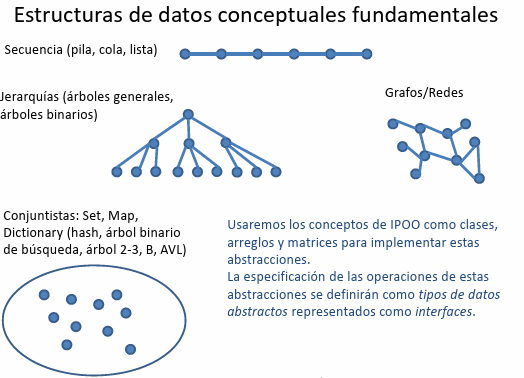


RESUMEN DE GP, EXCEPCIONES, RECORRIDOS, T(N) Y O(N):

**INTRODUCCION A ESTRUCTURAS DE DATOS (TEMA N°1):**

**ESTRUCTURA DE DATOS:** Manera sistemática de organizar y acceder datos



**ALGORITMO:** Procedimiento paso a paso para realizar una tarea en una cantidad finita de tiempo 🡪 Especificacion de un conjunto de pasos (instrucciones) necesarios para resolver un problema o una tarea en una cantidad finite de tiempo 🡪 Nuevas restricciones [Cuantificar tiempo del procedimiento (lo (+) pequeño posible, orden del tiempo de ejecucion)]

**FACTORES DE CALIDAD DEL SOFTWARE:**

* Usaremos POO porque permite construir software de buena Calidad
* ¿Cómo evaluar la calidad del software?

1. Factores externos de calidad: Cualidad del software que puede ser detectada por un usuario (sin conocer el código fuente)
2. Factores internos de calidad: Cualidad del software que puede ser percibida por el programador (conociendo el código fuente)

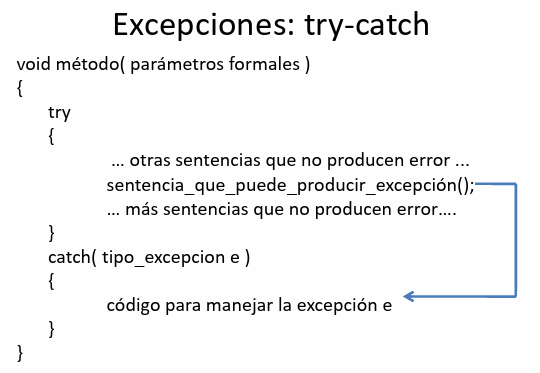
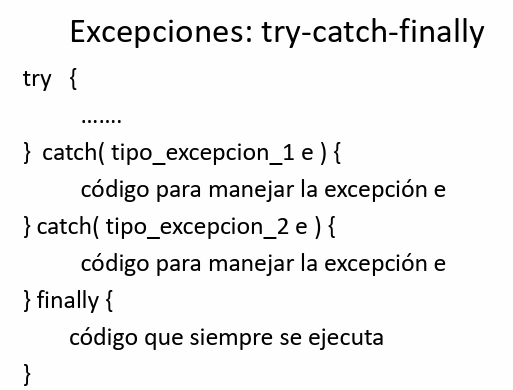
* **TIPOS DE FACTORES DE CALIDAD:**

