# GUIA I Formalismo, abstracción y complejidad

*Un hombre superior es impasible; le alaban, le censuran: sigue imperterrito su camino*

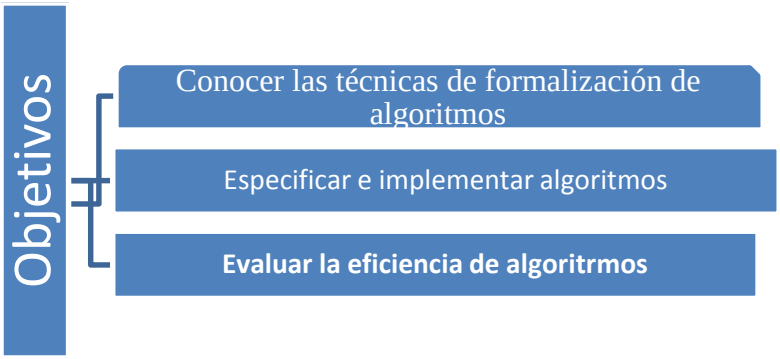
*Napoleon*

**Alan Mathison Turing (1912-1954)**

En 1935, el matemático y lógico ingles Alan Mathison Turing (1912-1954) se intereso en el problema de Hilbert, que preguntaba si podría haber un método general aplicable a cualquier enunciado para determinar si este era verdadero. El enfoque de Turing para la solución de este problema lo llevo a desarrollar lo que ahora se conoce como la *maquina de Turing.* Durante la segunda Guerra Mundial, Turing trabajo en la oficina para asuntos externos de Bletchley Park, donde hizo un amplio uso del criptoanálisis de los mensajes nazis. Sus esfuerzos produjeron una maquina descifradora mecánica, Enigma, elemento muy importante que contribuyo a la caída del Tercer Reich.

"Quien de nosotros no quisiera levantar el velo tras el cual yace escondido el futuro, y asomarse, aunque fuera por un instante, a los próximos avances de nuestra ciencia y a los secretos de su desarrollo ulterior en los siglos futuros?¿Cuáles serán las metas particulares que trataran de alcanzar los líderes del pensamiento matemático de las generaciones futuras?¿Que métodos y nuevos hechos nos depararan los siglos por venir en el ancho y rico campo del pensamiento matemático?"

**Texto Historico David Hilbert**



## Introducción

Antes de planear el programa, debe realizarse una revisión de la necesidad para la existencia del programa y definirse de alguna forma los objetivos que desean lograrse a través del programa (requerimientos). El proceso de plantación de un programa tiene como finalidad lograr el diseño de un programa que cumpla con los requerimientos. Generalmente es conveniente diseñar primero la estructura general del programa y trabajar luego la lógica detallada.
La plantación de un programa es semejante a la planeación de un edificio o una casa. Primero, se realiza un diagrama general en el que se muestran las diferentes habitaciones (las diferentes funciones que deberán realizarse), y la relación entre ellos. A continuación se realiza el diseño detallado de cada habitación. De manera similar es necesario que exista una estructura general para el programa de computadora.

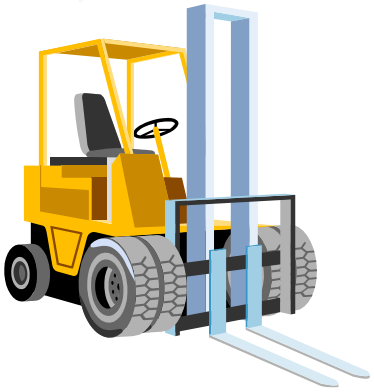
# *PROGRAMAS Y CODIGOS*

## La especificación de un problema

Programar una solución no solo es codificar el programa, sino hallar una solución (algoritmo), especificar la solución y luego implementar la solución. Especificar consiste en esencia, responder a la pregunta *¿que hace el programa?*, no interesa los detalles de cómo sería el programa. **[6]**

## La implementación de un problema

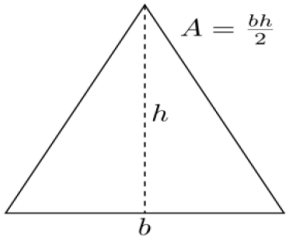
Después de que la especificación ha sido realizada, el siguiente paso consiste en responder *¿cómo se consigue la función pretendida?*, a este paso se le denomina implementación **[6]**



## *Ejemplo 1*

# Hallar el área de un triangulo

Hallar el área de un triangulo




Especificación

Entrada : b, h : reales
Salida : A : real

Diagrama de entrada y salida


**Precondición:** h, b : reales positivos
**FUN AREA**(b, h : real ) Dev ( A : real)
**Poscondición:** A = (b*h) /2

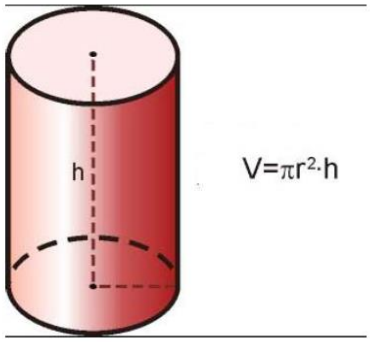
Implementación

```
Fun AREA()
Inicio
        Leer b,h
        A=(b*h)/2
        Escribir A
Fin
```

## *Ejemplo 2*

# Hallar el volumen de un cilindro

Hallar el volumen de un cilindro



Especificación

Entrada : r, h : reales
Salida : V : real

Diagrama de entrada y salida


**Precondición:** h, b : reales positivos
**FUN VOLUMEN**(r, h : real ) Dev ( V : real)
**Poscondición:** V = pi *$r^2$

Implementación

```
Fun VOLUMEN()
Inicio
        Leer r,h
        V = pi *r^2
        Escribir V
Fin
```

La especificación de un algoritmo tiene un doble destinatario:

*Los usuarios del algoritmo*.- La especificación debe incluir todo lo necesario para el uso correcto de los programas, así como las obligaciones del usuario cuando invoque al algoritmo

*El implementador del algoritmo*(programador) .- La especificación debe incluir las restricciones que el implementador debe considerar en el programa.

