



**Universidad Nacional Autónoma de México**  
**Facultad de Ingeniería**  
**Estructuras de datos y algoritmos**

**Tema 2:**

**ANÁLISIS BÁSICO DE ALGORITMOS**

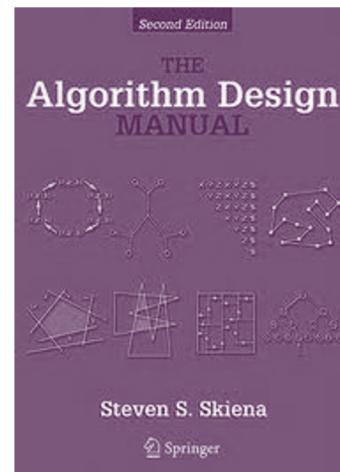
## 2 Análisis básico de algoritmos

**Objetivo:** Analizar algoritmos mediante medidas de rendimiento, espacio y tiempo.

## 2 Análisis básico de algoritmos

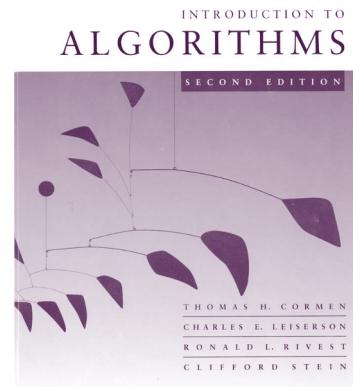
- 2.1 Fundamentos de algorítmica.**
- 2.2 Análisis asintótico de los límites superior y media.**
- 2.3 Notación O, omega y teta.**
- 2.4 Medidas empíricas de rendimiento.**
- 2.5 Compensación espacio y tiempo en los algoritmos.**

## Bibliografía



***The algorithm design manual.***  
**Steven S. Skiena, Springer.**

## Bibliografía



*Introduction to Algorithms.*

***Thomas H. Cormen, Charles E. Leiserson,  
Ronald L. Rivest, Clifford Stein, McGraw-Hill.***



## 2.1 Fundamentos de algorítmica.

## 2.1 Fundamentos de algorítmica.

**El diseño de algoritmos correctos, eficientes y aplicables a problemas cotidianos requiere el acceso a dos cuerpos distintos de conocimiento: las técnicas y los recursos.**

## Técnicas

**Un buen diseñador de algoritmos conoce diversas técnicas fundamentales del diseño de algoritmo, como lo son las estructuras de datos, la programación dinámica, la búsqueda profunda, la marcha atrás y las técnicas heurísticas.**

**Sin embargo, la técnica más importante en el diseño de algoritmos es el modelado, que es el arte de abstraer una aplicación confusa del mundo real en un problema adaptable para atacarlo por un algoritmo.**

## Recursos

**Un buen diseñador de algoritmos se apoya en los hombros de otros diseñadores. En lugar de volver a implementar algoritmos populares, buscan implementaciones existentes para usarlas como punto de partida.**

**Un diseñador de algoritmos está familiarizado con diversos problemas algorítmicos clásicos, lo que permite tener suficiente material inicial para modelar cualquier aplicación más robusta.**

**De manera formal se podría decir que un algoritmo es un método de solución para una instancia dada de un cierto problema, expresada en un lenguaje específico y que permite realizar una tarea en general, es decir, un conjunto de pasos, procedimientos o acciones que permiten alcanzar un resultado o resolver un problema.**

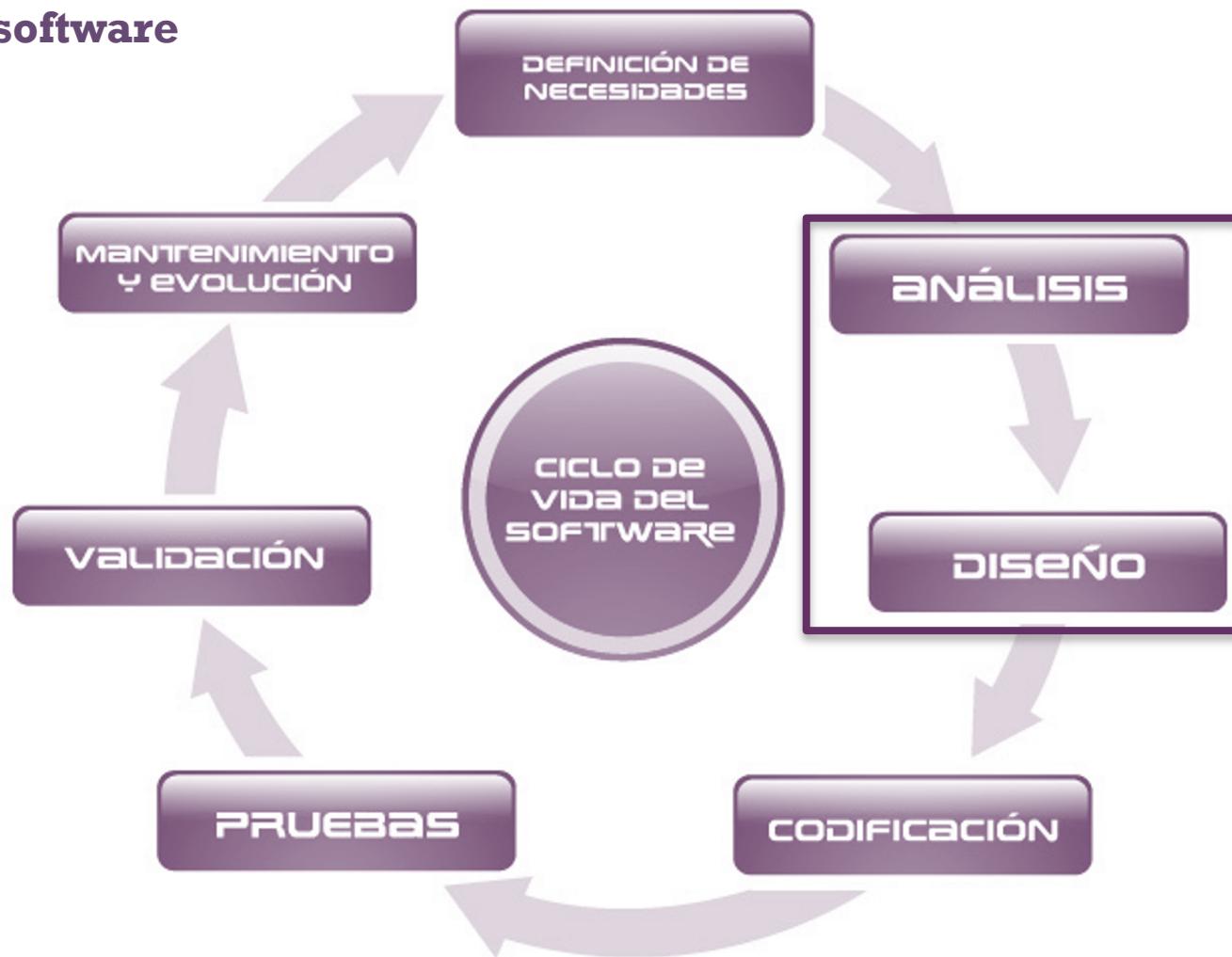
**Un algoritmo es la parte más importante y durable de las ciencias de la computación debido a que éste puede ser creado de manera independiente tanto del lenguaje como de las características físicas del equipo que lo va a ejecutar.**

**Lo que hace especial a un algoritmo es que sus reglas pueden ser aplicadas un número ilimitado de veces sobre una situación particular y siempre se obtiene un resultado consistente.**

**Las principales características con las que debe cumplir un algoritmo son:**

- **Preciso:** llegar a la solución en el menor tiempo posible y sin caer en ambigüedades.
- **Determinista:** a partir de un conjunto de datos idénticos de entrada, debe arrojar siempre los mismos resultados a la salida.
- **Finito:** Un proceso computable implica que, en algún momento, éste va a finalizar. Un algoritmo terminar en algún momento.