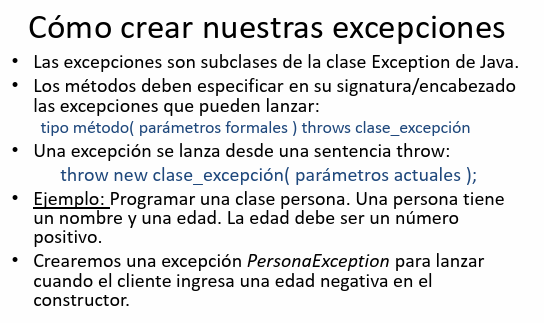
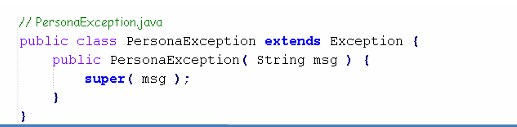
1. Correctitud: El programa respeta la especificación (para una entrada dada produce la salida esperada)[Enunciado]
2. Robustez: El programa es capaz de manejar entradas no contempladas por la especificación (se espera que se degrade “graciosamente” o “gracefully”). Ejemplo: Recibe un string cuando espera un entero. Factor crítico en aplicaciones industrials [Evitar que el programa termine en errores de ejecucion]
3. Extensibilidad: Capacidad de agregarle funcionalidad al software [Subclases]
4. Portabilidad: Habilidad del soft de correr con mínimo cambio en otro hard o sistema operativo. (por ello usamos Java)
5. Reusabilidad: Capacidad de usar el código en la construcción de otros programas [Librerias]
6. Eficiencia: Capacidad del soft de requerir pocos recursos de cómputo (tiempo de CPU y memoria) [Orden del tiempo de ejecucion]
7. Facilidad de uso: Capacidad del soft para ser usado por usuarios con distinta habilidad [Construccion de interfaz de ususario]

**EXCEPCIONES [Aportan Robustez]:**

Las excepciones son eventos inesperados que ocurren durante la ejecución de un programa

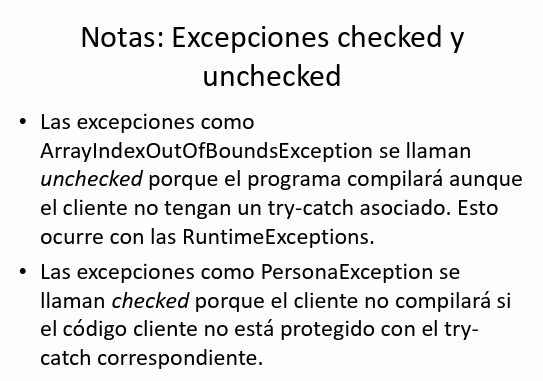
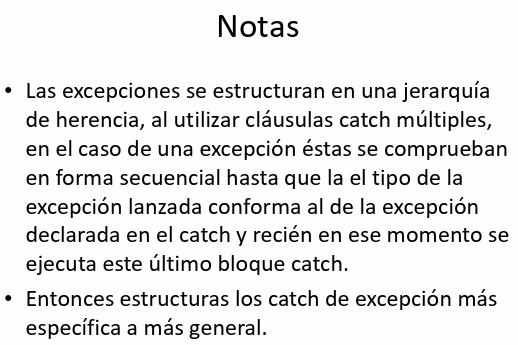
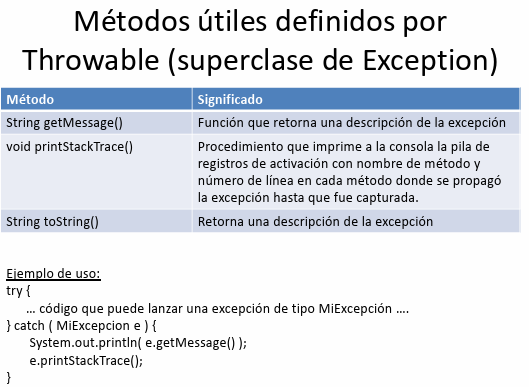
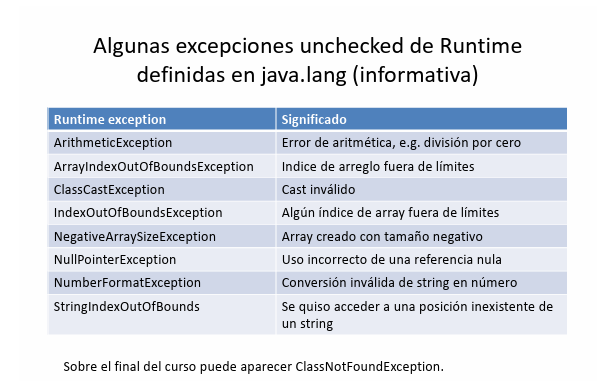
* Puede ser el resultado de una condición de error o una entrada incorrecta
* En POO las excepciones son objetos “lanzados” por un código que detecta una condición inesperada
* La excepción es “capturada” por otro código que repara la situación (o el programa termina con error en ejecución)
* Vamos a tener objetos de tipo excepcion que van a ser lanzados por un codigo que detecta una condicion inesperada; esa excepcion es capturada por otra poscion de codigo que repara la situacion (o termina en error de ejecucion)