Una solución mal especificada producirá un programa que no satisface el requerimiento del problema planteado.

Consideremos un lenguaje hipotético como un conjunto que contiene:

1 Todos los operadores aritméticos y funciones de biblioteca
2 Bloques de sentencias que comienzan con una llave { y termina con un }.
3 La estructura de decisión ***Si..Entonces.. Fin_si***
La estructura repetitiva ***Mientras hacer .. Fin_Hacer***

Los programas en este lenguaje trabajan con estados de programas y estos estados se modifican por acción de las partes del código. Las partes del código son funciones que proyecta el estado inicial, que es el estado al comienzo, sobre el estado final, que es el estado al finalizar la parte del código. Para convertirse en una función, las sentencias que comprenden la parte de un código deben ser independientes.

Varias partes del código pueden estar concatenadas, lo que significa simplemente que se ejecutan secuencialmente.

## *ASERCION*

Una aserción es cualquier sentencia referente a un estado del programa, y se denota como {A}

Si **P** es una parte de un código, entonces:

Una ***precondición*** de P, es cualquier aserción {A} que solo implica el estado inicial.

Una ***postcondición*** de P, es cualquier aserción {B} que solo implica el estado final.
Si P tiene como precondición a {A} y postcondición a {B}, escribimos la terna : {A} P {B} , llamada la ***terna de Hoare***.

## *Ejemplo 3*

Considere el código:
{A} x = x+y,y=x-y,x=x-y {B}

{A} es la precondicion
{B} es la postcondicion

es obvio que x, y son modificados por el codigo.
Supongamos que B: { x = **X**; y = **Y**} resultados de x , y despues de ejecutar el codigo.
Puede verse que el codigo realiza el intercambio de los valores de x , y, es decir **X=** x ; **Y =** y.

## *Ejemplo 4*

Considere el código:

a := 1/b precondición {b $\neq 0$}
postcondición {a = 1/b}

***terna de Hoare :*** {b $\neq 0$}*a:=1/b* {a = 1/b}

## *Ejemplo 5*

Queremos especificar un algoritmo que verifique si en una matriz existen dos columnas con igual suma de sus elementos

Definimos TIPO MAT = MATRIZ [1..MAX, 1..MAX] de enteros

A : { $1 \leq$ M,N $\leq$ MAX , M=N, es decir A es cuadrada}
P : **FUN VERIFCACOLU(A**:MAT, M,N:entero) **DEV** (b : booleano)

B : { b $\cong$ $\exists$ p, q $\in$ {1..n} tal que $\sum_{i=1}^{M} A[i,p] = \sum_{j=1}^{M} A[j,q]$ }

### *Ejemplo 6*

Queremos especificar un algoritmo que halla la recta L que pase por dos puntos P y Q

Sea el tipo struct punto{ x: real,
y: real}

Sea el tipo struct recta{ m: real,
b: real}

Sea P = $(X_1,Y_1)$ Q = $(X_2,Y_2)$ La recta L se define y = m x + b
Donde m es La pendiente de L

A : { P,Q son de tipo punto , P ≠Q}
P : **FUN HallarRecta(P,Q:** punto) **DEV** (L : recta)
B : { L : m= $(Y_2-Y_1)/(X_2-X_1)$ ⟶ b = $Y_1$- m $X_1$ }

## *DEBILITAMIENTO DE POSTCONDICIONES*

## *REGLAS DE LAS PRECONDICIONES Y POSTCONDICIONES*

- Si A y B son aserciones y A ⟶ B, entonces se dice que A es mas fuerte que B.

- Si A es mas fuerte que B, entonces se dice que B es mas débil que A

### *REFORZAMIENTO DE PRECONDICIONES*

Si una parte de un código P es correcta bajo una precondición {A}, entonces permanece correcta si se refuerza{A}.

***Si {A} P {B} es correcto. Y se demuestra que***
***$A_1$ ⟶ A , entonces, {$A_1$} P {B} es correcto,***
**es decir**

***$A_1$ ⟶ A***
***{A} P {B}***

***{$A_1$} P {B}***

El programa P que tenga la precondicion mas debil es el programa mas general.