## Ingeniería de software





How the customer explained it



How the Project Leader understood it



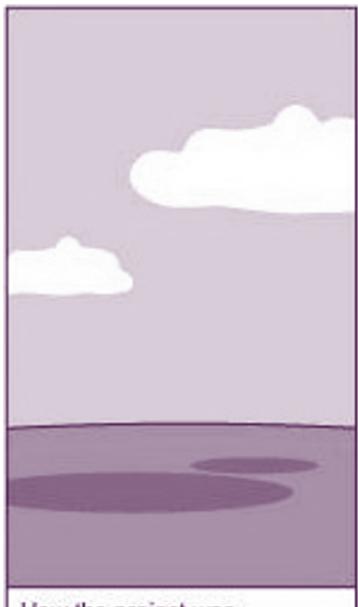
How the Analyst designed it



How the Programmer wrote it



How the Business Consultant described it



How the project was documented



What operations installed



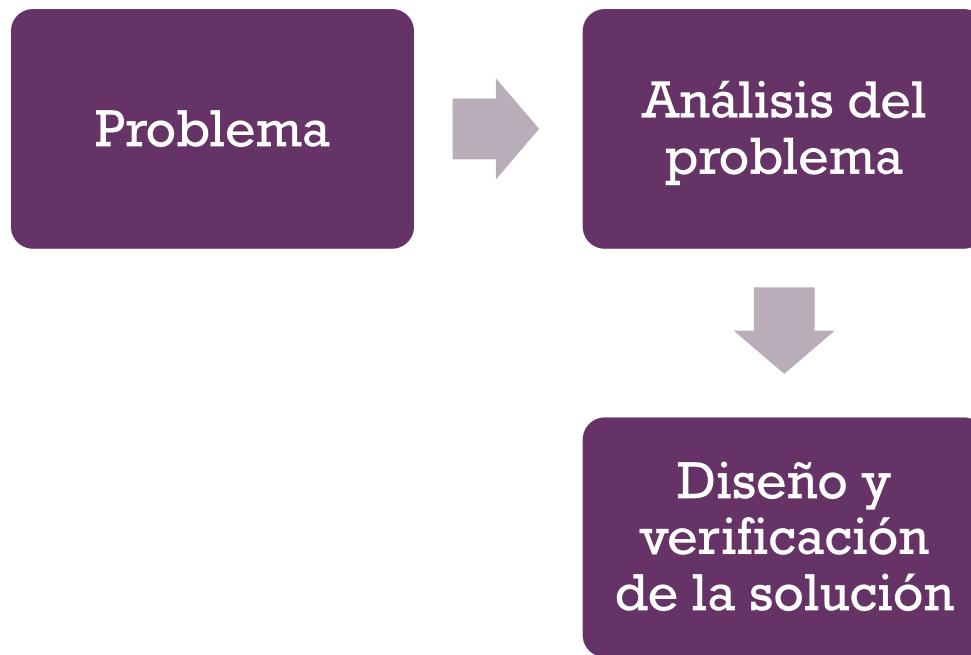
How the customer was billed



How it was supported

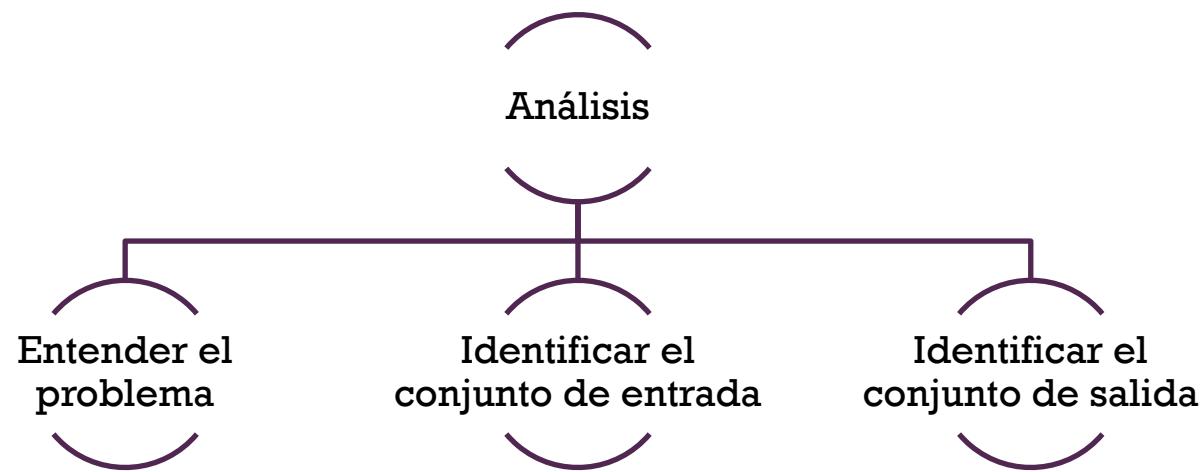


What the customer really needed

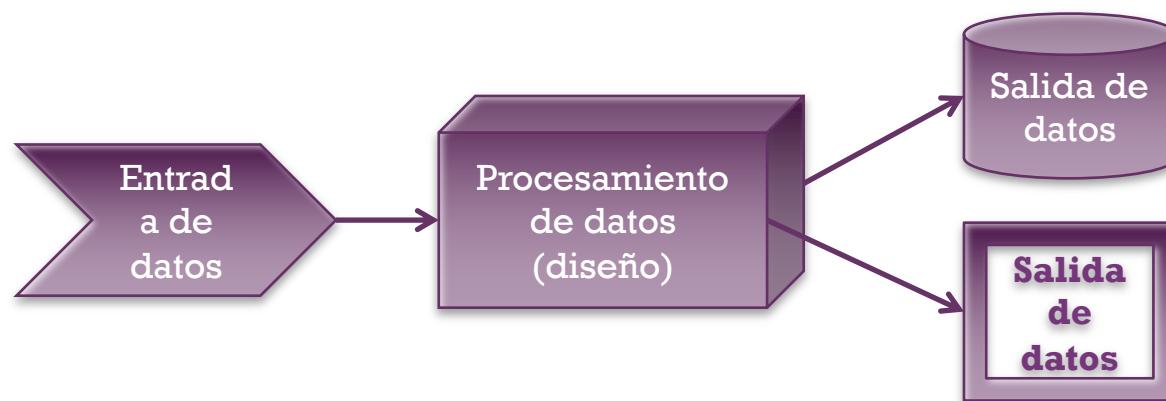


## Problema

De manera matemática, un problema se define como un conjunto de instancias al cual corresponde un conjunto de soluciones, junto con una relación que asocia para cada instancia del problema un subconjunto de soluciones (posiblemente vacío).



El análisis permite identificar el módulo de entrada, el módulo de salida y el módulo de procesamiento,





Entrada  
de datos

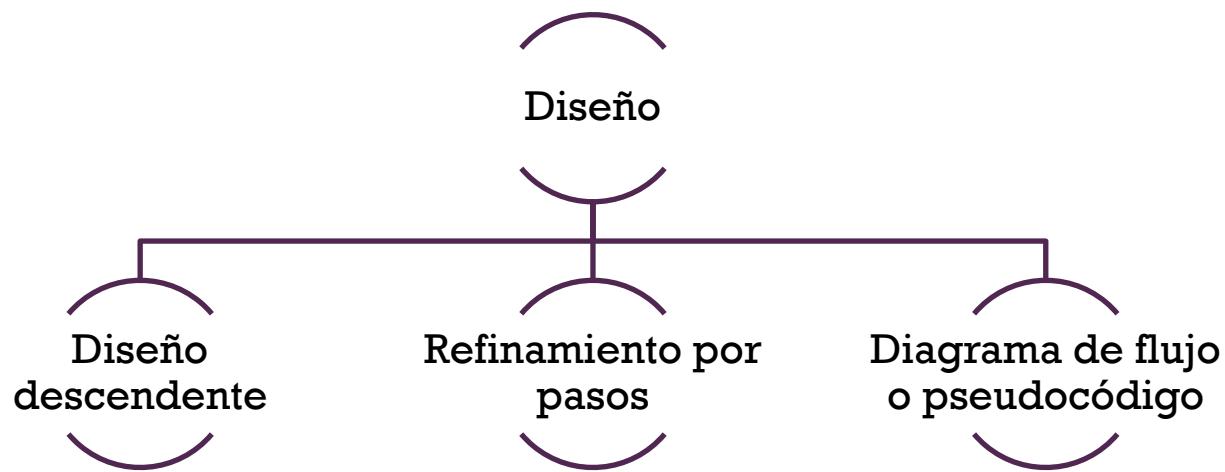
**Un algoritmo se define por un conjunto  $E$  de entradas que representan las instancias del problema, es decir, el módulo de entrada de datos representa los tipos de datos que alimentan al problema.**



**El módulo de procesamiento representa las operaciones necesarias para obtener un resultado a partir de los datos de entrada (diseño de la solución).**



**Un algoritmo posee un conjunto  $S$  de salidas que son los posibles resultados que emanen de la ejecución del algoritmo, es decir, el módulo de salida permite definir los tipos y valores que se deseean obtener al final del proceso.**



**El diseño descendente divide el problema en problemas más sencillos de tal manera que el conjunto de soluciones forme la solución general.**

**El refinamiento por pasos permite comprobar cada uno de los módulos en los que se dividió el problema en el diseño descendente.**

**La solución del problema se debe representar en un lenguaje común y fácil de entender, siendo el más utilizado el pseudocódigo.**

## Algoritmo para crear un algoritmo

```
PROGRAMA desarrollar_algoritmo ( problema )
    correcto: BOOLEAN;
    correcto := FALSO;
    MIENTRAS (NO correcto) O (NO rápido(tiempo_ejec))
        algoritmo = idear_algoritmo(problema)
        correcto = analizar_exactitud(algoritmo)
        tiempo_ejec= analizar_eficiencia(algoritmo)
    FIN_MIENTRAS
    RETURN algoritmo
FIN_PROGRAMA
```

**El análisis de algoritmos se ha convertido en una manera de predecir los recursos que un proceso va a consumir: memoria, ancho de banda, arquitectura de la computadora, etc.**

**Empero, generalmente lo que se quiere medir es el tiempo computacional. Para ello se deben analizar diversos algoritmos que resuelvan el problema, para identificar el más eficiente.**

**En el proceso de selección se puede tener más de un algoritmo que realice el trabajo correctamente y, a su vez, permite filtrar los que no lo son.**

**PROBLEMA: Determinar si un número dado es par o impar.****SOLUCIÓN:**

1. **INICIO.**
2. **X,M: ENTERO**
3. **X := 0.**
4. **HACER**
5.       **ESCRIBIR “Ingrese número entero.”**
6.       **LEER X.**
7. **MIENTRAS X = 0**
8.   **M := X%2**
9.   **SI M = 0.**
10.      **ESCRIBIR “X es par”**
11. **DE LO CONTRARIO**
12.      **ESCRIBIR “X es impar”.**
11. **FIN.**