EJEMPLO DE COMO CREAR UNA EXCEPCION:

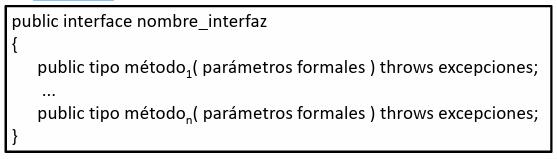
* **¿Que pasa cuando uno no captura una excepcion?** 🡪 Lo que ocurre es que los registros se propagan por la pila de registros de activacion [Memoria en tiempo de ejecucion de un segmento de codigo]

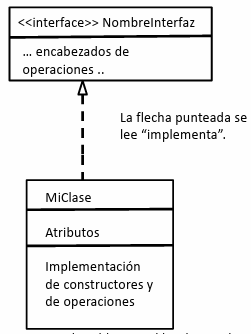


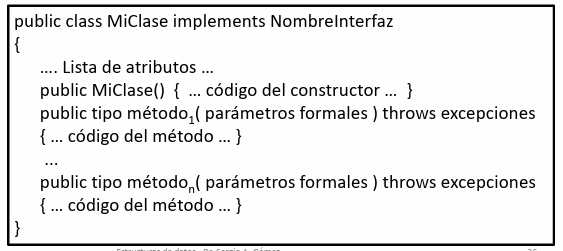
**PAQUETE:** Conjunto de carpetas en las cuales yo separo un proyecto

**INTERFACES:**

* Una interfaz es una colección de declaraciones de métodos



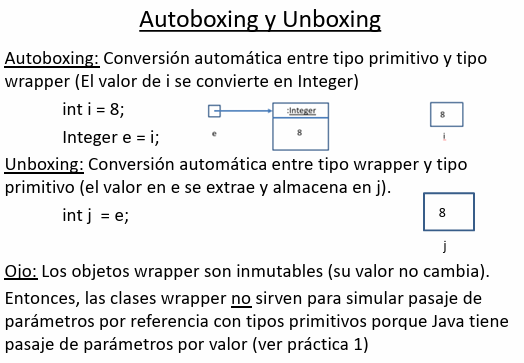
* Una clase que implementa una interfaz debe implementar todos los métodos provistos por la interfaz

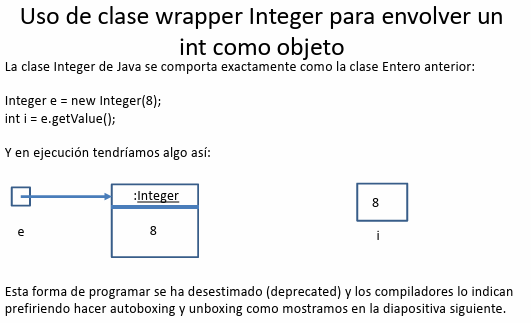


* **Ventajas del uso de interfaces:**

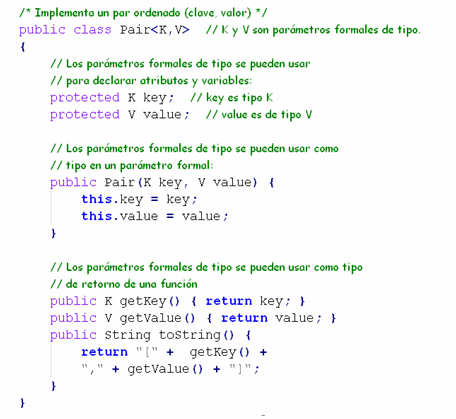
1. Al usar una interfaz uno especifica "qué" debe hacer una clase pero no "cómo" debe hacerlo
2. Las interfaces no hacen suposiciones de "cómo" serán implementadas
3. Una vez que se define una interfaz, ésta puede ser implementada por cualquier número de clases
4. Implementar una interfaz requiere implementar todos los métodos definidos por la misma
5. Brindan una forma limitada y segura de herencia múltiple (una clase puede implementar más de una interfaz)
6. Las interfaces están pensadas para soportar la resolución de métodos en tiempo de ejecución
7. Esto se puede lograr con clases abstractas también pero las mismas deberían estar más y más arriba en la jerarquía oscurenciéndola (este problema se evita con las interfaces ya que dos clases no relacionadas pueden implementar la misma interfaz)

