

Universidad Autónoma Gabriel Rene Moreno

FICCT

Semestre I/2013

►► Análisis de Algoritmos (I)



Justificación

Para desarrollar software, además de conocer la metodología con que se va a afrontar dicho problema, es también importante conocer, de manera específica y clara las funciones que puede realizar el computador y las formas en que se pueden manipular los elementos o dispositivos que lo conforman, de tal manera que se haga ***un óptimo uso de los recursos.***

Palabras Clave

- **Algoritmo:** Conjunto de pasos o instrucciones descritas en un lenguaje sencillo que permiten llegar a la solución sistemática de un problema.
- **Análisis de un Algoritmo:** Observar el comportamiento de un algoritmo y predecir la cantidad de recursos (tiempo, memoria, CPU) que un algoritmo requerirá para cualquier entrada.
- **Complejidad Algorítmica:** Se le denomina a la medida de la eficiencia de un programa.
- **Lenguaje Algorítmico:** Son una serie de símbolos y reglas que se utilizan para describir de manera explícita un proceso.

Contenido

1

Introducción

2

Análisis del Algoritmo

3

Notación Asintótica

4

Ejercicios

1. Introducción

Algoritmos



```
graph TD; A([Algoritmos]) --> B[Historia]; A --> C[Definición];
```

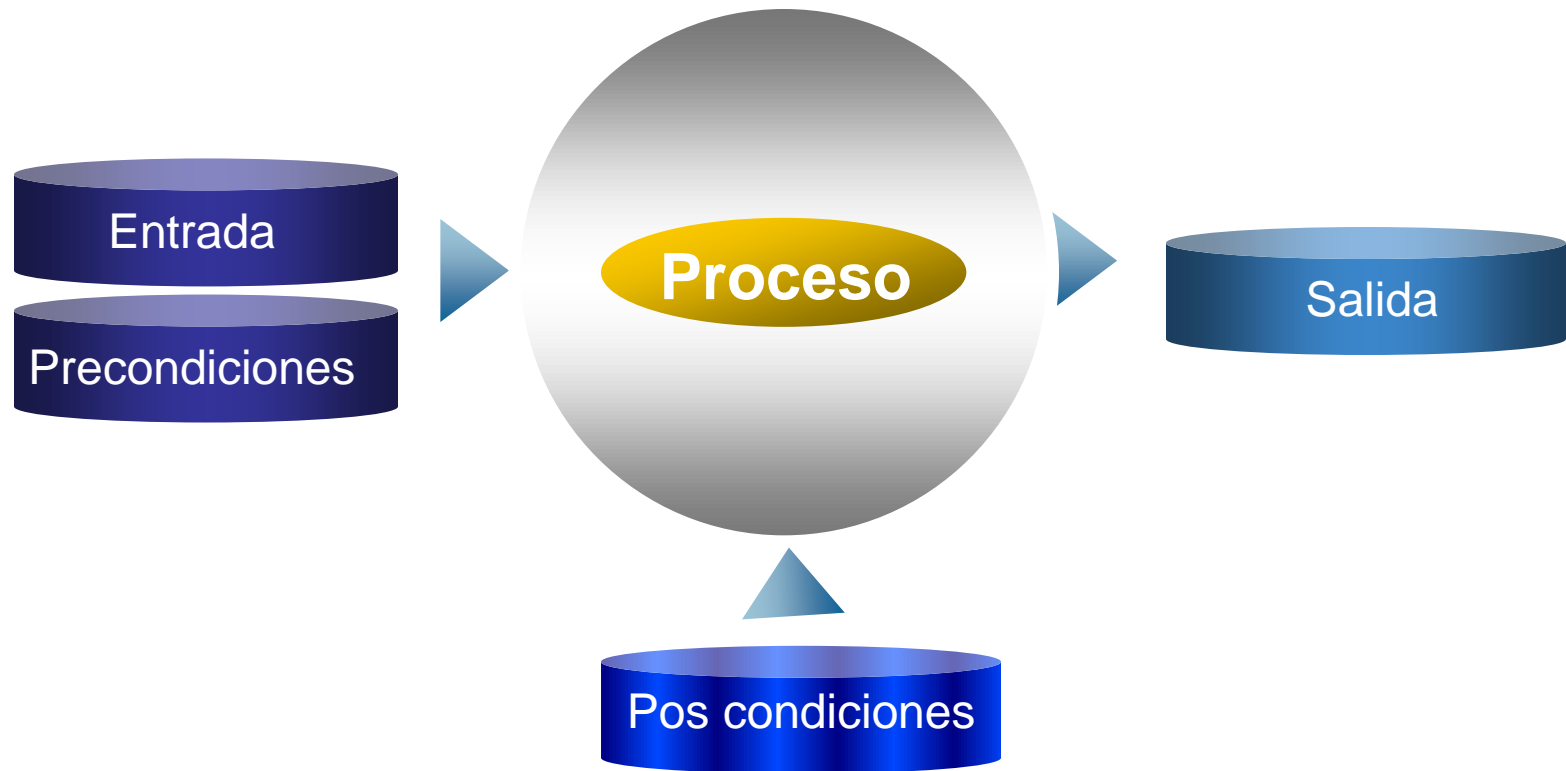
Historia

La definición de “algoritmo” data del siglo IX se atribuye su invención al matemático **Abu Ja’far Muhammad ibn Musa al-Khwarizmi**.

Definición

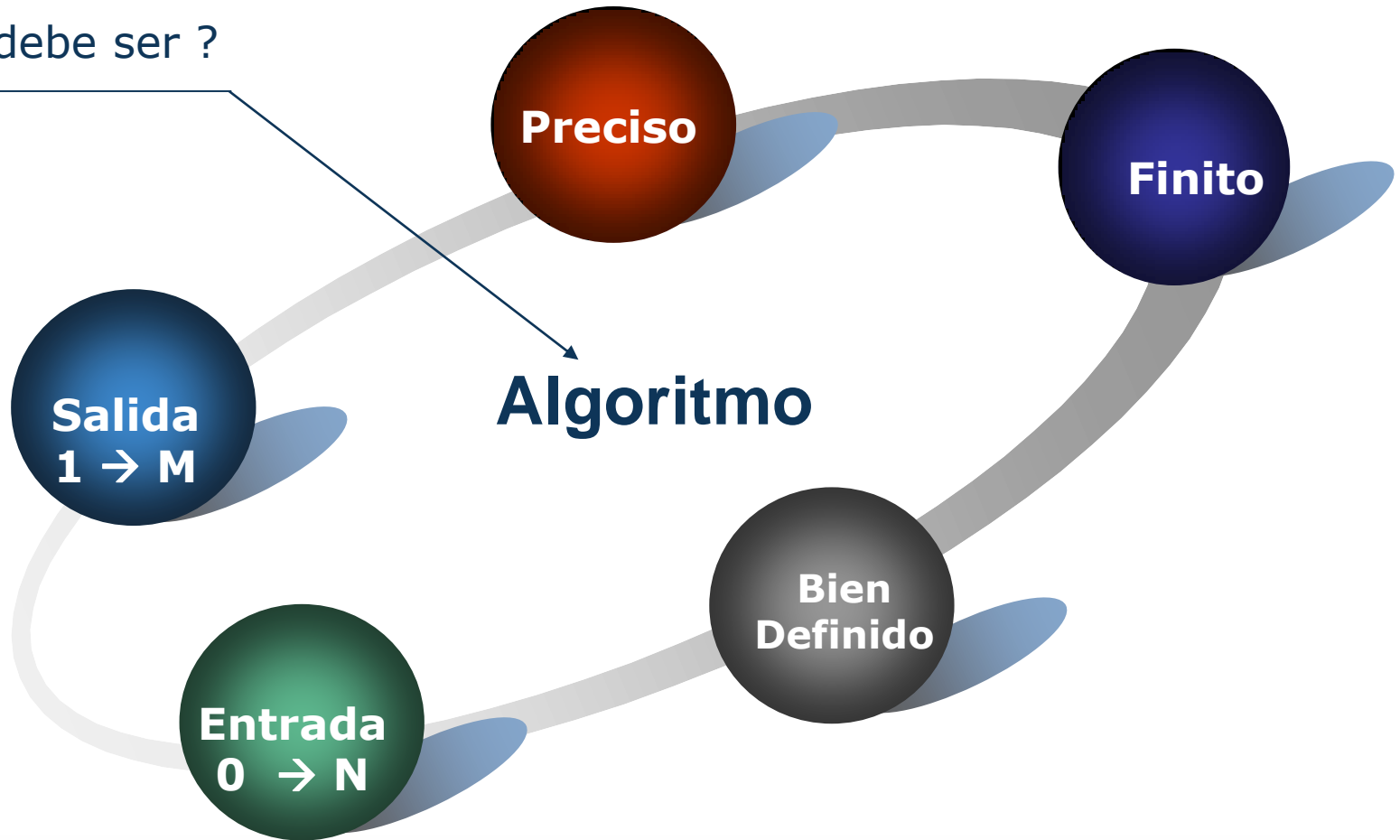
Un algoritmo es un **conjunto finito de instrucciones precisas que realizan una tarea**, la cual, dado un estado inicial, culminará por arrojar un estado final reconocible

Componentes de un Algoritmo



Características de un Algoritmo

Como debe ser ?



Que es el Diseño de un Algoritmo ?

- ❖ Resolver un problema mediante alguna tecnica:
 - *Algoritmos Voraces.*
 - *Algoritmos Recursivos.*
 - *Algoritmos Paralelos.*
 - *Algoritmos Probabilitsticos*
 - *Algoritmos Deterministicos*
 - *Algoritmos No Deterministicos*
 - *Algoritmos Modulares.*

- ❖ Este capitulo basara su estudio mas que todo en *Algoritmos Modulares.*

[2. Análisis de Algoritmos]

1

¿Cuándo realizar análisis de Algoritmos?

Si el propósito es resolver el problema **eficientemente**. Entonces habrá que medir el **grado de complejidad** del mismo.

2

¿Cuál es la base del análisis de Algoritmos?

- Medición del tiempo requerido para resolver **un determinado tamaño de problema con un número de elementos de entrada** en específico.

3

¿Que factores afectan el desempeño del algoritmo?

- Hardware:
Procesador, Memoria, Disco duro.
- Software:
Sistema Operativo, Compilador.

[2. Análisis de Algoritmos]

- ❖ Es una de las herramientas con las que cuenta un ingeniero, para hacer la evaluación de un diseño. A través de éste, es posible establecer la calidad de un programa y compararlo con otros programas que se puedan escribir para resolver el mismo problema, sin necesidad de desarrollarlos.
- ❖ ***El objetivo del análisis de algoritmos es establecer una medida de la calidad de los algoritmos, que permita compararlos sin necesidad de implementarlos. Para esto se asocia al algoritmo una*** función matemática que mida su eficiencia, tomando en cuenta únicamente sus características estructurales.

2.1. Elementos del AA

- ❖ Con el Análisis Algorítmico se lograra:
 - Una medida de la eficiencia dependiente del tamaño de los datos de entrada. Esta medida permitirá comparar con otros algoritmos
- ❖ Tomar en cuenta que algunos factores afectan el desempeño del algoritmo:
 - Los *datos de entrada*
 - La *complejidad* de tiempo del algoritmo.

2.1 Elementos del AA

❖ El Análisis de Algoritmos consta de 2 partes:

- Tiempo: Running Time
 - Calculo del ***contador de frecuencias*** $\rightarrow T(n)$
- Orden de Magnitud: Complejidad Algorítmica
 - ***Factor de crecimiento del algoritmo*** $\rightarrow O(n)$
 - $O(1)$
 - $O(n)$
 - $O(n*n)$
 - $O(2^n)$

2.2.1 El running time

- ❖ El calculo del running time es el conteo de pasos elementales para cierto tamaño de entrada. Cada paso tiene un tiempo constante único (**c**).
- ❖ Esta simplificación se considera adecuada ya que lo que realmente se quiere saber es cómo crece el número de instrucciones necesarias para resolver el problema con respecto a la talla del problema → **Complejidad no tiene que ver con su dificultad, mas bien con el rendimiento!!**
- ❖ Por ejemplo un programa pueden tener un tiempo de ejecución **$T(n) = cn^2$**
 - Donde **c** es una constante.
 - Las **unidades** de $T(n)$ se dejan sin indicar, ya que no es posible especificarla ni siquiera en segundos ya que esto variaría de computador a computador.

2.2.1 El running time

❖ Notese que el comportamiento de un algoritmo puede cambiar notablemente para diferentes entradas.

- Básicamente se estudian tres casos:



Peor caso: mayor número posible de instrucciones ejecutadas (Big O)

- ***Mejor caso: menor número posible de instrucciones ejecutadas (Ω Omega)***
- Caso medio: número de instrucciones igual a la esperanza matemática de la variable aleatoria definida por todos los posibles pasos del algoritmo para un tamaño de la entrada dado.

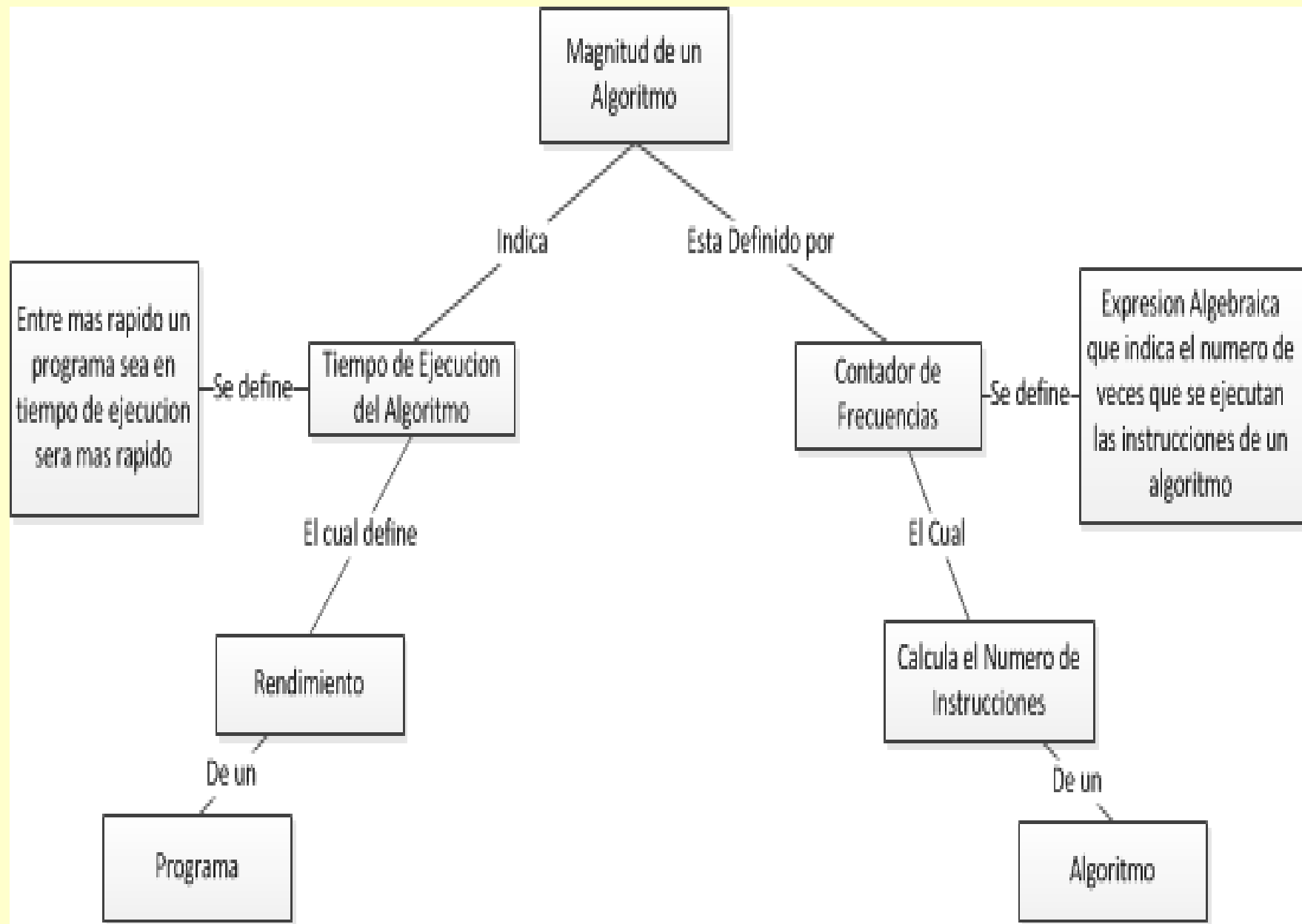
2.2.2 Factor de crecimiento

❖ *El FC es un análisis matemático asintótico que observa el crecimiento del algoritmo.*

- Se dice que un algoritmo tiene comportamiento asintótico cuando el tamaño de las entradas N tiende a infinito.

❖ El Factor de crecimiento indica la magnitud de la complejidad del algoritmo.

Como encontrar la expresión del conteo de frecuencias para N entradas?



- ❖ Para medirla se realizara el conteo de frecuencias:

Leer a, b	→ 1
x=a+b	→ 1
Mostrar a,x	→ 1

Contador de Frecuencia → 3

- Para encontrar una expresión algebraica se utilizara la Notación Asintótica.

3. Notación Asintótica

- ❖ En los siguientes ejercicios se analizará principalmente el caso peor de los algoritmos → Big O (“**o mayúscula**”).
- ❖ Dicho de otro modo se busca:
Cuando el tiempo de ejecución de un algoritmo es **$O(f(n))$** , se dice que tiene una velocidad de crecimiento **$f(n)$** en el **peor de los casos**:

$$T(n) = O(n^2).$$

Se lee “o de n al cuadrado”.

Ordenes de Crecimiento

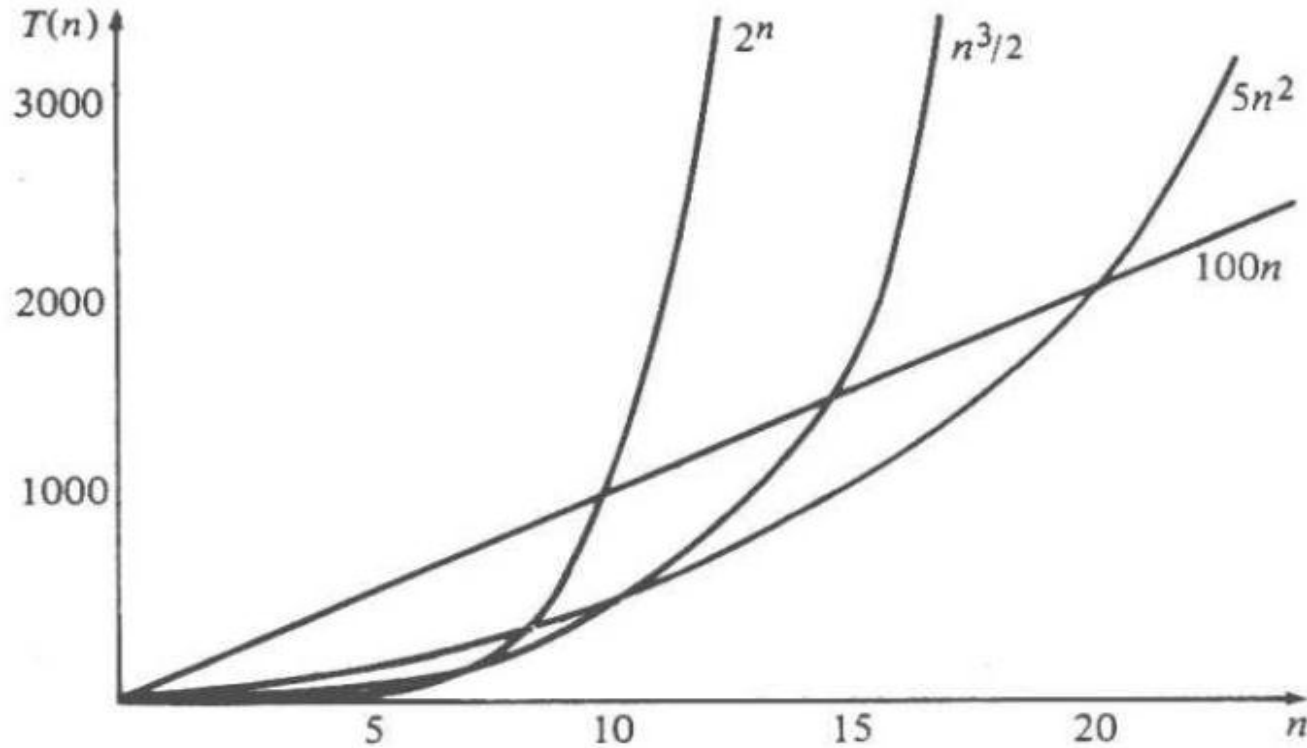
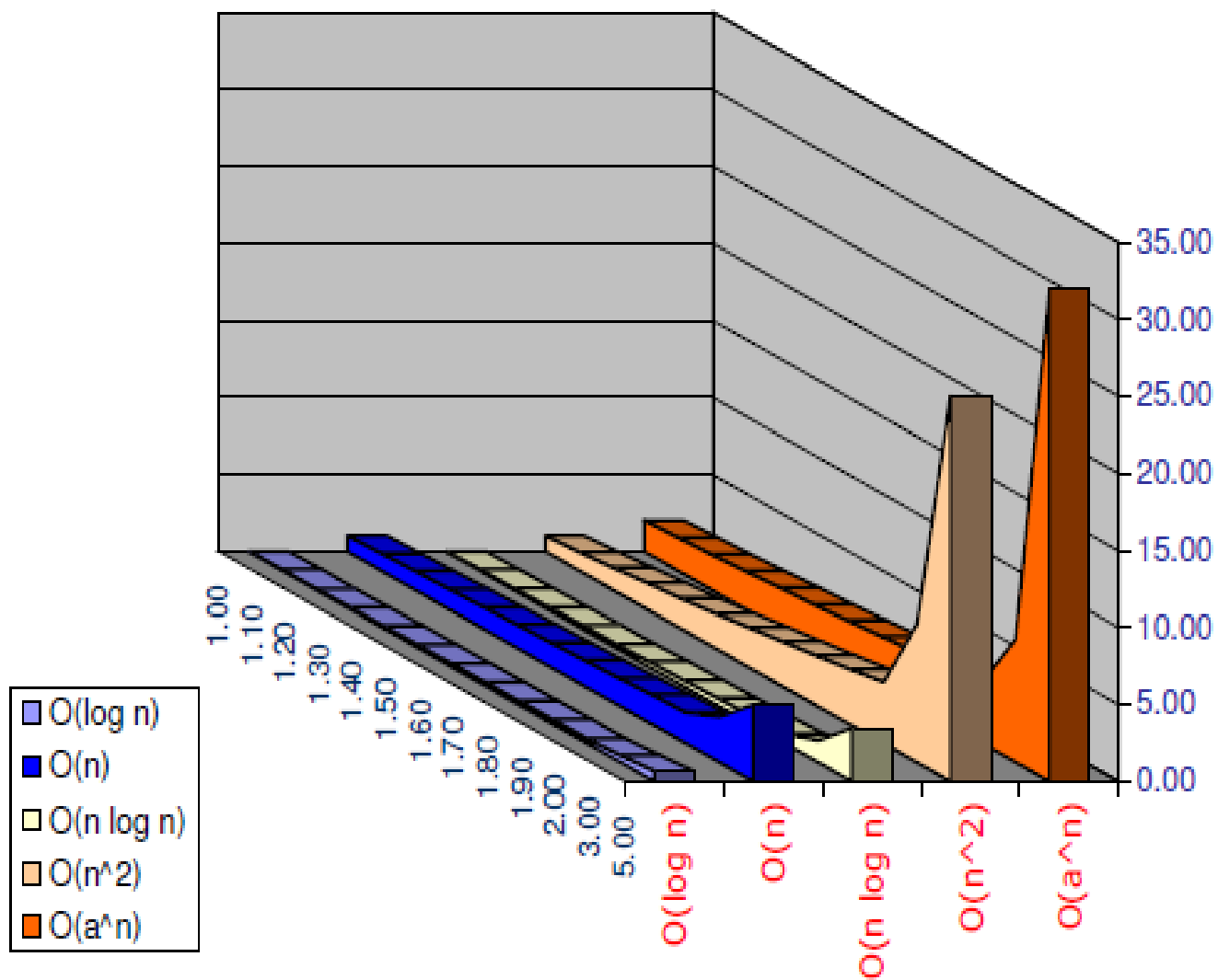


Figura 1.1 Comparación de 4 programas

Orden	Nombre	Comentario
$O(1)$	Constante	Sea cual sea la talla del problema responde a un tiempo constante.
$O(\log n)$	Logarítmico	Los que el tiempo crece con un criterio logarítmico, independiente de cual se la base mientras esta sea mayor que 1. Por este motivo no se indica la base. Implican que un bucle realiza menos iteraciones que la talla del problema (raro). Por ejemplo búsqueda binaria.
$O(n)$	Línea	El tiempo crece linealmente con respecto a la talla. Por ejemplo encontrar el valor máximo de un vector.
$O(n \log n)$	Loglineal	Es un orden relativamente bueno, porque la mayor parte de los algoritmos tienen un orden superior. En este grupo esta el algoritmo de ordenación QuickSort.
$O(n^c)$ con $c > 1$	Polinómico	Este es muy común. Cuando $c=2$ es cuadrático, cuando $c=3$ es cúbico. A partir de este orden el resto son bastante complejos.
$O(c^n)$ con $c > 1$	Exponencial	Este crece muchísimo más rápido que el anterior.
$O(n!)$	Factorial	Son algoritmos que para encontrar una solución prueban todas las combinaciones posibles.
$o(n^m)$	Combinatorio	Es tanto INTRATABLE como el anterior. Usualmente no se los distingue.



3.1 Aritmética Asintótica

- ❖ Se realiza el conteo de frecuencias por cada línea del algoritmo en función a N numero de entradas.
- ❖ Su suman las expresiones de cada línea del algoritmo.
- ❖ El coste máximo que puede esperarse de la ejecución de un algoritmo se basa en el termino dominante (el que mas crece cuando N aumenta).
- ❖ Para obtener el $O(n)$ se establece un orden relativo entre las funciones, mediante la comparación de términos dominantes. Los términos inferiores al dominante se ignoran (valor=0), la constante multiplicativa del termino dominante también se ignora (valor=1).

3.1 Aritmética Asintótica

- ❖ Se quitan los coeficientes, constantes y términos negativos. De cada conjunto de términos dependientes se selecciona el termino dominante (MAYOR). El resultado será la suma que usualmente llega a ser un solo termino.

Ejemplo 3: Si el coste del algoritmo es

$$an^2 + bn + c$$

- Se deprecian la constante sumativa $c=0$
- Se deprecian la constante multiplicativa $a=1$ y $b=1$
- Luego :

$$n^2 + n$$

- Luego el término dominante: $n^2 \rightarrow O(n^2)$

3.2 Reglas de Simplificación

Existen reglas que permiten simplificar la expresión algebraica de tal forma que se pueda llegar a un elemento de la tabla 1.1

A. Serie Aritmética

- La *diferencia* (d) entre términos (t) sucesivos es constante:

$$t, t + d, t + 2d, \dots, t + (n - 1)d$$

- Luego:

$$S_n = [(\text{PrimerTerm} + \text{UltimoTerm})/2] * \text{NroTerm}$$

- Regla útil para conteo de repeticiones en **ciclos o bucles anidados.**

3.2 Reglas de Simplificación

B. Operaciones

Regla de la Suma:

- Si:
T1(n) y T2(n) son tiempos de ejecución de dos algoritmos A1 y A2.
 - Donde:
 $T1(n)$ es $O(f(n))$ y $T2(n)$ es $O(g(n))$
 - Entonces:
 $T1(n) + T2(n)$
 - Luego el tiempo de ejecución de A1 seguido de A2:
 $O(\max(f(n), g(n)))$
- Ejemplo 4: Calcular la complejidad del programa:
 $x = 1; \quad y = 2; \quad z = 3; \quad \rightarrow O(\max(1, 1, 1)) \rightarrow O(1)$

3.2 Reglas de Simplificación

B. Operaciones

Regla del Producto:

- Si:
T1(n) y T2(n) son tiempos de ejecución de dos algoritmos A1 y A2.
 - Donde:
T1(n) es $O(f(n))$ y T2(n) es $O(g(n))$
 - Entonces:
T1(n)*T2(n) es $O(f(n)*g(n))$
- Si **C** es una constante:
 - $O(c*f(n)) = O(f(n))$
- Por ejemplo:
 $O((n^2)/2) \rightarrow O(n^2)$

3.3 Resumen Reglas

Regla 1: Si $f(n) \rightarrow O(g(n))$ y $g(n) \rightarrow O(h(n))$, entonces $f(n) \rightarrow O(h(n))$.

