo k constante.

Ejemplo: O(2n)
= O(2) * O(n) // por regla 3
= O(1) * O(n) // por regla 4
= O(n)

5) $\sum_{i=1}^{k} O(f) = O(\sum_{i=1}^{k} f) = O(k\ f)$
Si k es constante, entonces vale O(f)

6) $\sum_{i=1}^{k} i = \frac{k*(k+1)}{2} = O(k^2)$

Ejercicio 5: Utilizando las reglas de órdenes para demostrar el orden de complejidad resultante de las siguientes expresiones de f(n). Se debe expresar con la notación O y n es la variable de complejidad en cada caso.

a) $n^2 – n^2 + 100$
b) $n^{3/2} + n^{1/2} + 100$
c) $n^3 + 2n^2 + 10$
d) $\sqrt{n} + \log n + 1000$
e) $n^n + n^{10} + 10$

Variables de complejidad

Como ya se utilizaron en los ejercicios anteriores las variables de complejidad son las que representan el tamaño de los datos de entrada el algoritmo.

La función f(…) que expresa la cantidad de instrucciones del algoritmo , estará en función de dichas variables. En muchos algoritmos se expresará en función de más de una de estas variables ya que puede depender del tamaño de más de una estructura de datos.

Ejercicio 6: Demostrar mediante álgebra de órdenes

```
Void función1(k, h)
        for (i=0, i < k, i++)
                h++;
```

Los candidatos a variables de complejidad son k y h, pero como h++ está en O(1)[1] , la complejidad no depende de h. Así que la variable de complejidad es **k** y de esto surge que el ciclo se repite **k** veces.

Demostrar que la complejidad de funcion1 es **O(k)**

Ejercicio 7: Demostrar mediante álgebra de órdenes

```
Void función2(k, h)
        for (i=0, i < 2^k, i++)
                h++
```

Notar que el ciclo no termina en k pasos. (Explicar)

¿Cuál es la complejidad de función2? Demostrarla.

EJEMPLO que extiende el comportamiento de las funciones con dos variables: (no se pide demostración)

```
Void función3(k, h)
        for (i=0, i < k, i++) {
                O(1) }
        for (i=0, i < h, i++) {
                O(1) }
```

El primer ciclo depende de k, pero el segundo ciclo depende de h.

Entonces ¿qué variable usaremos?

En este caso se necesitan las dos.

Pero ¿cuál es la complejidad? ¿O(k) u O(h)? ¿Cómo saber si k > h o si h > k ?

La realidad es que en general no se sabe.

Entonces tendremos que poner alguna de las siguientes expresiones

(1) O(k + h)
(2) O(max(k, h))

[1] O sea que no depende del tamaño del problema

La segunda expresión es más precisa.

Supongamos como ejemplo que k > h. claramente k +h sigue siendo mayor que k.

Entonces como puede ser que la complejidad sea O(k)

Vamos a buscar una cota:

Como k > h, sabemos que 2k > k + h

Entonces la complejidad es O(2k) , pero por el álgebra de la complejidad es lo mismo que O(k)

$$O(2k) = O(2)\ O(k) = O(1)\ O(k) = O(k)$$

Ejercicio 8: Desafío … Varias variables

Implementar un algoritmo que recorra e imprima los elementos de una matriz de n filas y k columnas. Calcular la complejidad de dicho algoritmo utilizando la definición de complejidad para dos variables.

$f_{n,k} \in O(g_{n,k}) \Leftrightarrow \exists\ n_0, k_0, c$ tales que para todo $n > n_0, k > k_0 \Rightarrow f(n,k) < c\ g(n,k)$

Ejercicio 9: Caja de fósforos

Se tiene una caja de fósforos con n fósforos nuevos.
Cada vez que quiera utilizar uno, el procedimiento es el siguiente:
Tomo un fosforo de la caja. Si está quemado, tomo otro, y así hasta encontrar uno nuevo.
Luego utilizo el fosforo y lo mezclo junto con los otros fósforos usados en la caja

a) ¿Cuál es la complejidad de encontrar un fósforo sin quemar dado que ya consumí la mitad de la caja?
b) ¿Cuál es la complejidad de consumir n fósforos?