## Prueba de escritorio

iteración	X	M	Imprime
0	5	1	X es impar

iteración	X	M	Imprime
0	4	0	X es par

iteración	X	M	Imprime
0	-3	1	X es impar

iteración	X	M	Imprime
0	-10	0	X es par

## Prueba de escritorio

iteración	X	M	Imprime
0	0	-	-
1	0	-	-
2	13	1	X es impar

**Realizar un programa que multiplique dos números (equis y ye) sin utilizar el operador multiplicación (\*), es decir, no se puede especificar en el algoritmo la operación:**

equis \* ye

**La solución se debe proporcionar en pseudocódigo. Al final se tiene que verificar el algoritmo (pruebas de escritorio).**

1. **INICIO**
2. **X, Y, Z: ENTERO**
3. **HACER Z := 0**
4. **ESCRIBIR “Ingrese un número entero.”**
5. **LEER X**
6. **ESCRIBIR “Ingrese otro número entero.”**
7. **LEER Y**
8. **MIENTRAS X > 0**
  - 9.     **HACER Z = Z + Y**
  - 10.    **HACER X = X - 1**
11. **FIN\_MIENTRAS**
12. **ESCRIBIR “El resultado es: “ Z**
13. **FIN**

## Prueba de escritorio

iteración	X	Y	Z
0	5	7	7
1	4	7	14
2	3	7	21
3	2	7	28
4	1	7	35
5	0	7	35

**El algoritmo anterior cumple con el objetivo del problema planteado, empero se puede eficientar el tiempo de ejecución.**

**Debido a la propiedad de la multiplicación “el orden de los factores no altera el producto”, una vez que ya se hayan recibido ambos valores (X y Y), se comprueba cuál es el menor y se le asigna a X. Debido a que X determina cuantas veces se va a realizar el ciclo, entre más pequeña sea X menos iteraciones se realizarán.**

1. **INICIO**
2. **X, Y, Z, TMP: ENTERO**
3. **HACER Z = 0**
4. **ESCRIBIR “Ingrese un número entero.”**
5. **LEER X**
6. **ESCRIBIR “Ingrese otro número entero.”**
7. **LEER Y**
8. **Si Y < X**
  9.     **HACER TMP = X**
  10.    **HACER X = Y**
  11.    **HACER Y = TMP**
12. **TERMINA\_SI**
13. **MIENTRAS X > 0**
  14.    **HACER Z = Z + Y**
  15.    **HACER X = X - 1**
16. **FIN\_MIENTRAS**
17. **ESCRIBIR “El resultado es: “ Z**
18. **FIN**

**El diseño de un algoritmo puede llegar a ser tan exquisito o refinado como se deseé o experiencia se tenga.**

**Existe un algoritmo más eficiente para resolver una multiplicación en menos pasos. Para entender su funcionamiento es importante comprender las operaciones que realizan los operandos  $>>$  y  $<<$ , esto es, corrimientos de bits tanto a la izquierda como a la derecha.**

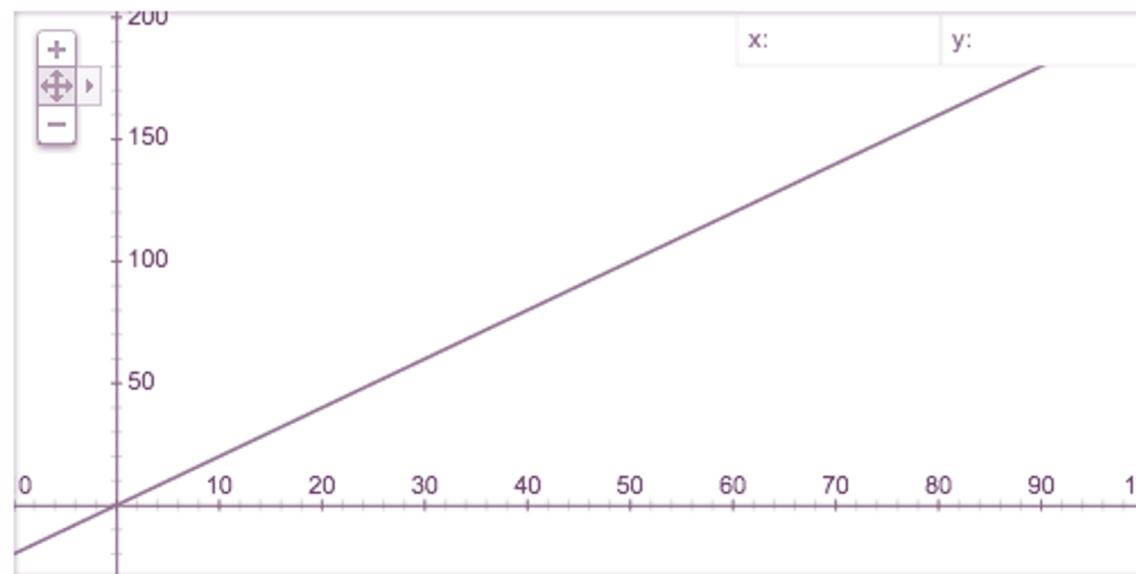
- 1. INICIO**
- 2. X, Y y Z: ENTERO**
- 3. HACER Z = 0**
- 4. ESCRIBIR “Ingrese un número entero.”**
- 5. LEER X.**
- 6. ESCRIBIR “Ingrese otro número entero.”**
- 7. LEER Y.**
- 8. MIENTRAS X > 0**
  - 9. Si (X % 2) = 1**  
**10.                  HACER Z = Z + Y**
  - 11.                  HACER Y << 1**
  - 12.                  HACER X >> 1**
- 11. FIN\_MIENTRAS**
- 12. ESCRIBIR “El resultado de la operación es: “ Z**
- 13. FIN**

## Prueba de escritorio

iteración	X	Y	Z
0	14	11	0
1	7	22	0
2	3	44	22
3	1	88	66
4	0	176	154

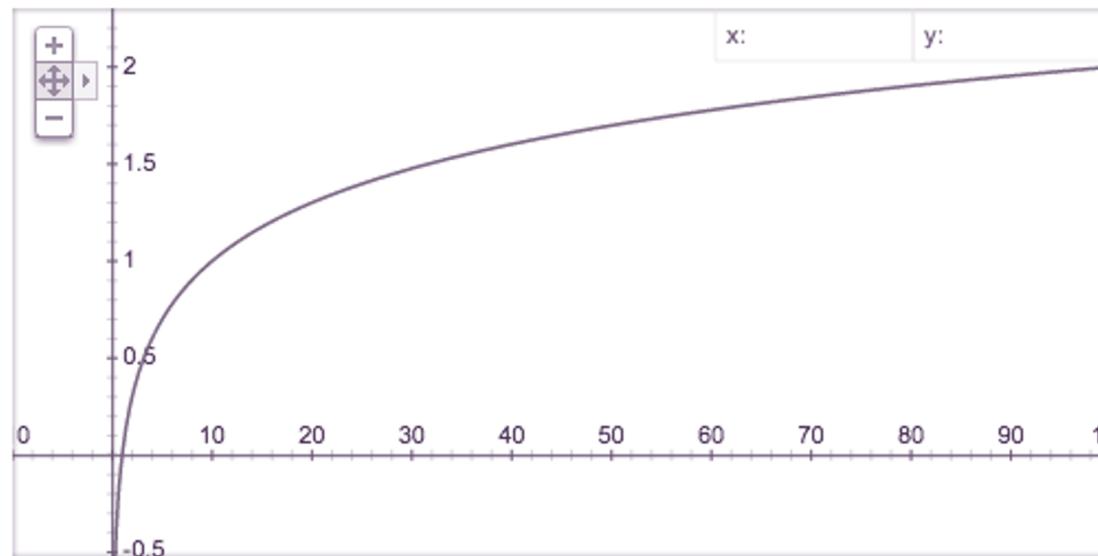
**El tiempo que le toma al algoritmo de las primeras dos soluciones en realizar la multiplicación es lineal:**

$$t = 2 * x$$



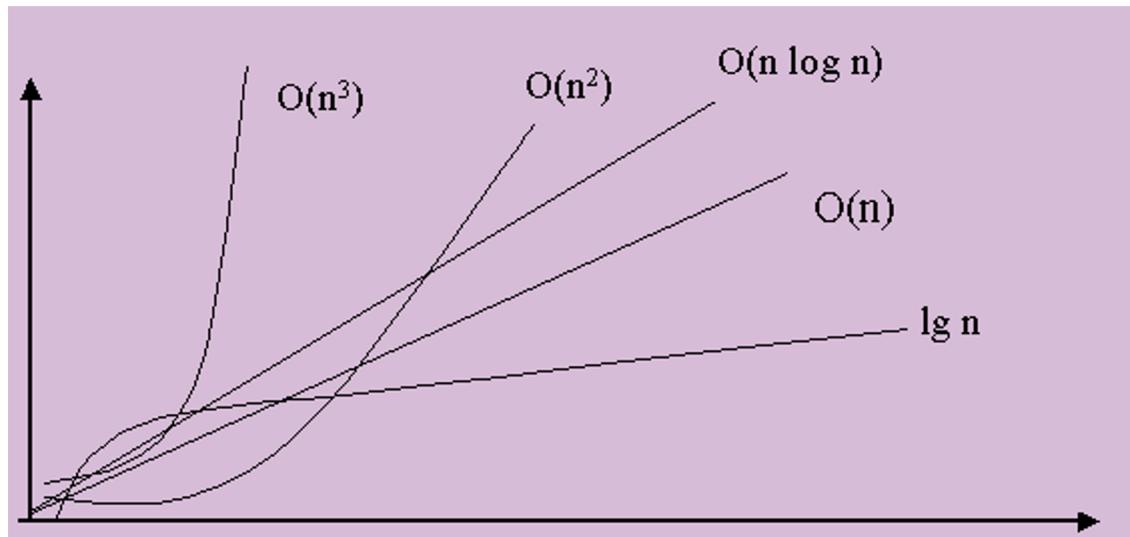
**El tiempo que le toma al algoritmo de la solución mejorada en realizar la multiplicación es logarítmico:**

$$\lceil \log_2 x \rceil + 1$$



**Por lo tanto, a pesar de que los tres algoritmos cumplen con la solución del problema planteado, el último es más eficiente.**