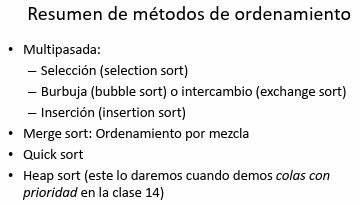
**CLASES WRAPPER:**

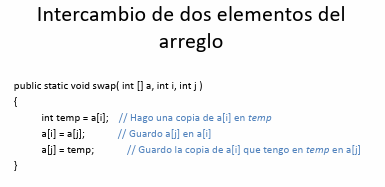
* En ocasiones es muy conveniente poder tratar los datos primitivos (int, boolean, float, char, etc.) como objetos
* Las clases wrapper existentes son, Byte para byte, Short para short, Integer para int, Long para long, Boolean para boolean, Float para float, Double para double y Character para char
* **Nota:** Los objetos de clases wrapper son inmutables (no están pensadas para actualizar los valores envueltos)

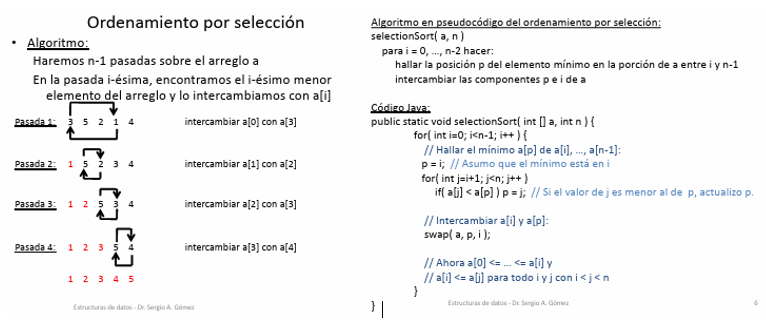


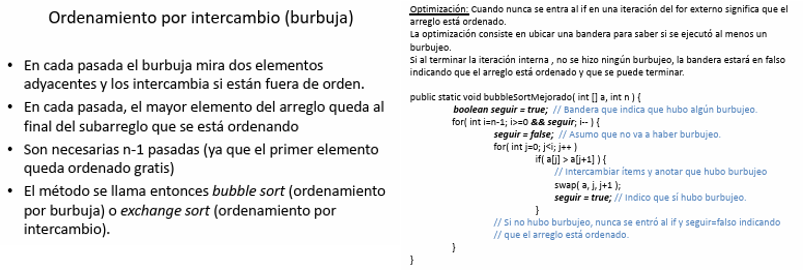
**GENERICIDAD PARAMETRICA:**

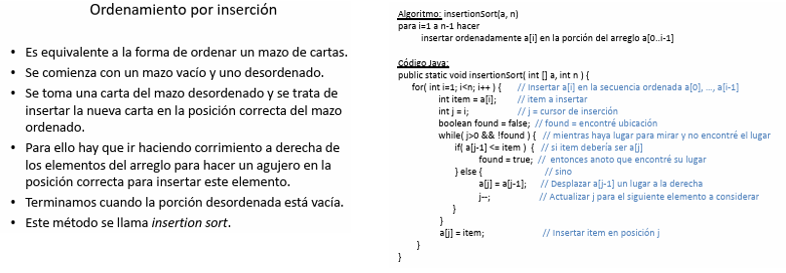
* Se utiliza para poder usar tipos de dato abstracto que evitan el castin explicito
* Un tipo genérico es un tipo que no es definido en compilación sino en tiempo de ejecución y permite definir una clase en función de parámetros formales de tipo que abstraen los tipos de algunos datos internos de una clase
* Dada una clase genérica, vamos instanciar objetos usando parámetros actuales de tipo
* **Ventaja de la genericidad parametrica:** Con la genericidad por polimorfismo el descubrimiento de los errores se hace en ejecucion, mientras que con la genericidad parametrica descubrir los errores se hacen en compilacion