Ejercicio 10: Desafío … Combinatoria

En una habitación hay **n** personas que se quieren saludar entre sí.
Las reglas para saludarse son:

1) Cada persona puede saludar solo una persona a la vez.
2) El saludo es simétrico: Si **a** saluda a **b**, se considera que b saluda a **a** en el mismo saludo.
3) Cada saludo demora 1 segundo
4) Si hay **n** personas, puede haber hasta **n/2** saludos simultáneos.

a) Cuanto tiempo (en segundos) se necesita que todos queden saludados?

Ayuda: Hacer una tabla para el caso de 4 y de 8 personas.

b) Cuantos saludos hay en total?

c) Como cambia a) si cambiamos la regla 4) de manera que solo puede haber un saludo a la vez?

d) Como cambia a) si cambiamos la regla 1) de manera que cada persona puede saludar a más de una persona a la vez?

e) Como cambia a) si quitamos la regla 1) y además, cada persona que es saludada sale de la habitación.

Ejercicio 11: Ejercicio de parcial

Calcular la complejidad computacional del siguiente algoritmo, **justificando por medio del álgebra de órdenes**. (No se pide contar instrucciones). Explique qué es **n**.

Recuerde que las instrucciones que no dependen del tamaño de los arreglos son O(1), solo debe considerar cuántas veces se ejecutan.

```
public static ArrayList<Integer>   soloNoRepetidos(ArrayList<Integer> vec) {
    ArrayList<Integer> auxVec  = new ArrayList<Integer>();
    for (int i=0; i<vec.size(); i++) {
        if (cuantasAparece(vec, i)==1)
            auxVec.add(vec.get(i));
         }
    for (int j=0; j<auxVec.size(); j++ ) {
        System.out.println(auxVec.get(j));
    }
  return auxVec;
}
```

Donde:
private static int cuantasAparece(ArrayList<Integer> vec, int i) es un método **O(n),** que retorna la cantidad de veces que aparece en el arreglo *vec*, el elemento que está en la posición *i*
El método *add* y el método *get* de ArrayList son O(1). System.out.println es O(1).

Ejercicio 12: Ejercicio de parcial con matrices

Calcular la complejidad computacional del siguiente algoritmo, **justificando por medio del álgebra de órdenes**. (No se pide contar instrucciones). Explique qué es n. La matriz es cuadrada.

```
static boolean filasConAlgunCero(int[][] mat) {
//ver si todas las filas tiene algun cero
```

```
        boolean hayCero = false;   boolean todas = true;
        for (int f=0; f < mat.length;f++) { // recorre las filas
                hayCero = false;
                for (int c=0; c < mat[0].length; c++)  //recorre las columnas
                        hayCero = hayCero || mat[f][c] == 0;
                todas = todas && hayCero;
        }
        return todas;
}
```

Nota: la matriz cuadrada significa que tiene igual cantidad de filas que de columnas.

Recuerde que las instrucciones que no dependen del tamaño de los arreglos son O(1). Por ejemplo, hayCero = hayCero || mat[f][c] == 0; no depende de la cantidad de elementos de la matriz por lo tanto es O(1). Lo que se debe considerar es cuantas veces se realiza esa instrucción.

Ejercicio 13: Ejercicio de parcial con matrices

```
void ordenarFilas(int[][] mat) {
        int max = mat[0][0];
        for (int f = 0; f < mat.length; f++) {
                ordenarFila(mat[f]);
        }
}
void ordenarFila(int[] fila){
        for (int i=0; i<fila.length;i++)
                ordenarNum(fila,i);
}
void ordenarNum(int[] fila, int pos) {
        for ( int i=pos+1; i < fila.length; i++) {
                if (fila[pos]>fila[i]){
                        int aux=fila[pos];
                        fila[pos]=fila[i];
                        fila[i]=aux;
                }
}       }
```

i. Se pide contar la cantidad de instrucciones que ejecuta la función ***void ordenarFilas(int[][] mat)***.
Asumir que la matriz es siempre cuadrada.
ii. Calcular el orden de complejidad aplicando la definición. Contando instrucciones.
iii. ¿Cuál es el peor caso?

Debe dejar en claro cual es la f(n) resultante, o sea de ordenarFilas(…) Indicar quién es g, c y n0 en su justificación. Expresar la complejidad resultante con notación O.