Si una función $g(n)$ es una cota superior para **una función de costo**, entonces cualquier cota superior para $g(n)$, también es una cota superior para **la función de costo**.

Regla 2: Si $f(n) \rightarrow O(k g(n))$ para cualquier constante $k > 0$, entonces $f(n) \rightarrow O(g(n))$

Se puede ignorar cualquier constante multiplicativa en las ecuaciones.

Regla 3: Si $f_1(n) \rightarrow O(g_1(n))$ y $f_2(n) \rightarrow O(g_2(n))$, entonces $f_1(n) + f_2(n) \rightarrow O(\max(g_1(n), g_2(n)))$.

Dadas dos partes de un programa ejecutadas en secuencia, sólo se necesita considerar la parte más cara.

Regla 4: Si $f_1(n) \rightarrow O(g_1(n))$ y $f_2(n) \rightarrow O(g_2(n))$, entonces $f_1(n) * f_2(n) \rightarrow O(g_1(n) * g_2(n))$.

Esta es **para simplificar ciclos o bucles simples en programas**. Si alguna acción es repetida un cierto número de veces, y cada repetición tiene el mismo costo, entonces el costo total es, el costo de la acción multiplicado por el número de veces que la acción tuvo lugar.

3.4 Reglas Practicas

1. Sentencias Simples

- A. Son las sentencias que son independientes al tamaño del problema: asignaciones, entradas, salidas, etc.

Su tiempo de ejecución es constante por lo tanto, su complejidad es $O(1)$

$y=2*x \quad \rightarrow O(1)$

mostrar $x,y \quad \rightarrow O(1)$

- B. El tiempo de ejecución de cada **proposición de asignación**, por lo común, puede tomarse como **$O(1)$** . *Hay unas cuantas excepciones, como aquellos casos en que se permitan llamadas a funciones en proposiciones de asignación.*

$$O(T(\text{Asignación})) = O(1)$$

$$O(T(\text{Asignación})) = O(T(\text{Función}))$$

3.4 Reglas Practicas

2. Sentencias If/Else

- A. Los condicionales suele ser $O(1)$, a menos que estos involucren un llamado a un procedimiento, y siempre se le debe sumar la peor complejidad posible de las alternativas del condicional, bien en la rama afirmativa, o bien en la rama negativa. En decisiones múltiples (*switch*) se tomará la peor de todas las ramas .

B. Por ejemplo

```
If (a>b)  —————> Complejidad=O(1)
{
for (x=1; x<=n; x++) —————> Complejidad=O(n)
suma += x;
}
else
{
suma=0;  —————> Complejidad=O(1)
}.

```

$$O(T(\text{Si-Entonces-Sino})) = O(T(\text{condición})) + \text{Máx}(O(T(\text{Entonces})), O(T(\text{Sino})))$$

[3.4 Reglas Practicas]

3. El tiempo para ejecutar un ciclo es la suma, sobre todas las iteraciones del ciclo, del tiempo de ejecución del cuerpo y del empleado para evaluar la condición de terminación (este último suele ser $O(1)$). A menudo este tiempo es, despreciando factores constantes, el producto del número de iteraciones del ciclo y el mayor tiempo posible para una ejecución del cuerpo.

$$O(T(\text{ciclo})) = O(T(\text{iteraciones})) * O(T(\text{cuerpo}))$$

[3.4 Reglas Practicas]

A. Si **k** es una constante.

```
for (int i= 0; i < K; i++) {  
    algo_de_O(1) ;  
}
```

$$\rightarrow K * O(1) = O(1)$$

B. Si **N** es el limite de las iteraciones

```
for (int i= 0; i < N; i++) {  
    algo_de_O(1) ;  
}
```

$$\rightarrow N * O(1) = O(n)$$

[3.4 Reglas Practicas]

C. Si se tiene ciclos anidados

```
for (int i= 0; i < N; i++) {  
    for (int j= 0; j < N; j++) {  
        algo_de_O(1) ;  
    }  
}
```

$$\rightarrow N * N * O(1) = O(n^2)$$

D. El ciclo anidado con variable dependiente: El ciclo exterior se realiza N veces, el interior 1, 2, ... N veces. \rightarrow Regla de la Serie

```
for (int i= 0; i < N; i++) {  
    for (int j= 0; j < i; j++) {  
        algo_de_O(1);  
    }  
}
```

$$\rightarrow (N * (0+N-1)/2)) * (1) = \\ ((1/2*N^2)-(1/2*N)) = O(n^2)$$

3.4 Reglas Practicas

A veces aparecen bucles multiplicativos, donde la evolución de la variable de control no es lineal ([Ver presentación anterior](#))

E. Si **C** es una variable.

```
c= 1;
while (c < N)
{ algo_de_O(1)
  c= 2*c;
}
```

El valor inicial de "c" es 1, llegando a "2k" al cabo de "k" iteraciones. El número de iteraciones es tal que:

$2^k \geq N \Rightarrow k = \text{es } (\log_2(N))$

[el entero inmediato superior]

Por tanto, la complejidad del bucle es $O(\log n)$.

F. Si **C** es una variable.

```
c= N;
while (c > 1) {
  algo_de_O(1)
  c= c / 2;
}
```

La variable se divide a la mitad en cada vuelta de ciclo.

Por lógica :

$\log_2(N)$ iteraciones

Por tanto, con un orden :

$O(\log n)$.

3.4 Reglas Practicas

G. Si **N** es el limite de iteraciones.

```
for (int i= 0; i < N; i++) {  
    c= i;  
    while (c > 0) {  
        algo_de_O(1)  
        c= c/2;  
    }  
}
```

El bucle interno de orden $O(\log n)$ que se ejecuta N veces, luego el conjunto es de orden **$O(n \log n)$**

4. Llamadas a procedimientos

La complejidad de la llamada a un procedimiento será igual a la complejidad del procedimiento.