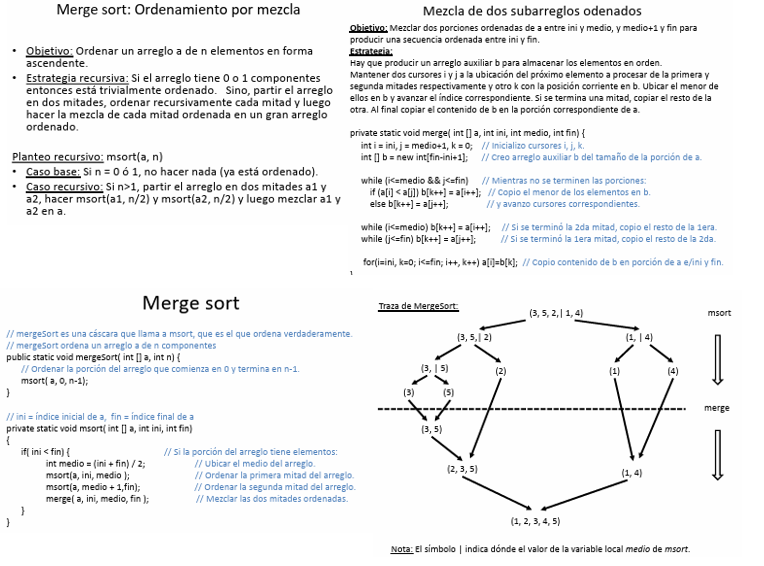
**RECORRIDOS Y ORDENAMIENTO DE ALGORITMOS:**

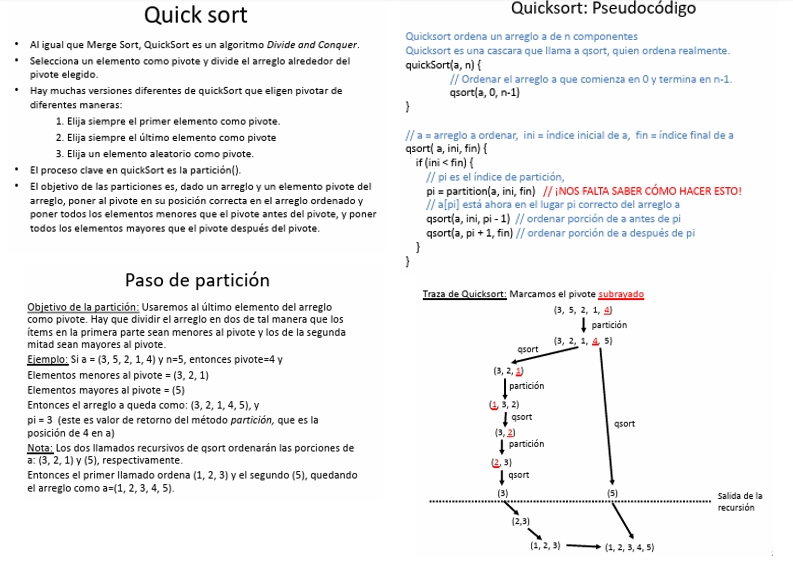












**ALISIS DE ALGORITMOS (TIEMPOS DE EJECUCION Y ORDEN DEL TIEMPO DE EJECUCION) (TEMA N°2):**

**OBJETIVO:** Una aplicación debe correr lo más rápidamente posible

**FACTORES QUE AFECTAN EL TIEMPO DE EJECUCION:**

* Aumenta con el tamaño de la entrada de un algoritmo
* Puede variar para distintas entradas del mismo tamaño
* Depende del hardware
* Depende del sistema operative
* Depende de la calidad del código generado por el compilador
* Depende de si el código es compilado o interpretado