### *Ejemplo 7*

**{x≠0}** *y:=1/x* **{y=1/x}**
**x = 5 ⟶ x ≠ 0**
**{x=5}** *y:=1/x* **{y=1/x}**

**{x ≥ 0}** *y := raíz(x)* **{x = y²}**
**x ≥ 1 ⟶ x ≥ 0**
**{ x ≥ 1}** *y := raíz(x)* **{x = y²}**

**{x<5}** *x:= x -1* **{x <4}**
**x = 4 ⟶ x < 5**
**{x=4}** *x:=x - 1* **{y<4}**

## *Ejemplo 8*

Queremos especificar un algoritmo que calcule el cociente por defecto y el resto de la division entera de dos enteros

A : { b ≠ 0 }
P : **FUN DIVIDE** (a,b : entero) **DEV** (q,r:entero)
B : { a =b*q + r }
Una implementacion de divide puede ser

P : q = 0 ; r = a ,pues satisface la postcondicion B
Sin embargo eso no es lo que se pretende implementar

Reforzamos la precondicion
A : {a ≥ 0 , b ≠ 0 }
P : **FUN DIVIDE** (a,b : entero) **DEV** (q,r:entero)
B : { a = b*q + r ∧ r ≥ 0 ∧ r < b}
Solo existen dos enteros a, b que satisfacen {B}

## *Ejemplo 9*

Queremos especificar un algoritmo que calcule el valor máximo en un vector de enteros tipo VECT
Definimos TIPO VEC = VECTOR [1..100] de enteros

A : { n ≥ 1 ∧ n ≤ 100 }
P : **FUN MAXIMO** (V : VEC) **DEV** (x:entero)
B : { ∀ α ∈ {1..n} . x ≥ V(α) }

B resulta muy debil y aceptaria una implementacion
P : x = K donde K > maximo{V(α)}
Pero no calcula realmente el maximo de V[1..100]

Reforzamos la precondicion

$B : \{\forall\ \alpha \in \{1..n\}\ .\ x \geq V(\alpha)\} \wedge \{\exists\ \beta \in \{1..n\}\ .\ x = V(\beta)\}$

## *DEBILITAMIENTO DE POSTCONDICIONES*

Si una parte de un código P es correcta bajo una postcondición {B}, entonces permanece correcta si se debilita {B}.

**Si *{A} P {B} es correcto. Y se demuestra que***
***B* ⟶ *$B_1$* , *entonces, {A} P { $B_1$} es correcto***
**Es decir**

$$\frac{\{A\}\ P\ \{B\} \qquad B \longrightarrow B_1}{\{A\}\ P\ \{B_1\}}$$

Si el estado final de P saatisface B, siempre que el estado inicial satisface A, y ademas ***B* ⟶ *$B_1$***, entonces el estado final de P debe satisfacer tambien ***$B_1$*.**

## *Ejemplo 10*

Consideremos los ssiguientes ejemplos

· **{}** *mayor = a* **{mayor =a}**
**(mayor = a) ⟶ (mayor ≥ a)**
**{}** *mayor = a* **{mayor ≥ a}**

· **{x>6}** *y= x+2* **{x>6 ,y>8}**
**{y>8} ÿ {x>6} ⟶ {y>8}**
**{x>6}** *y= x+2* **{y>8}**

***REGLA DE CONJUNCION***

**Si P *es un código, y {$A_1$} P {$B_1$} y {$A_2$} P {$B_2$} son correctos, se puede afirmar que {$A_1$ ∧ $A_2$} P {$B_1$ ∧ $B_2$} tambien lo es.* Es decir**

$$\frac{\{A_1\}\ P\ \{B_1\} \qquad \{A_2\}\ P\ \{B_2\}}{\{A_1 \wedge A_2\}\ P\ \{B_1 \wedge B_2\}}$$

## *Ejemplo 11*

***premisa*** ***{} a:=a+2 { $a_f$:=$a_i$+2}***
***premisa*** ***{$a_i$ >0} a:=a+2 { $a_i$ >0}***

***conclusion*** $\{a_i > 0\}\ a:=a+2\ \{(a_i > 0) \Rightarrow (a_f := a_i + 2)\}$

$a_i$ ***: estado actual*** $a_f$ ***: estado final***

podemos verificar que la postcondicion se debilita, produciendo una nueva postcondicion

| | | |
|---|---|---|
| ***1*** | $(a_i > 0) \Rightarrow (a_f := a_i + 2)$ | ***postcondicion*** |
| ***2*** | $(a_i > 0) \Rightarrow (a_i := a_f - 2)$ | ***aritmetica*** |
| ***3*** | $(a_f - 2 > 0) \Rightarrow (a_i := a_f - 2)$ | ***reemplazo*** |
| ***4*** | $(a_f - 2 > 0)$ | ***simplificacion*** |
| ***5*** | $(a_f > 2)$ | |

el debilitamiento de la postcondicion produce que:
***conclusion*** $\{a_i > 0\}\ a:=a+2\ \{a_i > 0\}$

# *COMPLEJIDAD ALGORITMICA*

*TIEMPO DE EJECUCIÓN DE UN PROGRAMA*

**Objetivos que se persiguen al elegir un algoritmo.**

$O_1$ Que el algoritmo sea fácil de entender, codificar y depurar.
$O_2$ Que el algoritmo use eficientemente los recursos del computador y en especial que se ejecute con la mayor rapidez posible.

**CONSIDERACIONES:**

- Si el programa se va a usar pocas veces o una vez $O_1$ es el mas importante. En tal caso el costo de tiempo de programación puede exceder en mucho al costo del tiempo de ejecución del programa.