**Al este último método se le conoce como algoritmo de los campesinos rusos (Russian peasant algorithm) para resolver una multiplicación.**



## 2.3 Notación O, omega y teta

## 2.3 Notación O, omega y teta

**Las dos medidas más importantes para establecer la calidad de un algoritmo son:**

- **El tiempo total de ejecución, medido por el número de operaciones de cómputo realizadas durante el proceso.**
- **La cantidad de memoria utilizada por dicho proceso.**

**Tanto la eficiencia en la ejecución como la cantidad de memoria utilizada durante un proceso son parámetros que permiten medir la complejidad de un algoritmo.**

**Las técnicas más utilizadas para comparar la eficiencia de algoritmos sin necesidad de implementarlos son el modelo de computación RAM (Random Access Machine) y el análisis asintótico del peor caso de complejidad.**

**El modelo RAM se refiere a la representación de una computadora hipotética que permite evaluar la eficiencia del diseño de un algoritmo de manera independiente de la arquitectura (hardware) donde se implemente.**

**El modelo RAM asume las siguientes reglas:**

- **Cada operación simple (+, -, \*, /, selección o llamada) toma exactamente un paso de tiempo.**
- **Los ciclos están compuestos por operaciones simples, por lo tanto, el tiempo que toma un ciclo depende del número de iteraciones del mismo.**
- **El acceso a memoria toma exactamente un paso de tiempo. La memoria de un modelo RAM es infinita.**

**Un modelo RAM permite medir el tiempo que consume la ejecución de un algoritmo, contando el número de pasos que contiene el mismo. Si se asume que un modelo RAM ejecuta un cierto número de pasos por segundo, en automático se puede obtener el tiempo de ejecución de un proceso.**

**La RAM es un modelo simple que intenta representar el funcionamiento de un equipo, y, por tanto, hay que tener ciertas consideraciones.**

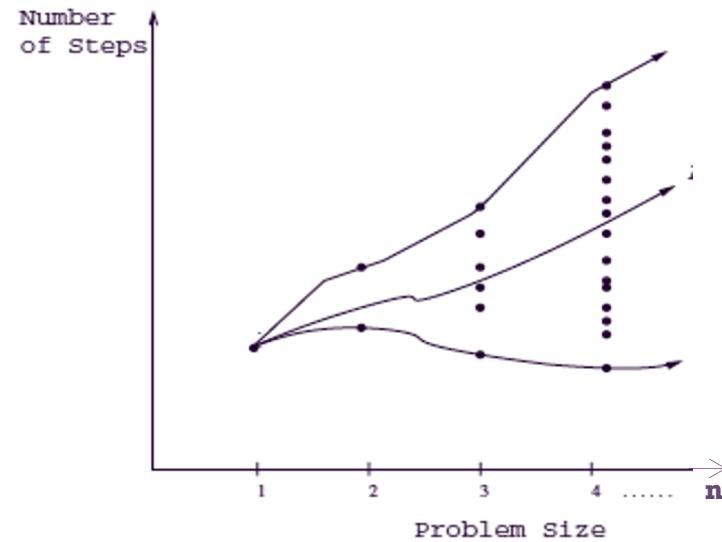
**En la mayoría de los procesadores, multiplicar dos números toma más tiempo que sumarlos. Esto viola la primera regla del modelo.**

**Además, los procesos paralelos o multihilos así como ciertos compiladores pueden violar la segunda regla.**

**Por si fuera poco, el acceso a la memoria difiere en tiempo dependiendo del lugar donde se encuentren los datos, con lo que se viola la tercera regla.**

**A pesar de los bemoles mencionados, el modelo RAM provee una excelente aproximación para entender cual sería el desempeño de un algoritmo en una computadora real. Es por lo anterior que es tan útil en la práctica.**

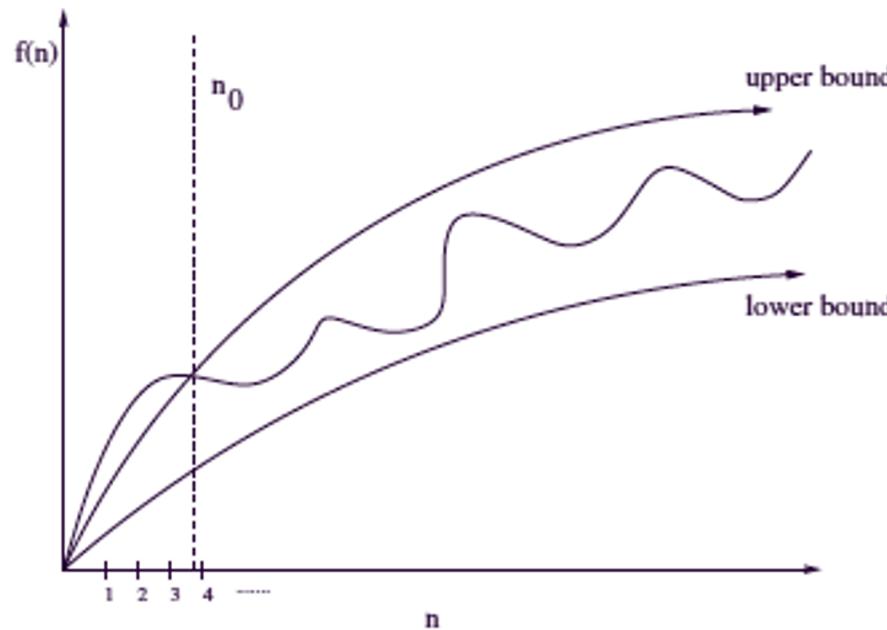
**El tiempo de complejidad para un algoritmo dado es una función numérica que depende del tamaño de las instancias dadas (valores o datos de entrada).**



**Sin embargo, trabajar con las tres funciones o, más específicamente, con los casos extremos de las instancias para un algoritmo es bastante difícil debido a dos razones básicas:**

- **Las funciones poseen muchos saltos: instancias nones pueden requerir más tiempo que las pares, por ejemplo.**
- **Las funciones requieren mucho detalles para ser precisas:  $t(n) = 12754n^2 + 4353n + 13546$**

Por tanto, es mejor trabajar con funciones de tiempo de complejidad con límites superior e inferior. Esto se logra utilizando la notación O.



**La notación O simplifica el análisis, ignorando niveles de detalle que no impacta en la comparación de algoritmos.**

**La notación O no hace diferencia entre multiplicación de constantes, es decir, la función  $f(n) = 2n$  y  $g(n) = n$  son idénticas en el análisis O.**

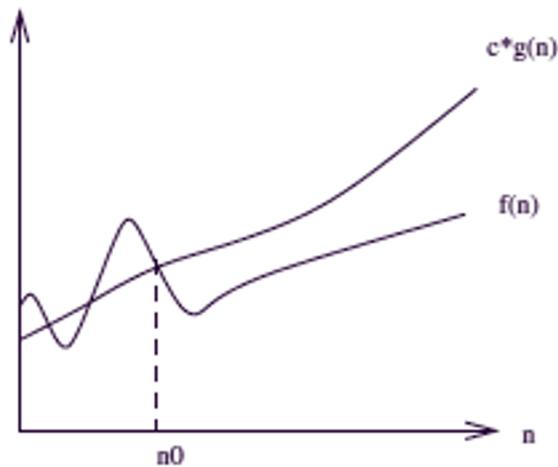
Las definiciones formales de la notación O son las siguientes:

- $f(n) = O(g(n))$  significa que  $c*g(n)$  es el límite superior en  $f(n)$ . Por tanto, existe una constante  $c$  tal que  $f(n)$  es siempre menor o igual a  $c*g(n)$  por muy grande que  $n$  sea, para  $n > n_0$ .
- $f(n) = \Omega(g(n))$  significa que  $c*g(n)$  es el límite inferior en  $f(n)$ . Por tanto, existe una constante  $c$  tal que  $f(n)$  es siempre mayor o igual a  $c*g(n)$  para  $n > n_0$ .