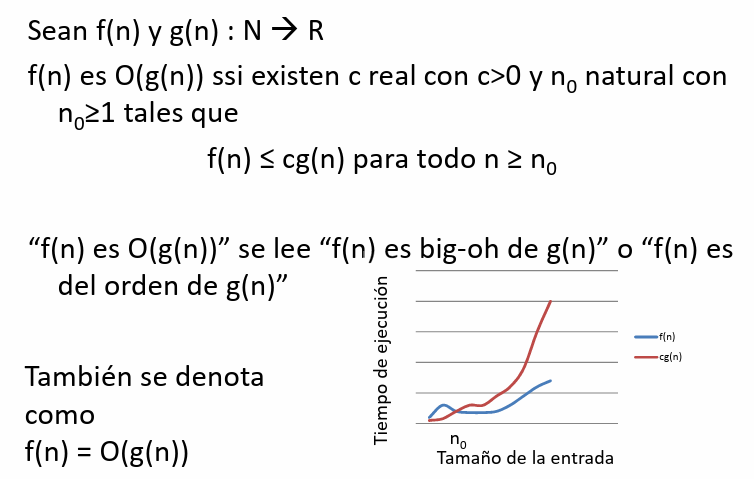
**ORDEN DEL TIEMPO DE EJECUCION:**

* Es una forma de medir el tiempo de ejecucion de un algoritmo, para evaluar su eficiencia
* Toma en cuenta todas las posibles entradas
* Permite evaluar la eficiencia relativa de dos algoritmos independientemente del ambiente de hardware y software
* Puede realizarse estudiando una versión de alto nivel del algoritmo sin necesidad de implementarlo o hacer experimentos



**TIEMPO DE UN ALGORITMO:**

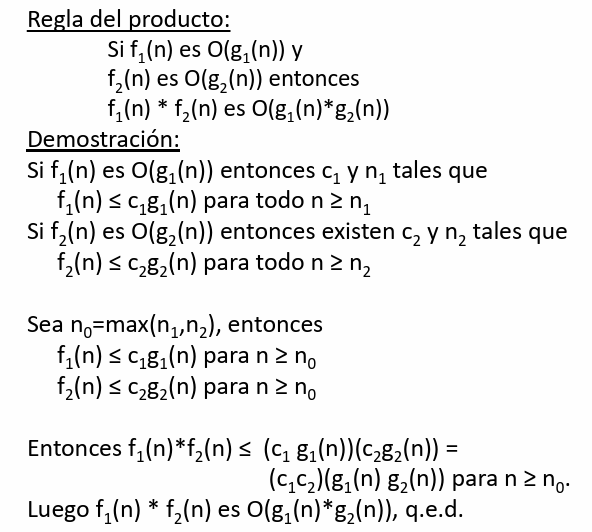
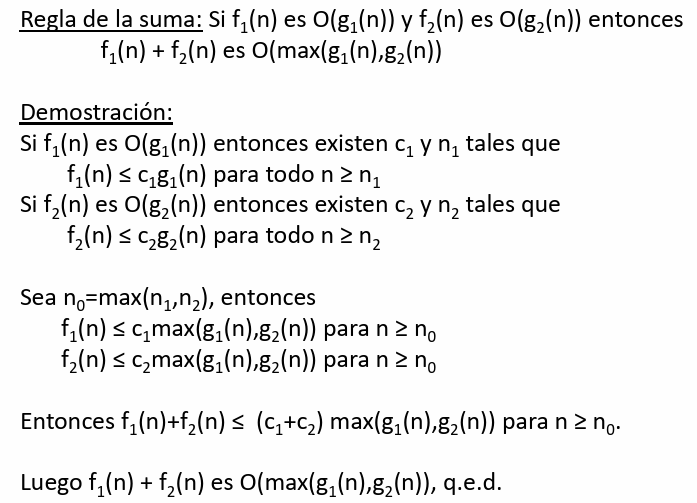
* El tiempo de ejecución de un algoritmo depende de la cantidad de operaciones primitivas realizadas
* Las operaciones primitivas toman tiempo constante y son:
* Asignar un valor a una variable
* Invocar un método
* Realizar una operación aritmética
* Comparar dos números
* Indexar un arreglo
* Seguir una referencia de objeto
* Retornar de un método
* Como estos tiempos dependen del compilador y del hardware subyacente, por ello los notaremos con constantes arbitrarias c1, c2, c3, …
* Tambien depende operaciones no primitivas, como los ciclos de iteracion [con tiempo dependiente del tipo, usualmente dependen del tamaño de la entrada]