- Cuando el programa se ve a usar muchas veces el costo de ejecución del programa puede superar en mucho al de escritura, en especial si en la mayor parte de las ejecuciones se dan entradas de gran tamaño. En este caso es más ventajoso desde el punto de vista económico, realizar un algoritmo complejo siempre que el tiempo de ejecución del programa **resultante se**a significativamente menor que el de un programa mas evidente

## FACTORES DEL CUAL DEPENDEN EL TIEMPO DE EJECUCIÓN DE UN PROGRAMA

1. Los datos de entrada al programa
2. Calidad del código generado por el computador utilizado para crear el programa objeto
3. La naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa.
4. La complejidad del tiempo del algoritmo.

## ASPECTOS IMPORTANTES :

1. Si un programa se va a utilizar solo algunas veces, el costo de escritura y depuración es dominante, de manera que el tiempo de ejecución raramente influirá en el costo

total. En este caso debe elegirse el algoritmo que sea mas fácil de aplicar correctamente.

2. Si un programa se va a ejecutar solo con entradas pequeñas la velocidad de crecimiento del tiempo de ejecución puede ser menos importante que el factor constante de la formula de ejecución.

3. Un algoritmo eficiente pero complicado puede no ser apropiado porque posteriormente puede tener que darle mantenimiento otra persona distinta del escritor.

4. Debe considerarse la posibilidad de que un programa resulte inútil debido a que nadie entienda sus sutiles y eficientes algoritmos.

5. Existen ejemplos de algoritmo eficientes que ocupan demasiado espacio para ser aplicados sin almacenamiento secundario lenta, lo cual anula la eficiencia.

6. En algoritmos numéricos, la precisión y estabilidad son tan importantes como la eficiencia.

## Evaluación de Algoritmos

* Elementos a ser tomados en cuenta:

-Eficacia (fin que se busca).

-Eficiencia (tiempo y espacio).

* Eficiencia: Determina la cantidad de recursos computacionales consumidos por el algoritmo. El tiempo y el espacio utilizados en dicha solución miden la mayor o menor eficiencia de un algoritmo.

* Para la evaluación de algoritmos se toma en cuenta:

- La eficiencia en cuanto al uso de recursos de computación.

-La calidad de la solución obtenida.

### Parámetros de la Eficiencia

* Por lo que solo se tomara en cuenta el evaluar a los algoritmos por eficiencia:

- Tasa de crecimiento en tiempo

-Tasa de crecimiento en espacio

* **Tiempo de ejecución:** Tiene que ver con el tiempo que tarda un programa para ejecutarse (tiempo de procesamiento).

* **Espacio de memoria:** Estudia la cantidad de espacio que es necesario para las operaciones durante la ejecución del programa (espacio de almacenamiento).

**Relación entre la Estructura de Datos y el Algoritmo**

* La relación determina un análisis cuantitativo.

*El tipo de estructura de datos elegida es dependiente de:

- Tipo de datos que administra el algoritmo.

-Operaciones que se realizan sobre dicha estructura

-El compromiso espacio de almacenamiento - tiempo de procesamiento que se desea obtener.

## Criterios para definir la complejidad

-**Criterio de Costo Uniforme:** Cada instrucción de la maquina es ejecutada en una unidad de tiempo.

*Sea $x_1$, $x_2$, ... $x_n$ instrucciones; entonces t(xi) = 1 para i = 1, 2, ..., n.

*Entonces el tiempo de ejecución es el número de operaciones.

-**Criterio de Costo Logarítmico:** El tiempo de ejecución como proporcional al tamaño de los datos (número de bits necesario para codificar los datos).

## Introducción al análisis de Eficiencia

-Es sencillo verificar la validez de una supuesta solución que encontrar una solución partiendo de cero.
-El análisis de los algoritmos suele efectuarse desde adentro hacia fuera.
-Primero se determina el tiempo requerido por instrucciones individuales.
-Luego se combinan de acuerdo a las estructuras de control que enlazan las instrucciones.
-Tomaremos como costo unitario una operación que requiera realmente una cantidad de tiempo polinómica.
-Contaremos las sumas y multiplicaciones con un coste unitario, aun en aquellos operándos cuyo tamaño crece con el tamaño del caso, siempre que esté acotado por algún polinomio.
-Si se necesita operándos tan grandes, es preciso descomponerlos n segmentos, guardarlos en un vector, e invertir el tiempo necesario para efectuar la aritmética de precisión múltiple; tales algoritmos no pueden ser de tiempo polinómico.

-Es necesario conocer principios generales para analizar las estructuras de control más frecuentemente conocidas.

### Estructuras Secuenciales

-Sean $P_1$ y $P_2$ dos fragmentos de un algoritmo (instrucciones simples ó una simple y una compuesta).

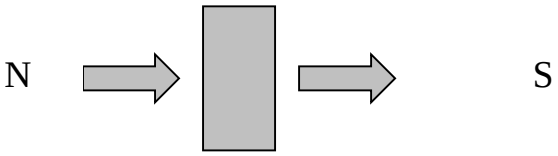
-Sean $t_1$ y $t_2$ los tiempos requeridos por $P_1$ y $P_2$ respectivamente.

-**Regla de la composición secuencial:** El tiempo necesario para calcular "$P_1$;$P_2$" (primero $P_1$ y luego $P_2$) es simplemente $t_1$+ $t_2$.

-Si las instrucciones son compuestas sólo se procede a sumar los tiempos respectivos.

## *Ejemplo 12*

**Hallar la suma y el promedio de los N primeros números enteros positivos**




```
Accion SumaPromedio()
Inicio
        Leer N                  1
        S=0                     2
        Para i desde 1 hasta N  3
                S = S + I       3.1
        FinPara
        P=S/N                   4
Fin
```

**precondición** : N $\in Z^+$
**Postcondicion** : P=S/N

| | | |
|---|---|---|
| 1 | 1 | lectura |
| 2 | 1 | asignación |
| 3 | 5n-1 | |
| 3.1 | 2 | adición y asignación |
| 4 | 2 | división y asignación |

Total 1+1+5n-1+2=5n+3

Paso 3

```
Para i desde 1 hasta N      3
        S = S + I           3.1
FinPara
```

En la primera iteración se realiza 2 operaciones : una asignación (i=1), y un test ( ¿ i < N ? )
En las demas iteraciones (n-1 veces) se realiza 3 operaciones: un incremento( i+1) , una asignación (i=i+1) y un test test (¿ i < N ?)

A esto debemos sumarle el numero de operaciones del cuerpo del para (2 operaciones)

En total se realizan :

| | |
|---|---|
| Primera iteración : | 2 + 2 = 4 operaciones |
| Segunda iteración: | 3 + 2 = 5 operaciones |
| Tercera iteración: | 3 + 2 = 5 operaciones |
| …. | |
| N-esimaiteración: | 3 + 2 = 5 operaciones |
| Total paso 3 | 4 + 5(n-1) = 5n - 1 |

Complejidad temporal: 5n+3
Complejidad espacial: 4

se obtiene contabilizando el numero total de datos, variables temporales y de salida: n, s, i, p

## *Ejemplo 13*

Crear una fila secuencial con los N primeros números enteros positivos

La entrada es el entero positivo N, y la salida es la fila secuencial F que contiene los N primeros números enteros positivos

**<precondición** : N $\in Z^+$

**Postcondicion** : F contiene los primeros números enteros positivos

```
Accion CrearFila()                          
Inicio                                      
        PRIMER(F)                   1       
        Leer N                      2       
        Para i desde 1 hasta N      3       
                V = i               3.1     
                PONER(F,V)          3.2     
        FinPara                             
        MARCAR(F)                   4       
Fin                                         
```

| **Pasos** | **operaciones** |
|---|---|
| 1 | 1 |
| 2 | 1 |
| 3 | 6n - 1 |
| 4 | 1 |
| Total | 1+1+6n-1+1=6n+2 |

**Complejidad temporal : 6n-2**

**Complejidad espacial : n+3 (n, v, i y n elementos de la fila)**

## *Ejemplo 14 Hallar el producto interno de dos vectores*

```
Algoritmo PROD_INT(n, A, B, z);          Inicio
        Leer (n, A, B);
        z := 0;                          Proceso
        Para i := 1, n    hacer
        z := z + ai bi;                  Salida
        Escribir (z);
Fin.
```

Evaluar el tiempo de complejidad del siguiente algoritmo:

Evaluaremos por partes

**Inicio:** **2n + 2**

El bloque inicio está conformado por una instrucción de lectura y una de asignación. Siendo X e Y dos n – vectores,

entonces la instrucción de lectura va requerir de 2n + 1 operaciones.

Luego el bloque Inicio tiene complejidad exacta de 2n + 2, el cual esta conformado de una operación de asignación (z:= 0) y 2n+1 operaciones de la instrucción de lectura.

**Proceso** **6n - 1** (por propiedad de sumatorias)

El bloque de Proceso esta conformado por una instrucción *para*. Podemos calcular esta instrucción de la

$$\sum_{i=1}^{n}(3+3)-1$$

siguiente manera:

Donde el primer 3 indica el número de asignaciones que se realizan dentro del bucle y el otro 3, indica las operaciones propias del control del bucle.

**Salida**: **1**

En este bloque salida es realizado solamente una operación (Escribir(z)).

Así la complejidad en tiempo del algoritmo PROD_INT es:

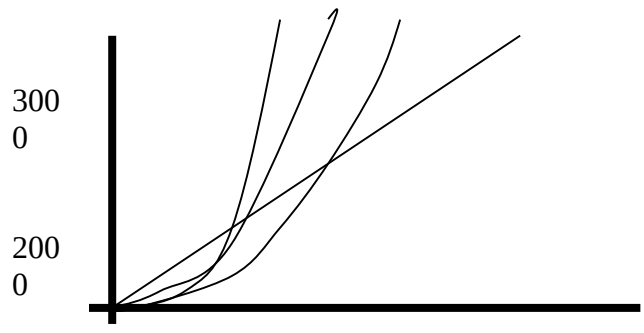
(2n + 2) + (6n –1) + 1 = **8n + 2**

## Algoritmos de tiempo polinomico y tiempo exponencial

Un algoritmo es llamado en tiempo polinomial si es de orden O(p(n)) para alguna función polinomial p(n), en caso contrario será superpolinomial. Esto es exponencial o peor.

Considérese las siguientes complejidades:
$T_1(n) = 2^n$
$T_2(n) = n^3$
$T_3(n) = 5n^2$
$T_4(n) = 100n$
Para cuatro algoritmos diferentes. Supongamos que cada operación es ejecutada en 1 segundo. Tenemos el siguiente comportamiento de las complejidades.

