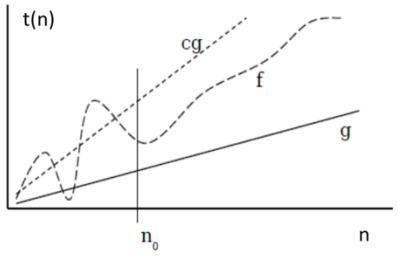
Programación II
Práctica 01: Complejidad algorítmica

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 las siguientes funciones en pseudocódigo, analizar y determinar la complejidad para el peor de caso.

a)
```
void funciónA(int[] arr, int n){
// n <= entero(raiz(arr.length))

    if arr.length = 1
        for (i = 1; i<n; i++)
           arr[i]= 0;
    else
        for (i = 1; i<n*n; i++)
           arr[i]= i*i;
}
```

b)
```
void funciónB(int[] arr, int n){
    if arr.length != 1
      for (i = 1; i<n; i++)
          arr[i]= 0;

    else
      for (i = 1; i<n*n; i++)
          arr[i]= i*i;
}
```

c)
```
void funciónC(int[] arr, int n){
// arr nunca esta vacio.

  int suma=0; int promedio;

  for (i = 0; i< arr.length; i++)
      suma = suma + arr[i];

  promedio=entero(suma/ arr.length);

  for (i = 0; i< arr.length; i++)
      if(arr[i] > promedio)
           arr[i] = 0;
}
```

Desafío: No se pide hacer una demostración sino analizar cuál es la complejidad de funciónD

d)
```
void funciónD(int x)
      if f(x)
            g(x);
      else
            h(x);
```

Ejercicio 2 : Por definición de O

Utilizando la definición de O ($f \in O(g) \Leftrightarrow \exists\ n_0$, c tales que para todo $n > n_0 \Rightarrow f(n) < c\ g(n)$), Encontrar n_0 y c para justificar el orden de los siguientes tiempos de ejecución.
Decidir en qué Orden(el más chico) están.

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) $n^n + n^{10} + 10$

Ejercicio 3 : Burbujeo (Por definición de O)

a) ¿Cuál es el orden de complejidad del siguiente algoritmo? Demostrar por definición

```
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) ¿Cambia el orden de complejidad 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)

Variables de complejidad

Antes de continuar identificaremos la o las variables que representan el tamaño de los datos de entrada el algoritmo.

La función f() que mide la complejidad , estará en función de dichas variables.

Ejemplo1

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

Los candidatos a variables 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.

Variable de complejidad: **k**
La complejidad de funcion1 es **O(k)**

Ejercicio 5-a:
Demostrar mediante álgebra de órdenes

Ejemplo2

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

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

La complejidad de funcion2 es **O(2^k)**

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

Ejercicio 5-b:

Demostrar mediante álgebra de órdenes

Ejemplo3

```
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))**

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 6: Varias variables

Implementar un algoritmo que recorra y muestre una matriz de n filas y k columnas.
Calcular la complejidad de dicho algoritmo utilizando la definición de O para dos variables

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

Ayuda: Nombrar las variables antes de calcular la complejidad

Ejercicio 7: 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 fosforo sin quemar dado que ya consumí la mitad de la caja?
b) ¿Cuál es la complejidad de consumir n fósforos?

Desafío:

Ejercicio 8: 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 9: 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**.

```
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).

Ejercicio 10: 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.