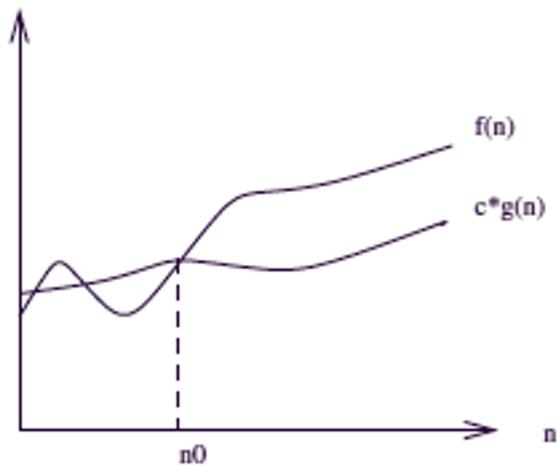
- $f(n) = \Theta(g(n))$  significa que  $c_1 \cdot g(n)$  límite superior en  $f(n)$  y  $c_2 \cdot g(n)$  es un límite inferior en  $f(n)$  para  $n > n_0$ . Por tanto, existen un par de constantes  $c_1$  y  $c_2$  tales que:

$$c_2 \cdot g(n) \leq f(n) \leq c_1 \cdot g(n)$$

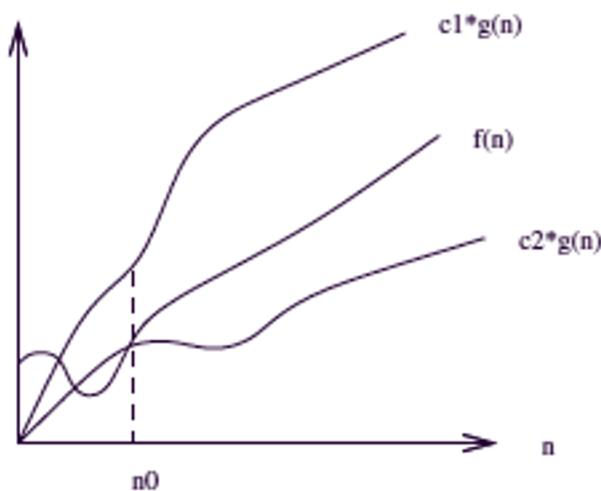
Lo que significa que  $g(n)$  provee una muy estrecha y buena aproximación de  $f(n)$ .



$$f(n) = O(g(n))$$



$$f(n) = \Omega(g(n))$$



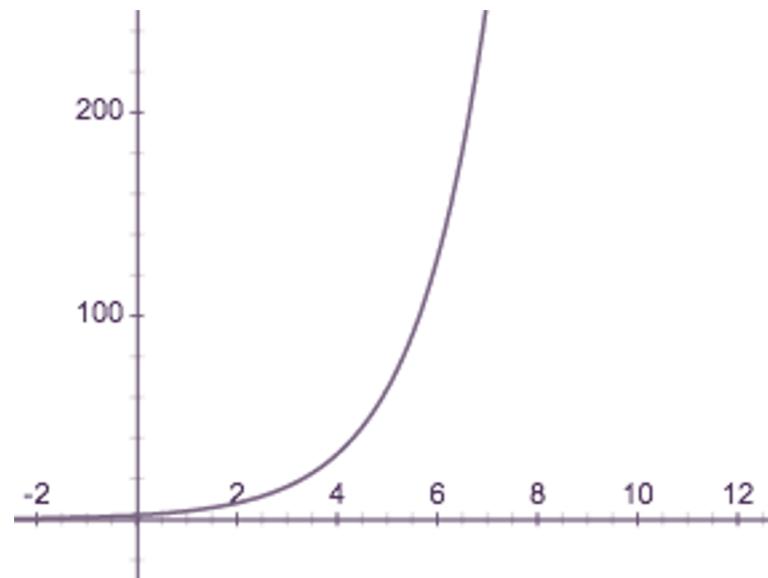
$$f(n) = \Theta(g(n))$$

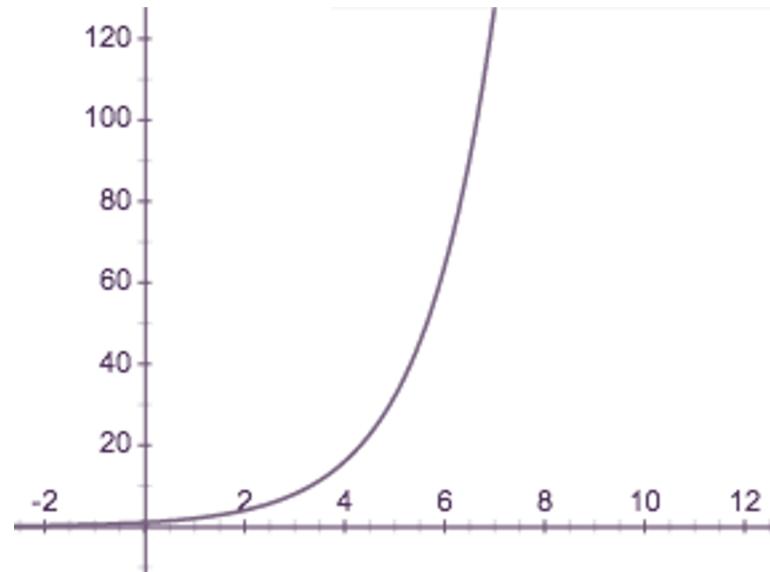
**¿Se cumple la siguiente aseveración?:**

$$2^{n+1} = \Theta(2^n)$$

**NOTA.** En este caso, el significado del símbolo “=” se puede leer como “es una función de”.

$$2^{n+1}$$



$2^n$ 

¿ $2^{n+1} = O(2^n)$ ?

Por definición,  $f(n) = O(g(n))$  si y solo si existe una constante  $c$  tal que para una valor de  $n$  muy grande, se cumpla la condición  $f(n) \leq c*g(n)$ .

Observando la gráfica se tiene que  $2^{n+1} = 2*2^n$ , por lo tanto, existe una función  $c*2^n$ , que cumple con la función  $f(n) = 2^{n+1}$  con  $c \geq 2$ .

¿ $2^{n+1} = \Omega(2^n)$ ?

Por definición,  $f(n) = \Omega(g(n))$  si y solo si existe una constante  $c > 0$  tal que para una valor de  $n$  muy grande, se cumpla la condición  $f(n) \geq c*g(n)$ .

Para la función  $2^{n+1}$  la condición se satisface para los valores  $0 < c \leq 2$ .

Juntos los límites superior ( $O$ ) e inferior ( $\Omega$ ), permiten validar que  $2^{n+1} = \Theta(2^n)$ .

**Con la notación O (big oh) se pueden descartar los multiplicandos constantes dentro de una función.**

Así, las funciones  $f(n) = 0.001n^2$  y  $g(n) = 1000n^2$  son tratadas de manera idéntica, a pesar de que  $g(n)$  es un millón de veces mayor que  $f(n)$  para cualquier valor de  $n$ .

**A continuación se muestran las tasas de crecimiento en tiempo de las funciones más comunes, en una computadora donde cada operación toma un nano segundo  $10^{-9}$  [s]:**

$n$	$f(n)$	$\lg n$	$n$	$n \lg n$	$n^2$	$2^n$	$n!$
10	0.003 $\mu$ s	0.01 $\mu$ s	0.033 $\mu$ s	0.1 $\mu$ s	1 $\mu$ s	3.63 ms	
20	0.004 $\mu$ s	0.02 $\mu$ s	0.086 $\mu$ s	0.4 $\mu$ s	1 ms	77.1 years	
30	0.005 $\mu$ s	0.03 $\mu$ s	0.147 $\mu$ s	0.9 $\mu$ s	1 sec		$8.4 \times 10^{15}$ yrs
40	0.005 $\mu$ s	0.04 $\mu$ s	0.213 $\mu$ s	1.6 $\mu$ s	18.3 min		
50	0.006 $\mu$ s	0.05 $\mu$ s	0.282 $\mu$ s	2.5 $\mu$ s	13 days		
100	0.007 $\mu$ s	0.1 $\mu$ s	0.644 $\mu$ s	10 $\mu$ s	$4 \times 10^{13}$ yrs		
1,000	0.010 $\mu$ s	1.00 $\mu$ s	9.966 $\mu$ s	1 ms			
10,000	0.013 $\mu$ s	10 $\mu$ s	130 $\mu$ s	100 ms			
100,000	0.017 $\mu$ s	0.10 ms	1.67 ms	10 sec			
1,000,000	0.020 $\mu$ s	1 ms	19.93 ms	16.7 min			
10,000,000	0.023 $\mu$ s	0.01 sec	0.23 sec	1.16 days			
100,000,000	0.027 $\mu$ s	0.10 sec	2.66 sec	115.7 days			
1,000,000,000	0.030 $\mu$ s	1 sec	29.90 sec	31.7 years			

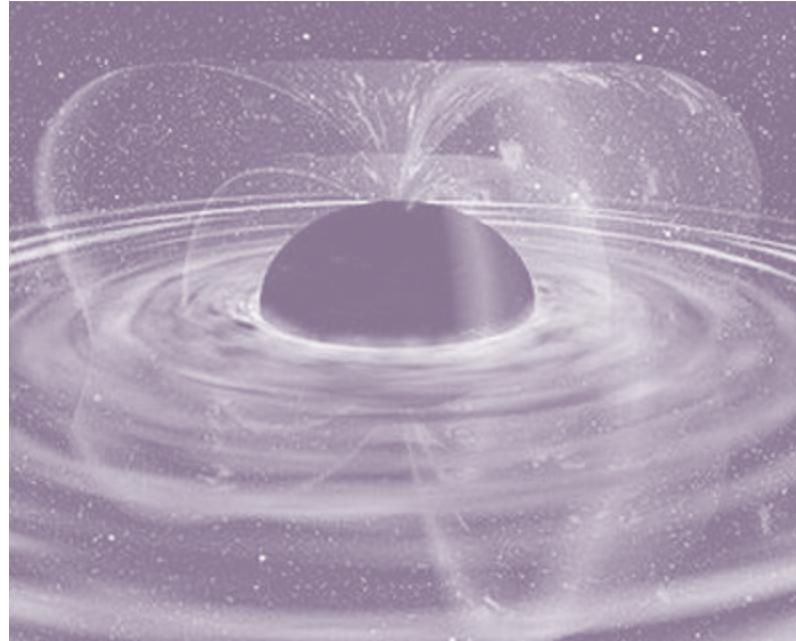
**A partir de la tabla anterior se pueden obtener las siguientes conclusiones:**

- **Todos los algoritmos toman aproximadamente el mismo tiempo para  $n = 10$ .**
- **El tiempo para cualquier algoritmo que ejecute  $n!$  se vuelve inoperante para  $n \geq 20$ .**
- **Para algoritmos con tiempo de ejecución  $2^n$  tienen un buen rango operativo, pero se vuelven imprácticos para  $n > 40$ .**

- **Algoritmos cuyos tiempos de ejecución sean  $n^2$  son rápidos mientras que  $n < 10,000$ , pero el tiempo se deteriora con rapidez con instancias más largas.**
- **Algoritmos lineales ( $n$ ) o de la forma  $n * \log(n)$  son prácticos con instancias hasta de 1 billón.**
- **Un algoritmo de la forma  $O(\log(n))$ , mantienen un tiempo de ejecución bajo para cualquier valor de  $n$ .**

Como se puede observar, a pesar de ignorar los factores constantes, se obtiene una idea muy aproximada de si un algoritmo es apropiado con base en el tamaño de los elementos de entrada.

Sin embargo, un algoritmo cuyo tiempo de ejecución esté dado por  $f(n) = n^3$  será más rápido que un algoritmo cuyo tiempo de ejecución sea  $g(n) = 1,000,000 n^2$  cuando la instancia de entrada  $n < 1,000,000$ .



## 2.2 Análisis asintótico de los límites superior y media.

## 2.2 Análisis asintótico de los límites superior y media.

**La tasa de crecimiento del tiempo de ejecución de un algoritmo permite obtener una aproximación de la eficiencia del mismo, así como comparar el comportamiento con otros algoritmos.**

**Cuando se comparan valores de entrada suficientemente grandes se hace relevante el orden de crecimiento del tiempo de ejecución y, por tanto, se está estudiando la eficiencia asintótica del algoritmo.**

**El objetivo es analizar el incremento en el tiempo de ejecución de un algoritmo con respecto al tamaño de la entrada en los límites.**

**Usualmente, un algoritmo asintóticamente eficiente será la mejor opción para cualquier conjunto de entradas.**

**Los límites asintóticos están definidos por la función  $\Theta(g(n))$ , es decir:**

$$\Theta(g(n)) = f(n)$$

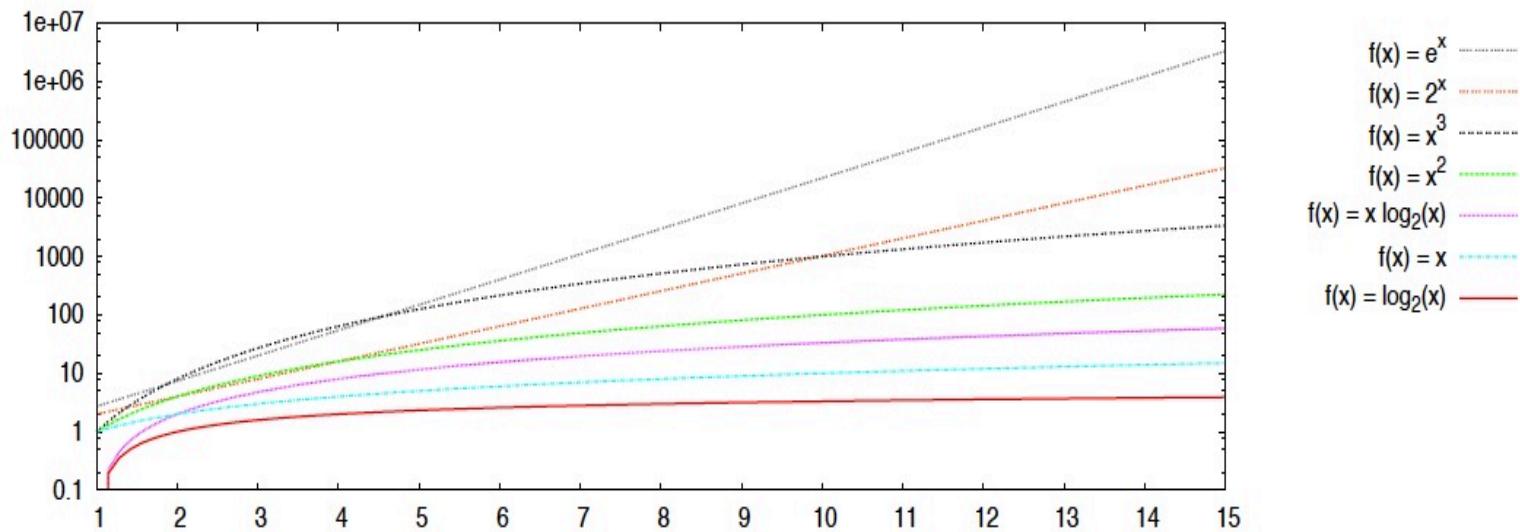
**para la cual existen tres constantes positivas ( $c_1$ ,  $c_2$  y  $n_0$ ) tal que**

$$0 < c_2 * g(n) < f(n) < c_1 * g(n)$$

**para cualquier  $n > n_0$ .**

Dado que  $c_1 \cdot g(n) = O(g(n))$  y  $c_2 \cdot g(n) = \Omega(g(n))$ , se puede afirmar que:

- **$O(g(n))$  es una cota asintótica superior.**
- **$\Omega(g(n))$  es una cota asintótica inferior.**
- **$\Theta(g(n))$  es un conjunto al que pertenece  $f(n)$ .**
- **$g(n)$  es un límite asintótico estrecho.**



**Teorema:** Para cualesquiera dos funciones  $f(n)$  y  $g(n)$ , se tiene que  $f(n) = \Theta(g(n))$ , si y sólo si  $f(n) = O(g(n))$ , así como  $f(n) = \Omega(g(n))$ .

Por lo tanto, la igualdad  $an^2 + bn + c = \Theta(n^2)$  para cualquier valor de las constantes  $a$ ,  $b$  y  $c$ , con  $a > 0$ , permite afirmar que  $an^2 + bn + c = \Omega(n^2)$  y  $an^2 + bn + c = O(n^2)$ .

**En la práctica, el teorema anterior se utiliza para probar lo estrecho de los límites asintóticos de las cotas superior e inferior.**

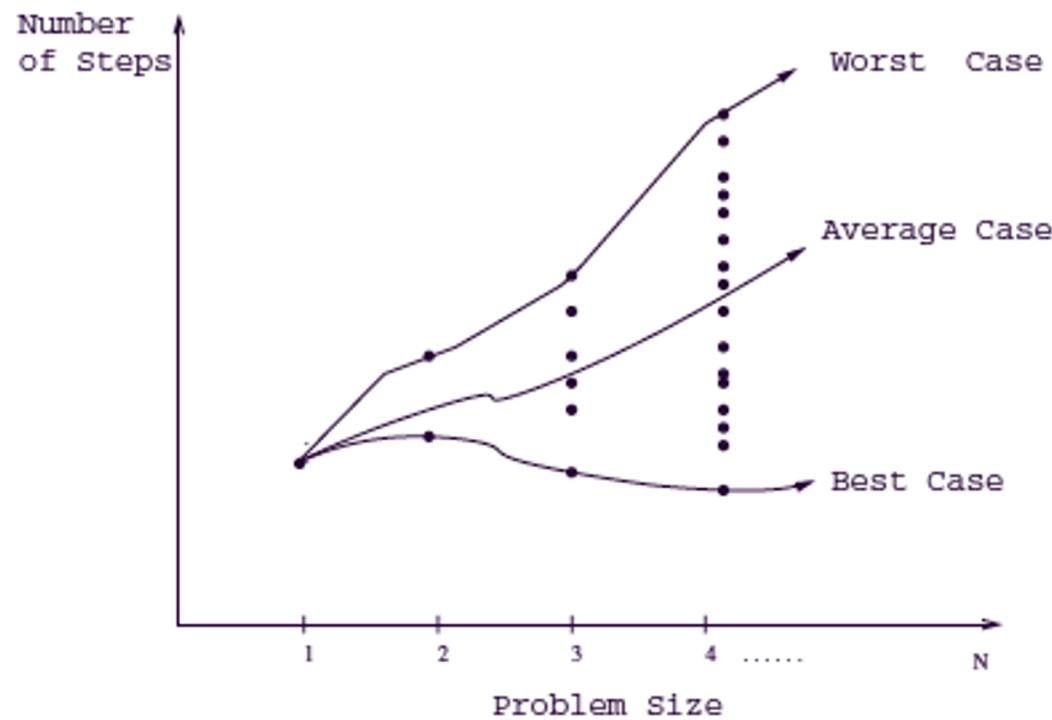
**Debido a que la notación  $\Omega$  describe una cota inferior, cuando se usa  $\Omega$  para indicar el límite del mejor caso en tiempo de ejecución de un algoritmo, también se indica el tiempo de ejecución del algoritmo para cualquier entrada arbitraria.**

**Utilizando el modelo computacional RAM, es posible contar cuantos pasos toma un algoritmo en realizar una tarea con base en una instancia de entrada (datos de entrada).**

**Sin embargo, para ponderar que tan eficiente o ineficiente es un algoritmo, es necesario probar todas las instancias de entrada posibles.**

**Para entender el concepto de el mejor, el peor y el caso promedio de complejidad es necesario que, para un algoritmo dado, se ejecuten todas las posibles instancias de datos que puedan alimentarlo.**

**Por ejemplo, para un algoritmo de ordenamiento el conjunto de instancias de entrada consta de todas las posibles combinaciones de  $k$  elementos de todas las posibilidades de  $n$ .**



**El eje x representa el tamaño de la entrada del problema. El eje y representa el número de pasos que toma al algoritmo en cada instancia.**

A partir de la gráfica anterior se pueden observar tres funciones diferentes.