Complejidad Total = Sumatoria de cuantas veces
se repite cada línea del algoritmo

$$O(\text{Total}) = \sum_{p=1}^N O_p$$

Ejercicio 1

Supongamos en lenguaje C:

```
for j=1; j<=n; j++ ;  
...a=a+1;
```

¿Cual es la función $T(N)$ del algoritmo?

for j=1; j<=n; j++; → se ejecutara

1	asignacion
n+1	comparaciones
n	incrementos

a=a+1; → se ejecutara n veces

1	asignacion
---	------------

$$T(N) = (1 + N + 1 + N) + N = (2N + 2) + N = 3N + 2$$

¿Cual es el orden de crecimiento del algoritmo?

Los términos no dominantes valen 0 y las constantes valen 1.

Reformulando el $T(n)$:

$$O(N) = N$$

Orden de crecimiento
Lineal.

Resulta lo mismo que aplicar la regla del CICLO (Ver comentario)

Ejercicio 2

Supongamos en lenguaje C :

```
for i=1;i<=n; i++;  
...for j=1; j<=n; j++;  
.....a=a+1;
```

¿Cual es la función $T(N)$ del algoritmo?

for i=1;i<=n; i++ $\rightarrow 1 + n + 1 + n$
for j=1; j<=n; j++; $\rightarrow (1 + n + 1 + n) * n$
a=a+1 $\rightarrow 1 * n * n$

Luego:

$$T(N) = (2N + 2) + (2N + 2)N + NN$$
$$= 2N + 2 + 2N^2 + 2n + N^2$$

¿Cual es el orden de crecimiento del algoritmo?

Los términos no dominantes valen 0 y las constantes valen 1.

Reformulando el $T(n)$:

$$O(N) = N^2$$

Orden de crecimiento Cuadrático.

Resulta lo mismo que aplicar la regla del CICLO (Ver comentario)

Ejercicio 3

Supongamos en un lenguaje X :

```
for i=1 to n:  
...for j=i to n:  
.....a=a+1;
```

¿Cual es la función T(N) del algoritmo?

for i=1 to n → se ejecutara n+1 veces
for j=i to n → se ejecutara (n+1)*(n/2) veces
a=a+1 → se ejecutara (1)*(n+1)*(n/2)

Luego:

$$\begin{aligned} T(N) &= N+1+1/2 N^2 + 1/2 N + 1/2 N^2 + 1/2 N \\ &= N^2 + 3/2 N + 1 \end{aligned}$$

¿Cual es el orden de crecimiento del algoritmo?

Los términos no dominantes valen 0 y las constantes valen 1.

Reformulando el T(n):

$$\mathbf{O(N) = N^2}$$

Orden de crecimiento
Cuadrático.

Resulta lo mismo que aplicar la regla **DE LA SERIE** (Verifique)

Ejercicio 4

Proceso Intercambiar ($\downarrow\uparrow x, \downarrow\uparrow y \in \text{Entero}$)

Constantes

\emptyset

Variables

$\text{aux} \in \text{Entero}$

Acciones

$\text{aux} \leftarrow x$

$x \leftarrow y$

$y \leftarrow \text{aux}$

Si ($N \bmod 2 = 0$) **Entonces**

Para $l \leftarrow 1..N$ **Hacer**

$X \leftarrow X + 1$

Fin Para

Fin Si

Fin Proceso

Ejercicio 5

Búsqueda Binaria

```
int Buscar2(int a[] , int N,int valor){
    int centro=0, int encontro=0
    int izq=0;
    int der=N-1;
    while(izq<=der && encontro=0){
        centro=(izq+der)/2;
        if(a[centro]==valor)
            encontro=1;
        else if(valor<a[centro])
            der=centro-1;
        else
            izq=centro+1;
    }
    if (encontro=1)
        return centro;
    return -1
}
```


¿Qué es la búsqueda binaria?

- ❖ Permite encontrar un elemento dentro de una secuencia (o un arreglo), mientras sus elementos **estén ordenados**.
- ❖ El número que se intenta buscar será llamado **objetivo**.
- ❖ El **espacio de búsqueda** a la secuencia donde **seguramente se encuentra el objetivo**.

¿Cuál es la idea?

- ❖ Inicialmente el espacio de búsqueda es la secuencia completa.
- ❖ La idea es ir reduciendo la cantidad de lugares donde el objetivo pueda estar, es decir, ir reduciendo el **espacio de búsqueda**.
- ❖ En cada paso, el algoritmo compara el valor de en medio (**mediana**) del espacio de búsqueda con el objetivo.
- ❖ Basado en la comparación y utilizando la propiedad de que los elementos están ordenados, **se puede eliminar la mitad del espacio de búsqueda**.

¿Cuál es la idea?

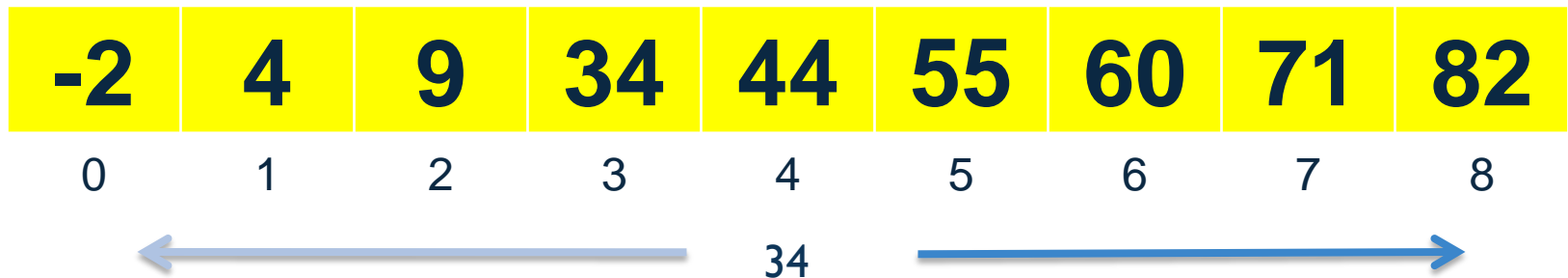
- ❖ Si se continua reduciendo de mitad en mitad, eventualmente se llegara a un **espacio de búsqueda de tamaño 1**.
- ❖ ¿Qué quiere decir esto?
 - El numero ha sido encontrado
 - El numero no existe
- ❖ Si tiene la siguiente secuencia:

-2	4	9	34	44	55	60	71	82
0	1	2	3	4	5	6	7	8

- ❖ Y se busca el **34**

Por ejemplo

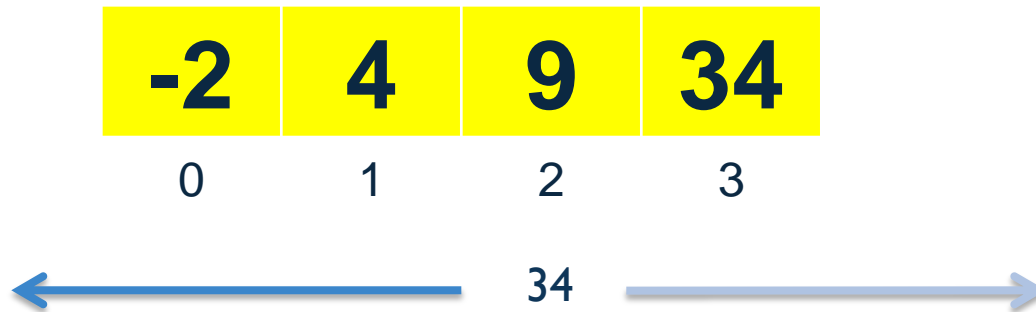
- ❖ Ahora se escoge el valor de en medio del arreglo, que es el valor en el **índice 4** (el punto medio entre **0 y 8**) en este caso el **44**.
- ❖ Se Compara esta mediana con el valor objetivo (**34**)



Por ejemplo...

- Utilizando la propiedad de **ordenación** del arreglo, podemos concluir que el **34 sólo puede estar en la parte izquierda de la secuencia**. ($34 < 44$)
- Por lo que descartamos la mitad del espacio de búsqueda que **no nos sirve**. (Nos quedamos con los elementos **0 a 3**).
- (Nota: Esta la primera operación) \rightarrow 1 búsqueda

Por ejemplo



- El proceso se repite. **34 > 10** por lo tanto tomamos la mitad de la derecha (elemento en la posición 3).
- (Nota: Esta la segunda operación) → 2 búsqueda

Por ejemplo

34

3

34

- ❖ En **3 operaciones** encontramos un elemento en un arreglo ordenado de **tamaño 8**.
- ❖ (Nota: Esta la tercera operación) → 3 búsqueda



Preguntas ...



¿Cuál es el peor caso de una Búsqueda Binaria?

Cuándo el elemento a buscar no se encuentra en el arreglo hace la mayor cantidad de preguntas.



Comportamiento ...

❖ Implementa una Búsqueda Binaria sobre un arreglo de enteros que regrese el índice de la posición donde se encuentra el objetivo.

*Si el elemento no está en el arreglo,
regresar -1.*

❖ ¿Cuál es la condición de paro del ciclo?

**¿CUÁL ES LA COMPLEJIDAD
DE LA BÚSQUEDA BINARIA?**

**ES VALIDO INDICAR QUE SU
COMPLEJIDAD ES $O(N)$?**

Complejidad de la búsqueda binaria

- ❖ La complejidad de la búsqueda binaria no es tan intuitiva como las revisadas anteriormente.
- ❖ Antes de responder debemos analizar detenidamente el algoritmo

¿Qué sucede con el tamaño del espacio de búsqueda en cada paso?

Ejemplo...

❖ Si $n=12$ esto es lo que sucede:



$$\frac{12}{2} = 6$$


$$\frac{6}{2} \approx 3$$

$$\frac{3}{2} = 1$$

Complejidad de la búsqueda binaria

- ❖ Es dividido a la mitad...
- ❖ Entonces la pregunta clave es:
¿Cuántas veces puedo dividir por la mitad a N ?
- ❖ Progresivamente va disminuyendo el número de elementos sobre el que realizar la búsqueda a la mitad hasta quedarse con 1 único elemento

El proceso al revés


$$\frac{12}{2} = 6$$


$$\frac{6}{2} \approx 3$$

$$\frac{3}{2} = 1$$

$$\frac{12}{2^1} = 6$$

$$\frac{12}{2^2} \approx 3$$

$$\frac{12}{2^3} \approx 1$$


$$2^4 = 16$$

$$2^3 = 8$$

$$2^2 = 4$$

$$2^1 = 2$$

$$2^0 = 1$$

Luego

❖ El problema trata de encontrar una ***x*** tal que:

$$\frac{n}{2^x} = 1$$

$$2^x = n$$

$$\log_2 2^x = \log_2 n$$

$$x \log_2 2 = \log_2 n$$

$$x = \log_2 n$$

❖ ***x*** se convierte en el número de pasos realizados.

Conclusión

- Búsqueda binaria tiene complejidad de:

$$O(\log n)$$

- Lo que interesa es SIEMPRE la función de crecimiento, por lo ***que la base del logaritmo*** se lo puede omitir. Además para llevarlo a log en base 10 tendríamos que multiplicar por la constante !!!

The background is a dark blue grid. A wavy, multi-colored line (blue, green, yellow, red) flows from the bottom left towards the center. A large, upward-pointing arrow, also filled with the multi-colored pattern, starts from the bottom left and points towards the top right. On the right side, there is a vertical axis with numerical labels: 16, 17, 18, and 19. The text "Fin Parte I !" is centered in the middle of the image.

Fin Parte I !