Obsérvese que aquellos que tienen menor complejidad presentan un menor crecimiento respecto a la variación del tamaño de la instancia (n), esto es, realizan menos operaciones y por consiguiente deberán ser más eficientes.



5 10 15 20

Un algoritmo es considerado bueno si posee complejidad polinomial, esto es debido a que conforme el tamaño de la entrada n crece, los tiempos de ejecución de los algoritmos de tiempo polinomico se mantienen

Para resolver un problema pueden existir varios algoritmos. Por tanto, es lógico elegir el ”mejor”. Si el problema es sencillo o no hay que resolver muchos casos se podría elegir el algoritmo mas “fácil”, sin embargo , si el problema es complejo o existen muchos casos habría que elegir el algoritmo que **menos recurso utilice.**
Los recursos más importantes son el tiempo de ejecución y el espacio de almacenamiento. Generalmente , el más importante es el tiempo.
Al hablar de eficiencia de un algoritmo nos referimos a lo “rápido” que se ejecuta.
La eficiencia de un algoritmo dependerá, en general, del tamaño de los valores de entrada N, en donde N puede significar el numero de elementos de un vector, el numero de nodos de una lusta enlazada, en numero de nodos de un arbol o un grafo etc.
Una operación elemental es aquella cuyo tiempo de ejecución tiene una cota superior constante que sólo depende de su implementación (por ejemplo: el ordenador o el lenguaje de programación utilizado.)

*Las operaciones elementales son consideradas de coste unitario.

*Ala hora de analizar un algoritmo importaran, por tanto, el numero de operaciones elementales que precisa.

*La decisión de determinar que una operación determinada es de coste unitario dependerá de los ejemplares del problema que la utilice. Por ejemplo: En la mayor parte de los casos la suma se considera de coste unitario puesto que al operar con los números que se manejan habitualmente en un ordenador los tiempos que emplea la suma son similares; sin embargo, en caso de manejar números muy grandes la suma no tendrá un coste unitario puesto que tardara más cuanto más sean los números a sumar .

* Explorar el comportamiento de una función ó de una relación entre funciones, cuando algún parámetro de la función tiende hacia un valor asintótico.

* Se denomina “asintótica” pues trata de funciones que tienden al límite.

* Las notaciones asintóticas nos permiten hallar la tasa de crecimiento del tiempo de ejecución

* Un algoritmo que sea superior asintóticamente es preferible; por que se considera el máximo límite que llega una función.

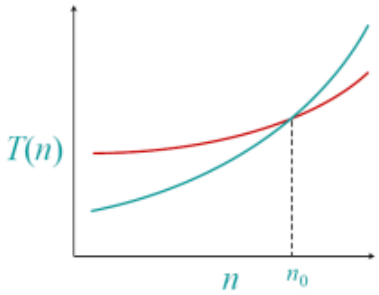
*Funciones Conocidas($\log_2 n$, n ,$n \log_2 n$, $n^2$, $n^3$, $2^n$)

| N | $\log_2 n$ | n | $n \log_2 n$ | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|
| 5 | 3 | 5 | 15 | 25 | 125 | 32 |
| 10 | 4 | 10 | 40 | 100 | 1000 | 1000 |
| 100 | 7 | 100 | 700 | 10000 | $10^6$ | $10^{30}$ |
| 1000 | 10 | 1000 | 10000 | $10^6$ | $10^9$ | $10^{300}$ |

## Notación O

Decimos que una función **T(n) es O(f(n))**
si existen constantes $n_0$ y c
tales que $T(n) \leq cf(n)$ para todo $n \geq n_0$:

**$T(n)$ es $O(f(n)) \Leftrightarrow$**
**$\exists c \in R, \exists n_0 \in N$, tal que $\forall n > n_0 \in N$, $T(n) \leq cf(n)$**




*La notación anterior básicamente nos dice que si tenemos un algoritmo cuyo tiempo de ejecución es T(n) podemos encontrar otra función f(n) , y un tamaño de problema **$n_0$** de tal forma que f(n) acota superiormente al tiempo de ejecución para todos los problemas de tamaño superior a $n_0$.

*Esta notación, como veremos a continuación, facilitara en las comparaciones de eficiencia entre algoritmos diferentes.

*Por ejemplo si $T(n)= n^2 + 5n + 100$ y $f(n) = n^2$ entonces f(n) domina a T(n).

En próximas lecciones se verá la forma de calcular complejidad de algoritmos. Sin embargo, podemos adelantar algunas reglas:

.O(C.g(n)) = O(g(n)) , C es una constante

. O(f(n).g(n)) = O(g(n))* O(f(n)) , y viceversa
.O(f(n)/g(n)) = O(g(n))/ O(f(n)) , y viceversa
. O(f(n)+g(n)) = función dominante de O(g(n)) y O(f(n))
.n* $\log_2 n$ domina a $\log_2 n$

### Dominancia entre funciones

Sea $\phi$ , $\varphi : N \rightarrow R^+$ funciones.
Se dice que $\phi$ domina asintoticamente a $\varphi$ si se verifica que
$\exists$ c, $n_0 > 0$ tal que

$\varphi(n) \leq c\,\phi(n) \quad \forall n \geq n_0$

obsérvese que a partir de $n_0$ la función $c\phi$ es siempre mayor que la función $\varphi$

## *Ejemplo 15*

**$n^3$ domina asintoticamente a $2n^2 + n$**
Para c =1 $n_0 = 3$ es fácil verificar que $2n^2 + n \leq n^3$ $\forall n \geq n_0$

**$n^3$ domina asintoticamente a $100n^2 \log n$**
Para c =10 $n_0$ = 40 es fácil verificar que **$100n^2 \log n \leq n^3$** $\forall\ n \geq n_0$

**$n^3$ domina asintoticamente a $3n^3 + 2n^2$**
Para c =5 $n_0$ = 0 es fácil verificar que **$3n^3 + 2n^2 \leq n^3$** $\forall\ n \geq n_0$

$T(n) = 3n$ ⇨ T(n) es O(n), O(n log n), $O(n^2)$, $O(n^3)$ y $O(2^n)$.