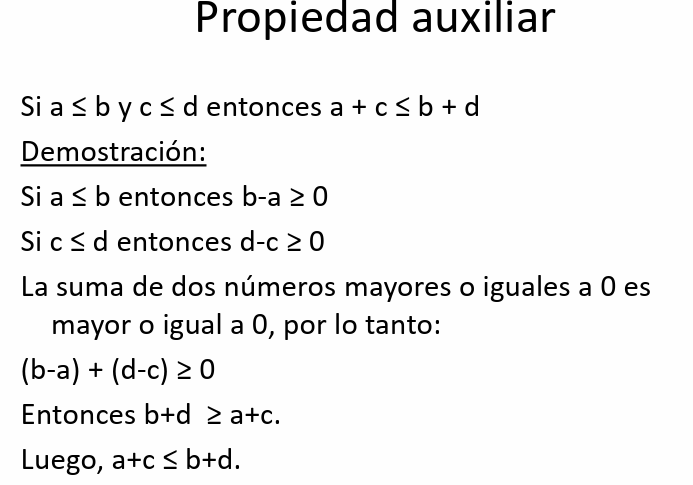
**NOTACION ASINTOTICA BIG-OH:**

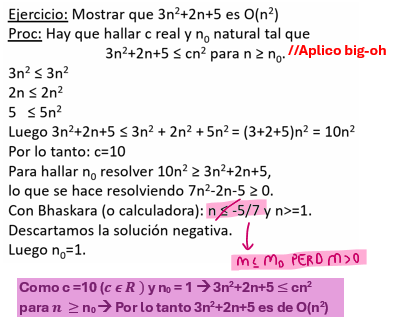
**REGLAS DE LA SUMA Y EL PRODUCTO:**

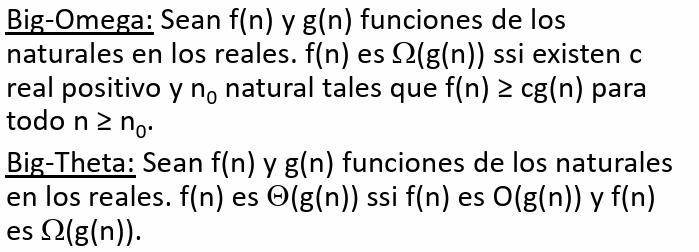
REGLA DE LA SUMA

REGLA DEL PRODUCTO





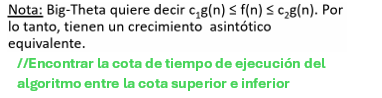


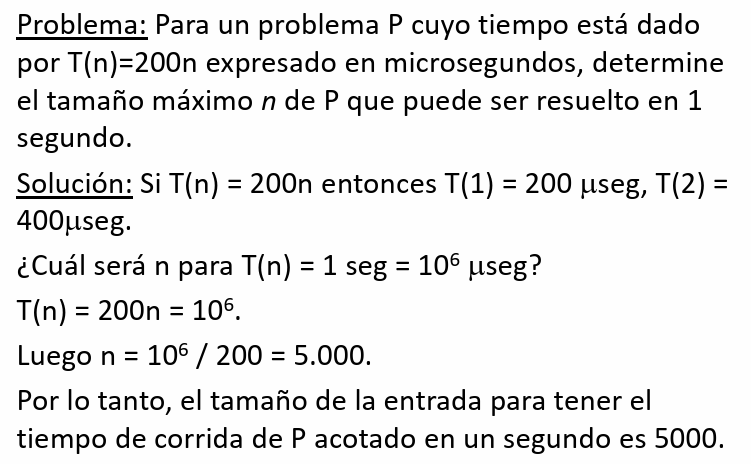
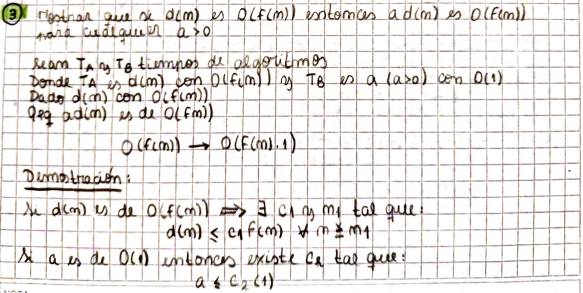
**BIG-OMEGA Y BIG-THETA:**



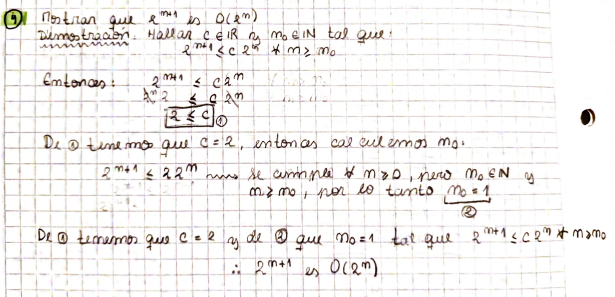
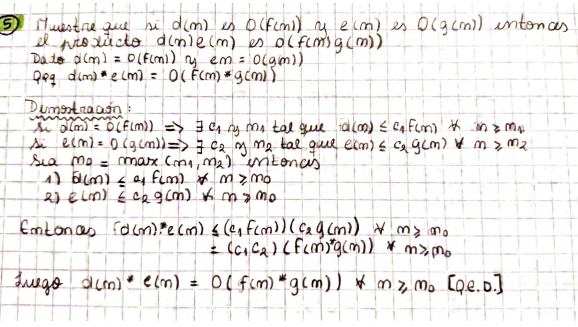
**//Calculo de cota superior**

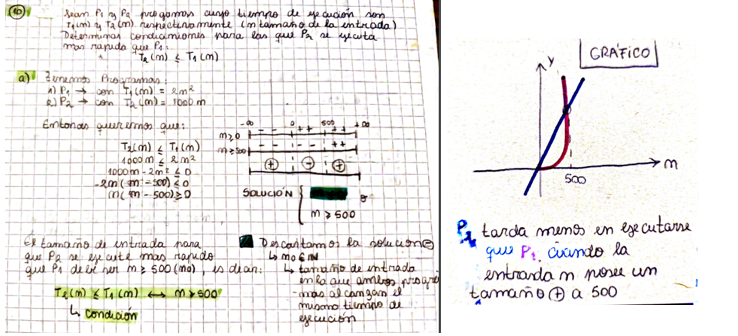
**//Calculo de cota inferior**





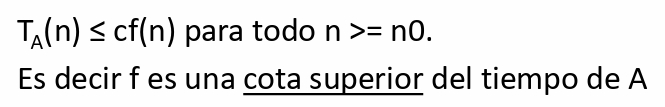






**TIEMPO DE UN ALGORITMO “A”:**

* TA(n) = O(f(n)) (el orden del tiempo de ejecución de un algoritmo A ejecutado sobre una entrada de tamaño n) es del orden de f(n)) cuando el tiempo de corrida del peor caso puede ser acotado por arriba por f(n) multiplicado por un constante c a partir de un cierto tamaño de entrada n0:



* TA(n) = (f(n)) (el tiempo de ejecución de un algoritmo A ejecutado sobre una entrada de tamaño n es de omega de f(n)) cuando el tiempo de corrida del peor caso puede ser acotado por abajo por f(n) multiplicado por un constante c a partir de un cierto tamaño de entrada n0:

