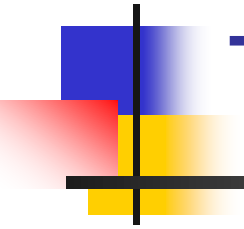


# Taller de Programación

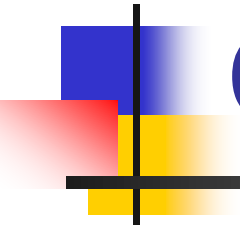
**Parte II**

**Análisis de Algoritmos**

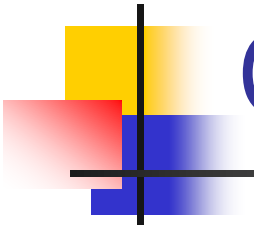
Jhonny Felípez Andrade
jrfelizamigo@yahoo.es

# Contenido

- Consideraciones.
- Complejidad algorítmica.
  - Análisis de estructuras de control.
  - Solución de Recurrencias.
    - Método Iterativo.
    - Teorema Maestro



# Consideraciones



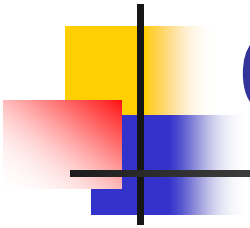
# Consideraciones

¿Dado un problema será posible encontrar más de un algoritmo que sea correcto?

# Consideraciones. Cont.

Dado que sea cierto lo anterior:

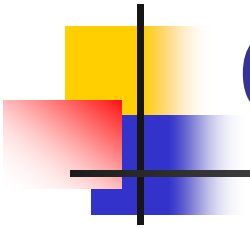
¿Cómo elegir el mejor de ellos?



# Consideraciones. Cont.

Criterio posible: Cantidad de recursos consumidos por el algoritmo.

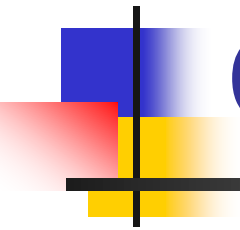
- Espacio de memoria.
- Tiempo de ejecución.



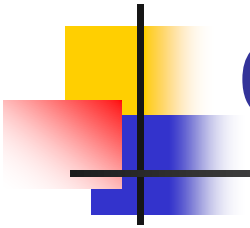
# Consideraciones. Cont.

De lo anterior se ocupa el:

**Análisis de la complejidad de algoritmos**



# Complejidad Algorítmica



# Complejidad de Algoritmos

¿Qué es la complejidad de algoritmos?

La complejidad de algoritmos obtiene una medida de la cantidad que consume un algoritmo de un recurso determinado (espacio o tiempo).

# Complejidad de Algoritmos. Cont.

Por ejemplo existe:

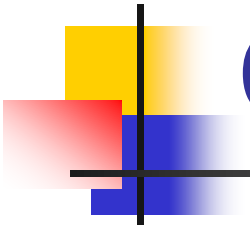
- La Complejidad Espacial o la eficiencia en la memoria, que toma en cuenta a la cantidad de memoria que ocupan todas las variables del algoritmo (no se vera en la materia).
- La Complejidad Temporal o la eficiencia en el tiempo de ejecución.

# Complejidad de Algoritmos. Cont.

Existen dos métodos para obtener la complejidad algorítmica.

- Estimación teórica o a priori.
- Verificación empírica o a posteriori.

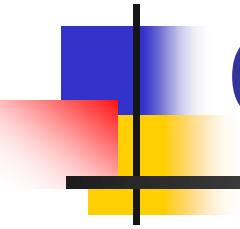
# Complejidad de Algoritmos. Cont.



- El *método empírico* también denominado *a posteriori*, consiste en implementar en un computador los algoritmos a comparar, se prueban para distintos tamaños de los datos del problema y se comparan.

# Complejidad de Algoritmos. Cont.

- El *método teórico* también denominado *a priori*, consiste en determinar matemáticamente la cantidad de recursos necesitados por cada algoritmo como una función cuya variable independiente es el tamaño de los datos del problema.



# Complejidad Algorítmica

Estimación teórica o a priori

# Estimación teórica o priori

Suponiendo que en algún lugar de nuestro programa se tiene la siguiente instrucción:

$$x = x + 1$$

¿Cuánto tiempo tarda en ejecutarse esta instrucción?

¿Cuál es el número de veces que ésta instrucción se ejecuta?

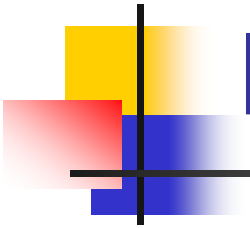
El producto de ambos resultados nos dará el tiempo total que toma esta instrucción!

La segunda respuesta se denomina frecuencia de ejecución

# Estimación teórica o priori

- Es imposible determinar exactamente cuanto tarda en ejecutar una instrucción a menos que se tenga la siguiente información:
  - Características de la máquina.
  - Conjunto de instrucciones de la maquina.
  - Tiempo que tarda una instrucción de maquina.
  - Compilador utilizado.

Lo cual es difícil si se elige una maquina real!.



# Estimación teórica o priori

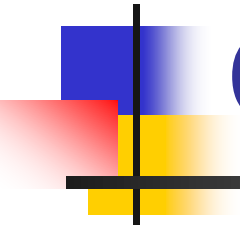
- Otro alternativa para aproximar cuanto tarda en ejecutar una instrucción quizá será definir una maquina hipotética (con tiempos de ejecución imaginarios).

# Estimación teórica o priori

- En ambos casos el tiempo exacto que se determina no es aplicable a muchas maquinas o a ninguna maquina
- Más aún es más dificultoso obtener cifras exactas en tiempo por las limitaciones del reloj y el ambiente de multiprogramación o tiempo compartido

# Estimación teórica o priori

- Todas estas consideraciones nos limitan solamente a obtener la *frecuencia de ejecución* de todas las instrucciones para un *análisis a priori.*



# Complejidad Algorítmica

Estructuras de Control

# Análisis de Estructuras.
# Análisis a priori. Cont.

- Consideremos tres simple programas.

.

.

x = x + 1

.

.

La cantidad de frecuencia es uno

# Análisis de Estructuras.
# Análisis a priori. Cont.

```
for i = 1 to n do
    x = x + 1
end
```

La misma sentencia será ejecutada n veces en el programa.

# Análisis de Estructuras.
# Análisis a priori. Cont.

```
for i = 1 to n do
    for j = 1 to n do
            x = x + 1
    end
end
```

$n^2$ veces (asumiendo $n \geq 1$)

# Análisis de Estructuras.
# Análisis a priori. Cont.

- 1, n y $n^2$ son diferentes.
- Si n = 10 aumentaran en un orden de magnitud próximo a 1, 10 y 100.
- En nuestro análisis de ejecución principalmente determinaremos el Orden de Magnitud de un algoritmo. Esto significa, determinar aquellas sentencias que tiene una frecuencia de ejecución grande.

# Análisis de Estructuras.
# Análisis a priori. Cont.

- Para determinar el Orden de Magnitud, a menudo se presentan fórmulas tales como

$$\sum_{1<=i<=n} 1, \sum_{1<=i<=n} i, \sum_{1<=i<=n} i^2$$

# Análisis de Estructuras.
# Análisis a priori. Cont.

- Formas simples para las tres formulas de anteriores se tienen:

$$n, \frac{n(n+1)}{2}, \frac{n(n+1)(2n+1)}{6}$$

- En general

$$\sum_{1<=i<=n} i^k = \frac{n^{k+1}}{k+1} + terminos\ de\ menor\ grado, k >= 0$$

# Análisis de Estructuras.
## Ejemplo método a priori.

- Calcular el número n-ésimo de Fibonacci. La secuencia Fibonacci comienza como sigue

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ...

- Cada nuevo término es obtenido tomando la suma de los dos anteriores términos. Si llamamos al primer término de la secuencia $F_0$, entonces $F_0 = 0$, $F_1 = 1$ y en general

  $Fn = Fn\text{-}1 + Fn\text{-}2, \quad n \geq 2$

# Análisis de Estructuras.

## Ejemplo obtiene la cantidad de frecuencia para n > 1

| Paso | Algoritmo | Frecuencia |
|---|---|---|
| 1 | `void fibonacci(int n)` | 1 |
| | `{` | |
| | `  int a, b, c, i;` | |
| 2-3 | `  if (n < 0) { printf("Error"); return;}` | 1 |
| 4-5 | `  if (n == 0) { printf("0"); return;}` | 1 |
| 6-7 | `  if (n == 1) { printf("1"); return;}` | 1 |
| 8 | `  a = 0; b = 1;` | 2 |
| 9 | `  for (i = 2; i <= n; i++)` | n |
| | `  {` | |
| 10 | `    c = a + b;` | n-1 |
| 11 | `    a = b;` | n-1 |
| 12 | `    b = c;` | n-1 |
| | `  }` | |
| 13 | `  printf("%d\n",c);` | 1 |
| | `}` | |
| | t(n) = | 4n + 4 |

# Análisis de Estructuras.
## Ejemplo método a priori . Cont.

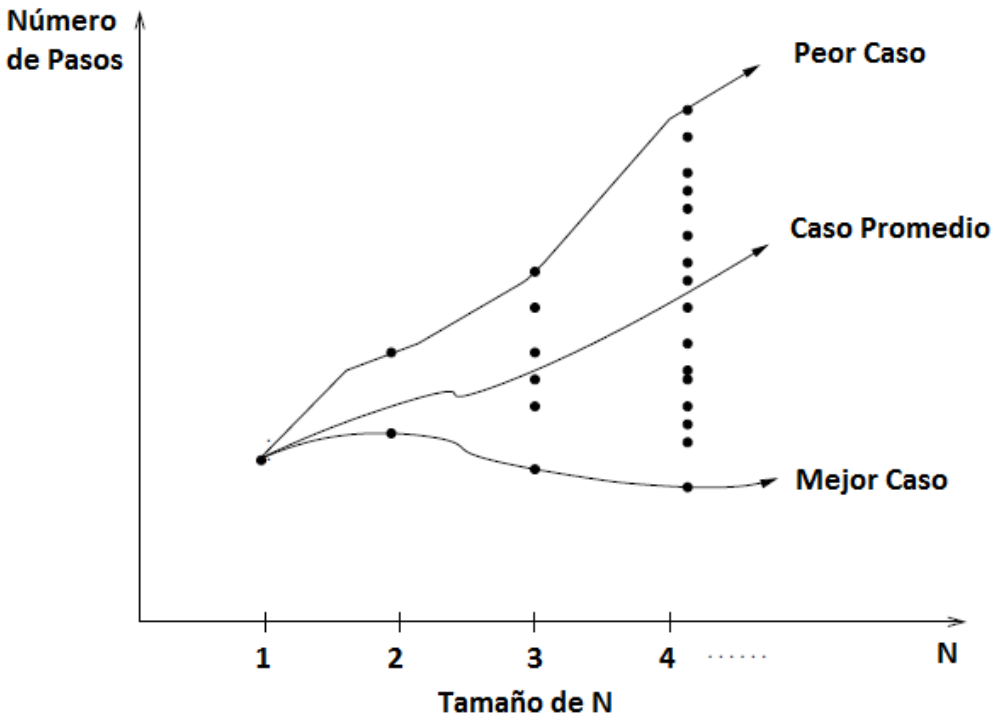
- El tiempo de cálculo es 4n + 4.
- Ignorando las dos constantes 4, obtenemos que la complejidad algorítmica u Orden de Complejidad es O(n).

# Análisis de Estructuras.
## Mejor caso y peor caso.

- El tiempo que toma un algoritmo puede variar considerablemente en función a las diferentes entradas de datos.
- Para el ejemplo con n>1 se da en el peor de los casos y el Orden de Complejidad será O(n).
- En el mejor de los casos se da con n = 0 o 1, entonces el algoritmo tendrá una complejidad constante O(1).
- En la materia se solicitará obtener en el peor de los casos.

# Complejidad en el peor de los casos

- La *complejidad en el peor de los casos* de un algoritmo es la función definida por el mínimo número de pasos que toma en cualquier instancia de tamaño *n*.



# Complejidad
# mejor y promedio de los casos

- La *complejidad en el mejor de los casos* de un algoritmo es la función definida por el mínimo número de pasos que toma en cualquier instancia de tamaño *n*.
- La *complejidad en el caso promedio* de un algoritmo es la función definida por el número promedio de pasos que toma en cualquier instancia de tamaño *n*.
- Para cada una de estas complejidades se define una función numérica: tiempo vrs tamaño!

# Complejidad de Algoritmos.
## Complejidades mas usuales.

- $O(1)$ Complejidad constante.
- $O(\log n)$ Complejidad algorítmica.
- $O(n)$ Complejidad lineal.
- $O(n \log n)$
- $O(n^2)$ Complejidad cuadrática.
- $O(n^3)$ Complejidad cúbica.
- $O(n^k)$ Complejidad polinómica.
- $O(2^n)$ Complejidad exponencial.