$T(n) = (n+1)^2$ ⇨ T(n) es $O(n^2)$, $O(n^3)$ y $O(2^n)$.
T(n) no es O(n) ni O(n log n).

$T(n) = 32n^2 + 17n + 32$ ⇨ T(n) es $O(n^2)$ pero no es O(n).

$T(n) = 3n^3 + 345n^2$ ⇨ T(n) es $O(n^3)$ pero no es $O(n^2)$.

$T(n) = 3^n$ ⇨ T(n) es $O(3^n)$ pero no es $O(2^n)$.

## Órdenes de eficiencia más habituales

| N | $O(\log_2 n)$ | $O(n^2)$ | $O(n \log_2 n)$ | $O(n^2)$ | $O(2^n)$ | O(n!) |
|---|---|---|---|---|---|---|
| 10 | 3 µs | 10 µs | 30 µs | 0.1 ms | 1 ms | 4 s |
| 25 | 5 µs | 25 µs | 0.1 ms | 0.6 ms | 33 s | $10^{11}$ años |
| 50 | 6 µs | 50 µs | 0.3 ms | 2.5 ms | 36 años | ... |
| 100 | 7 µs | 100 µs | 0.7 ms | 10 ms | $10^{17}$ años | ... |
| 1000 | 10 µs | 1 ms | 10 ms | 1 s | ... | ... |
| 10000 | 13 µs | 10 ms | 0.1 s | 100 s | ... | ... |
| 100000 | 17 µs | 100 ms | 1.7 s | 3 horas | ... | ... |
| 1000000 | 20 µs | 1 s | 20 s | 12 días | ... | ... |

Tiempos calculados suponiendo 1 µs por operación elemental.

**$O(1) \subset O(\log n) \subset O(n) \subset O(n \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n) \subset O(n!)$**

## *Ejemplo 16* *Evaluar el polinomio P*

```
Algoritmo POL_E(n, B,x)
Inicio
        Leer (B,n,x)
        p:=a0
Proceso
        Para i=1, n   hacer
                p:=p+aixi
Salida
        Escribir (p)
Fin.
```

**Inicio**: ***O(n)***

Este bloque está constituido por dos instrucciones independientes con complejidades:

Leer(.) n+2 → O(n)

*p*:= 1 → O(1)

Así por la regla 1, la complejidad asintótica del bloque inicio será:

$O(\text{máx}\{n,1\}) = \boldsymbol{O(n)}$

**Proceso**:$\boldsymbol{O(n^2)}$

Ciclo:

$p:=p+a_ix^i \rightarrow i+2 \rightarrow O(i)$

**Control:**

Para i:=1,n → $3n-1$ → $O(n)$

Usando la regla 2, $O(ni)$

Desde que *i* asume en el peor caso el valor de *n*, entonces la complejidad asintótica arriba será: $\boldsymbol{O(n^2)}$

**Salida:** $\boldsymbol{O(1)}$

Finalmente, la complejidad asintótica del algoritmo POL_E será:

$$O(max\{n,n^2,1\}) = O(n^2)$$

## *Ejemplo 17*

```
        Algoritmo POL_P()
        Leer (A,n,x)
        p:=a0,  y:=1

Proceso
        Para i=1, n   hacer
                y:=y*x , p:=p+aiy
        Fin-para;

Salida
```

**Inicio:** $\boldsymbol{O(n)}$

**Proceso:** $\boldsymbol{O(n)}$

Como en el ejemplo anterior, este está conformado por un ciclo de complejidad $O(1)$ que se repite un número de veces dado por un control de orden $O(n)$. Esto es, este bloque presenta complejidad asintótica de: $O(1n) = O(n)$

**Salida**: ***O*(1).**

La complejidad asintótica del algoritmo POL_P es: ***O(max{n,n,1}) = O(n)***

## Reglas sencillas

Los algoritmos estructurados combinan sentencias de tipo

- Secuencia
- Decisión binaria(if..then...else)
- Bucles (do..whhile, Repeat ...Until)
- Llamadas a procedimientos o funciones

## Sentencias sencillas

| | |
|---|---|
| Asignación | Id = Expresión |
| Entrada | Leer Lista_Id |
| Salida | Escribir Lista_Id |

Llamadas a funciones o procedimientos

Tienen complejidad O(1) , requieren tiempo de ejecución constante, siempre y cuando no trabajen sobre variables estructuradas cuyo tamaño esta relacionado con el tamaño del problema N

## Secuencias

La complejidad de una secuencia esta determinada por la suma de las complejidades individuales, aplicando las operaciones antes expuestas.