1. El peor caso de complejidad de un algoritmo es una función definida por el máximo número de pasos para cada instancia de tamaño  $n$ . Esta curva representa el punto más alto para cada columna (dato de entrada).

2. El mejor caso de complejidad de un algoritmo es la función definida a partir del menor número de pasos para cada instancia de tamaño  $n$ . Esta curva representa el punto más bajo para cada columna (dato de entrada).
3. El caso promedio de complejidad de un algoritmo es una función definida por un número promedio de pasos de cada intancia de tamaño  $n$ .

**Así como las pruebas de un algoritmo pueden revelar errores de diseño, el cronometrar el tiempo de ejecución de un algoritmo puede revelar anomalías en el comportamiento del mismo en tiempo de ejecución para entradas específicas.**

A partir de una función es posible estimar los recursos computacionales que necesita un equipo para llevar a cabo su rutina.

```
FUNC cuantos (lista:REAL) DEV num:ENTERO
    SI (lista=Vacía)
        return 0;
    FIN_SI
    EN_CASO CONTRARIO
        DEV cuantos(dism lista) + 1;
    FIN_CASO CONTRARIO
FIN_FUNC
```

Si, para la función anterior, se ingresa una lista de 3 elementos, el resultado debe ser 3:

```
lista ← 'a', 'b', 'c'
```

Por tanto, si se aplica la función ‘cuantos’ a una lista más larga, se van a necesitar más pasos de recursión y viceversa.

**El tiempo de ejecución de un algoritmo está dado por el número de operaciones primitivas o pasos a ejecutar. Un paso está definido por una cantidad constante de tiempo requerido para ejecutar cada línea del pseudocódigo.**

**Cada línea se ejecuta en una cantidad de tiempo diferente, pero se debe asumir que la ejecución de la iésima línea toma un tiempo  $c_i$ , donde  $c_i$  es una constante.**

**A continuación se presenta el algoritmo de ordenamiento para un conjunto de elementos dados.**

**Cada línea genera un costo en tiempo de ejecución. Para cada  $j = 2, 3, \dots, n$  donde  $n =$  número de elementos,  $t_j$  es el número de veces que se ejecuta la condición del ciclo MIENTRAS, para cada valor de  $j$ .**

**Cuando existen ciclos del tipo MIENTRAS, la condición lógica se evalúa una vez más de las veces que se ejecuta el cuerpo del ciclo.**

## ORDENAMIENTO(A)

```

1 for j ← 2 to length[A]
2     key ← A[ j ]
3     i ← j - 1
4     while i > 0 and A[i] > key
5         A[i + 1] ← A[i ]
6         i ← i - 1
7     A[i + 1] ← key

```

## costo      iteraciones

<b>c<sub>1</sub></b>	<b>n</b>
<b>c<sub>2</sub></b>	<b>n - 1</b>
<b>c<sub>3</sub></b>	<b>n - 1</b>
<b>c<sub>4</sub></b>	$\sum_{j=2}^n t_j$
<b>c<sub>5</sub></b>	$\sum_{j=2}^n (t_j - 1)$
<b>c<sub>6</sub></b>	$\sum_{j=2}^n (t_j - 1)$
<b>c<sub>7</sub></b>	<b>n - 1</b>

**El tiempo de ejecución de un algoritmo está dado por la suma de los tiempos de ejecución para cada línea de código.**

**Cada línea toma  $c_i$  pasos para ejecutarse y es ejecutada  $n$  veces, por tanto, el tiempo total de ejecución es  $c_i n$ .**

**Para computar el tiempo de ordenamiento  $T(n)$  se suman los productos de costo y veces de cada línea:**

$$\begin{aligned} T(n) = & c_1 n + c_2(n - 1) + c_3(n - 1) + c_4 \sum_{j=2}^n t_j + \\ & + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n - 1) \end{aligned}$$

**Hay que recordar que el tiempo de ejecución de un algoritmo depende del tamaño de la entrada del mismo.**

Por ejemplo, para el algoritmo de ordenamiento el mejor caso ocurre cuando los elementos se dan de manera ordenada. Para cualquier valor de  $j=2,3,\dots,n$ , la condición del ciclo MIENTRAS siempre será  $A[i] \leq key$ . Por tanto,  $t_j = 1$  para cualquier valor de  $j$  y, por tanto, su tiempo de ejecución es:

$$T(n) = c_1n + c_2(n - 1) + c_3(n - 1) + c_4(n - 1) + c_7(n - 1)$$

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

Por tanto, el tiempo de ejecución del programa se puede expresar como  $an + b$ , es decir, una función lineal.

**Por otro lado, si el arreglo se da en forma decreciente, entonces se presenta el peor caso y, por tanto, se deben comparar todos los elementos  $A[j]$  con cada elemento del subarreglo. En este caso  $t_j = j$ , para  $j=2,3,\dots,n$ , las sumatorias se expresan de la siguiente manera:**

$$\sum_{j=2}^n t_j = \frac{n(n+1)}{2} - 1$$

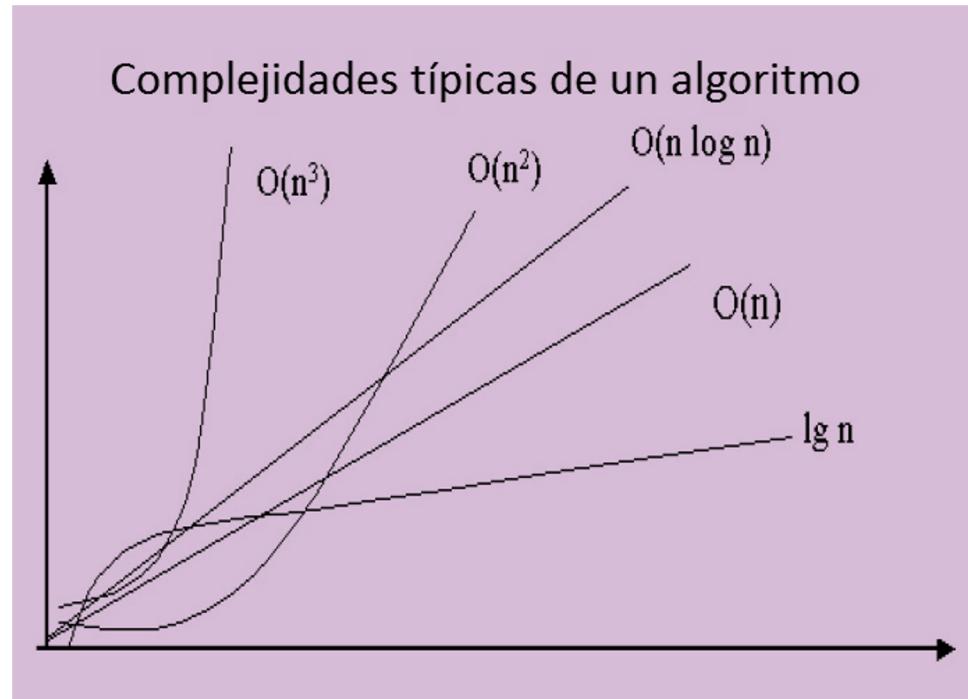
$$\sum_{j=2}^n (t_j - 1) = \frac{n(n-1)}{2}$$

Por lo tanto, para el peor caso, el tiempo de ejecución del algoritmo de ordenamiento es:

$$\begin{aligned} T(n) = & c_1n + c_2(n - 1) + c_3(n - 1) + c_4 (((n*(n+1))/2)-1) + \\ & + c_5 ((n*(n-1))/2) + c_6 ((n*(n-1))/2) + c_7(n - 1) \end{aligned}$$

$$\begin{aligned} T(n) = & (c_4/2 + c_5/2 + c_6/2)*n^2 + \\ & + (c_1+c_2+c_3+c_4/2+c_5/2+c_6/2+c_7)*n + \\ & + (c_2+c_3+c_4+c_7) \end{aligned}$$

De aquí se obtiene que el tiempo de ejecución del peor caso se puede expresar como  $an^2 + bn + c$ , es decir, una función cuadrática.



## 2.4 Medidas empíricas de rendimiento.

## 2.4 Medidas empíricas de rendimiento.

**Un algoritmo se puede analizar de manera teórica a través de su complejidad ya sea utilizando el análisis asintótico de la función o la notación O.**

**También es posible analizar la complejidad de un algoritmo de manera empírica, viendo su comportamiento a lo largo del tiempo para un cierto número de instancias dadas.**

**Las medidas empíricas de rendimiento permiten analizar un algoritmo de manera práctica, es decir, utilizando la gráfica que tiene el algoritmo para determinar tanto los límites asintóticos de la función como las funciones  $\Theta(g(n))$  que rodean a la misma.**

## Algoritmo para generar un diccionario para contraseñas de 4 dígitos.

```
1 FUNC crearDico (abecedario[]: CARACTER)
2   MIENTRAS d1< 27 HACER
3     MIENTRAS d2< 27 HACER
4       MIENTRAS d3< 27 HACER
5         MIENTRAS d4< 27 HACER
6           ESCRIBIR d1d2d3d4
7           FIN MIENTRAS
8       FIN MIENTRAS
9   FIN MIENTRAS
10 FIN MIENTRAS
11 FIN
```

**Algoritmo para convertir un número entero decimal a binario.**

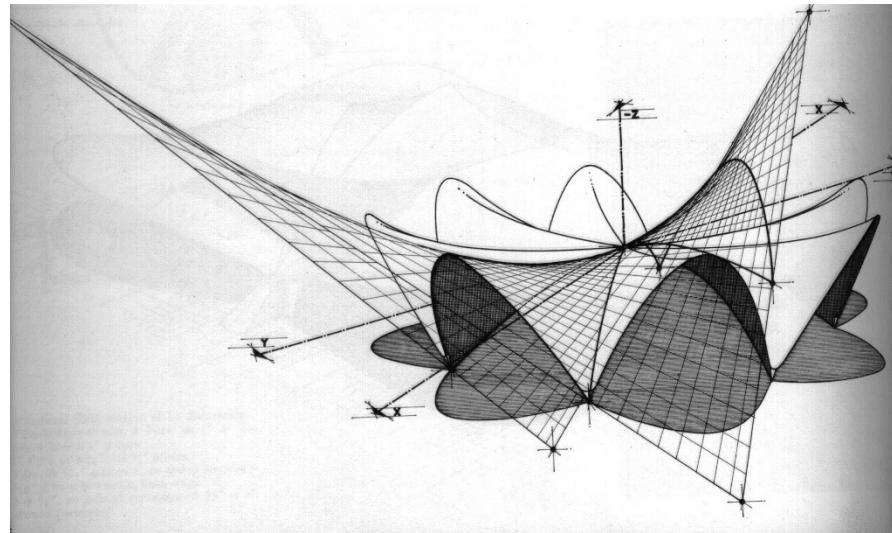
```
1      FUNC dec2bin (x: Entero) DEV vacío
2          SI x > 0 HACER
3              dec2bin (x/2);
4              ESCRIBIR x%2
5          FIN_SI
6      FIN_FUNC
```

## Algoritmo para calcular el enésimo número de la serie de Fibonacci.

```
1 FUNC fibonacci(ene: ENTERO) DEV ENTERO
2     a ← 0
3     b ← 1
4     contador ← 1
5     fibo ← 0
6     MIENTRAS contador < ene HACER
7         fibo ← b + a
8         a ← b
9         b ← fibo
10        contador ← contador + 1
11    DEV fibo
12 FIN_FUNC
```

## Algoritmo para obtener el factorial de un número.

```
1      FUNC factorial(ene):
2          SI ene = 0 ENTONCES
3              return 1
4          SI ene == 1 ENTONCES
5              return 1
6          DEV ene * (factorial(ene - 1))
7      FIN_FUNC
```



## 2.5 Compensación espacio y tiempo en los algoritmos.

## 2.5 Compensación espacio y tiempo en los algoritmos.

**La eficiencia de un programa se puede plantear como un compromiso entre el tiempo y el espacio utilizados. Por eso, la etapa de diseño es tan importante dentro del proceso de construcción de algoritmos ya que va a determinar, en muchos aspectos, la calidad del software generado.**

**Por tanto, el uso eficiente de los recursos suele medirse en función del espacio (determinado por la cantidad de memoria que utiliza) y el tiempo (representado por el tiempo que tarda un algoritmo en ejecutarse).**

**El espacio y el tiempo representan el costo de la solución al problema planteado mediante un algoritmo. Dichos parámetros sirven para comparar varios algoritmos entre sí y determinar con ello el más adecuado.**

**El mismo concepto que se utiliza para medir la complejidad del tiempo de ejecución de un algoritmo se utiliza para medir su complejidad en espacio.**

**Por tanto, un programa tiene una complejidad en espacio  $\Theta(n)$  significa que sus requerimientos de memoria aumentan de manera proporcional con el tamaño del problema.**

**Para un programa de complejidad en espacio  $\Theta(n^2)$ , la cantidad de memoria que se necesita para almacenar los datos crece con el cuadrado del tamaño del problema: si el problema se duplica, se requiere cuatro veces más memoria.**

**Al aumentar el espacio utilizado para almacenar la información se puede conseguir un mejor desempeño. Por tanto, entre más sencillas sean las estructuras de datos más lentos resultan los algoritmos.**

**Las estructuras de datos llevan implícitas ciertas limitaciones de eficiencia en sus operaciones básicas. Por eso la etapa de diseño es tan importante dentro del proceso de construcción de software ya que ahí se determinan muchos aspectos de la calidad del algoritmo.**

**Como ya se ha mencionado, el análisis de algoritmos se ha convertido en un sinónimo de predicción. Un buen análisis permite predecir los recursos que un algoritmo va a ocupar en tiempo de ejecución.**

**Determinar la complejidad temporal y/o espacial de un algoritmo puede ayudar a predecir de forma confiable el tiempo de ejecución y el espacio en memoria que se requiere para diferentes instancias sin necesidad de implementar el algoritmo.**

**Es por eso que se ocupa el modelo de computación RAM para ponderar los recursos utilizados durante el proceso y, por tanto, este modelo es muy utilizado para predecir el rendimiento de un equipo real.**

**Una algoritmo eficaz no necesariamente es un algoritmo eficiente, por tanto, cuando un algoritmo logra cumplir su objetivo (resolver el problema) es necesario ponderar su rendimiento y/o comportamiento.**

**Los criterios para medir su eficiencia se centran en su simplicidad y en el uso eficiente de los recursos (modelo RAM).**

**El uso eficiente de los recursos suele medirse en función del espacio utilizado y del tiempo de ejecución.**

**Por lo tanto, el tiempo de ejecución de las instancias dadas (datos de entrada), el código generado por el compilador (código obj), la velocidad del procesador y, por supuesto, la complejidad del algoritmo.**

**El siguiente algoritmo permite buscar un número c dentro de un conjunto de datos a:**

```
0 FUNC Buscar(a[]:ENTERO, c:ENTERO) DEV ENTERO
1     j←1:ENTERO;
2     MIENTRAS (a[j]<c) AND (j<n) HACER
3         j←j+1
4     FIN_MIENTRAS
5     SI a[j]=c ENTONCES
6         DEV j
7     FIN_SI
8     DE_LO_CONTRARIO
9         DEV 0
10    FIN_DLC
11 FIN_FUNC;
```

**Para contabilizar la eficiencia del algoritmo es preciso calcular el número de operaciones elementales que éste realiza:**

- **La línea 1,  $j:=1$ , ejecuta dos operaciones elementales (declaración y asignación).**
- **La línea 2, MIENTRAS ( $a[j] < c$ ) AND ( $j < n$ ) HACER, efectúa 4 operaciones elementales (dos comparaciones, un acceso al vector y una operación lógica).**
- **La línea 3,  $j:=j+1$ , ejecuta un incremento y una asignación (2 operaciones elementales).**

**Para contabilizar la eficiencia del algoritmo es preciso calcular el número de operaciones elementales que éste realiza:**

- La línea 5, SI  $a[j]=c$  ENTONCES, valida una condición y un acceso al vector (2 operaciones elementales).
- La línea 6, DEV  $j$ , ejecuta la operación elemental si la condición se cumple.
- La línea 9 DEV 0, ejecuta la operación elemental si la condición no se cumple.

**El siguiente algoritmo permite buscar un número c dentro de un conjunto de datos a:**

	Costo
0 <b>FUNC Buscar(a[]:ENTERO, c:ENTERO) DEV ENTERO</b>	
1       j←1:ENTERO;	2
2       MIENTRAS (a[j]<c) AND (j<n) HACER	4
3           j←j+1	2
4       FIN_MIENTRAS	
5       SI a[j]=c ENTONCES	2
6           DEV j	1
7       FIN_SI	
8       DE_LO CONTRARIO	
9           DEV 0	1
10      FIN_DLC	
11 FIN_FUNC;	

Dado el siguiente código, contabilizar el tiempo que se tarda el algoritmo siguiente en ejecutarse (las veces que se ejecuta cada sentencia):

```
FUNC doSomething() ENTONCES
    i ← 0 : ENTERO
    MIENTRAS i < 5 ENTONCES
        ESCRIBIR i
        i ← i + 1
    FIN MIENTRAS
    ESCRIBIR i
FIN FUNC
```

**Para el ejemplo anterior, el número de veces que se ejecuta el ciclo es fijo (5 en este caso). Sin embargo, cuando el número de iteraciones está en función de una variable, el análisis del algoritmo utilizando los puntos que asume el modelo RAM permite generar el polinomio que describe su tiempo de ejecución.**

Dado el siguiente algoritmo, obtener el polinomio que describe su comportamiento en función de la variable x.

```
FUNC countdown(x):VACIO  
    y ← 0: ENTERO  
    MIENTRAS x > 0 ENTONCES  
        x ← x - 5  
        y ← y + 1  
    FIN MIENTRAS  
    ESCRIBIR y  
FIN FUNCIÓN
```

## 2 Análisis básico de algoritmos

**Objetivo:** Analizar algoritmos mediante medidas de rendimiento, espacio y tiempo.

- 2.1 Fundamentos de algorítmica.**
- 2.2 Análisis asintótico de los límites superior y media.**
- 2.3 Notación O, omega y teta.**
- 2.4 Medidas empíricas de rendimiento.**
- 2.5 Compensación espacio y tiempo en los algoritmos.**