## Decisión binaria

La condición suele ser de orden O(1), a ello se le sumara la complejidad del caso peor posible, sea ya la rama then o la rama else. EN decisiones múltiples (ELSE IF, CASE Switch... ) se tomara el caso peor posible.

## Bucles

Se distinguen dos casos en los bucles con contador en el que el tamaño N forme parte de los limites o que no. Si el bucle se realiza un numero fijo de veces, independientes de N, entonces la repetición solo introducirá una constante multiplicativa que puede absolverse

## *Ejemplo 18*

```
S1
Para i desde 1 hasta N
        S2
FinPara
```

**T(n) = t1 + t2 * n**

**T1 : tiempo que lleva ejecutar la serie S1**
**T2 : tiempo que lleva ejecutar la serie S2**

## *Ejemplo 19*

**Si (Condicion )**
**S1**
**Else**
**S2**
**FinSi**
**Dependiendo de la condición, si es verdad se ejecuta S1, en caso contrario se ejecuta S2**
**Esto hace que exista más de un valor para T(n)**
**Tmin(n) ≤ T(N) ≤ Tmax(N)**
**Caso peor caso mejor**

**T(n) = t1 + t2 * n**
**T1 : tiempo que lleva ejecutar la serie S1**
**T2 : tiempo que lleva ejecutar la serie S2**

## *Ejemplo 20*

**Para i desde 1 hasta K**
**Algo de O(1)** **complejidad K* O(1) = O(1)**
**FinPara**

**Para i desde 1 hasta N**
**Algo de O(1)** **complejidad N* O(1) = O(N)**

**FinPara**

## *Ejemplo 21*

**Para i desde 1 hasta N**
**Para j desde 1 hasta N**
**Algo de O(1)** **complejidad N*N* O(1) = O($N^2$)**

*FinPara*
**FinPara**

**Para i desde 1 hasta N**
**Para j desde 1 hasta i**
**Algo de O(1)** **complejidad N*N* O(1) = O($N^2$)**

***FinPara***
**FinPara**

El bucle mayor se realiza N veces. Mientras que el bucle menor se realiza, 1, 2, ...n veces respectivamente. Por tanto se realiza 1+2+3+...n= N*(N+1)/2 **O($N^2$)**

# Ejercicios propuestos

**I Proporcione una especificación para**

1. Hallar el mayor y el menor elemento de un vector
2. Verificar si en un vector existe un elemento igual a la suma de los sucesores

3. Verificar si dos matrices tienen por lo menos una fila con los mismos elementos
4. Hallar un ordenamiento topológico de un grafo
5. Si la población de vicuñas es de 200, y en el año n habrá el 10% más de lo que hubo en el año anterior. Cuantas vicuñas habrá en el año N.
6. Considere las comunidades $C_1, C_2 \ldots C_n$, $D_{ij}$ es la distancia de la comunidad i a la comunidad j. $p_i$ es la población de la comunidad i. Se quiere construir un centro comunal de alfabetización para todas las comunidades, para ello deberá elegirse un lugar físico en la comunidad con menor costo de desplazamiento. Especifique e implemente una solución.
7. Hallar la solución de un sistema triangular inferior

**II** Impleméntese los algoritmos de la parte I (elija solo los pares o los impares)

a) Proporcione 2 ejemplos de algoritmos de orden lineal. Evalúe la complejidad temporal, complejidad espacial
b) Proporcione 2 ejemplos de algoritmos de orden cuadrático. Evalué la complejidad temporal, complejidad espacial
c) Proporcione 2 ejemplos de algoritmos de orden cúbico. Evalúe la complejidad temporal, complejidad espacial
d) Proporcione 2 ejemplos de algoritmos de orden exponencial.
e) Proporcione 2 ejemplos de algoritmos de orden logarítmico.
f) Proporcione 2 ejemplos de algoritmos de orden n-logarítmico.

## REFERENCIAS BIBLIOGRAFICAS


1) [AHO 1988], Alfred V.
"Estructura de Datos y algoritmos" Addison Wesley. 1988.
2) [ALMEIDA 2008], Francisco Almeida "Introducción a la programación paralela"
Paraninfo 2008 España ISBN 978-84-9732-674-2.

3) [ANDERSON 2008] Anderson James "Redes Neuronales" Edit AlfaOmega México
ISBN 978-970-15-1265-4

4) [BRASSARD 2001], G. / BRATLEY, T.
"Fundamentos de Algoritmia". Prentice Hall. 2001

5) [CORTEZ 2010] Cortez Vásquez Augusto. "Algoritmia". Edit EsVega Lima Perú
ISBN 978-612-00-0257-5

6) [CORTEZ 2012] Cortez Vásquez Augusto. "Algoritmia, Técnicas algorítmicas"
Edit San Marcos Lima Peru ISBN 978-612-00-0964-2

7) [LEE 2005] R.Lee. "Introducción al diseño y análisis de algoritmos"
Edit Mc Graw Hill Mexico ISBN 978-970-10-6124-4

8) [LEIJA 2009] Lorenzo Leija "Metodos de procesamiento avanzado"
Edit Reverte Mexico 209 ISBN 978-607-7815-01-3

9) [WEISS 2001] ALLEN WEISS, Mark
"Lenguaje de programación Java". Addison Wesley. Madrid 2001.

10) [WEISS 2003] ALEN WEISS, Mark
"Estructura de Datos en Java". Addison Wesley. 2003

1.