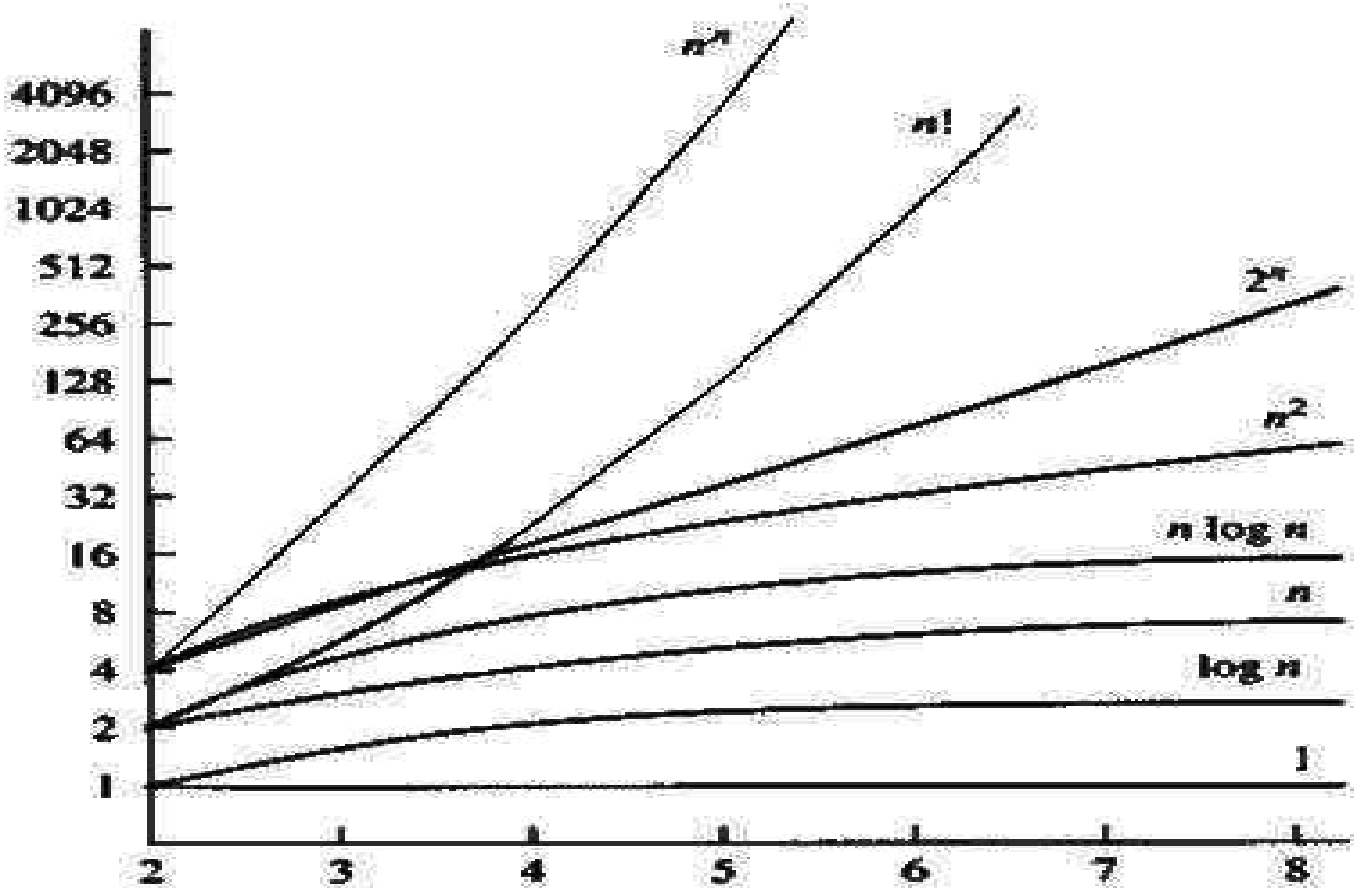
# Complejidad de Algoritmos.

## Valores de crecimiento de las funciones.

| n | log n | n | n log n | $n^2$ | $n^3$ | $2^n$ |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 1 | 2 |
| 2 | 1 | 2 | 2 | 4 | 8 | 4 |
| 4 | 2 | 4 | 8 | 16 | 64 | 16 |
| 8 | 3 | 8 | 24 | 64 | 512 | 256 |
| 16 | 4 | 16 | 64 | 256 | 4096 | 65536 |
| 32 | 5 | 32 | 160 | 1024 | 32768 | 2.147.483.648 |

# Complejidad de Algoritmos.

## Promedio de crecimiento de las funciones.



# Complejidad de Algoritmos. Cont.

- *O(log n), O(n)* y *O(n log n)* crecen mas lentamente que los otros.
- Para grandes conjuntos de datos, los algoritmos con una complejidad mayor que $O(n^3)$ son a menudo no aplicables.
- Un algoritmo que es exponencial trabajara solamente con entradas muy pequeñas.

# Complejidad de Algoritmos.
## Ejemplo.

| Paso | Algoritmo | Frecuencia |
|---|---|---|
| 1 | `int minimo(int *vector, int n)` | 1 |
| | `{` | |
| | `  int i, min;` | |
| 2 | `  min = 1;` | 1 |
| 3 | `  for (i = 2; i <= n; i++)` | n |
| | `  {` | |
| 4 | `    if (vector[i] < vector[min])` | n-1 |
| 5 | `      min = i;` | n-1 |
| | `  }` | |
| 6 | `  return (vector[min]);` | 1 |
| | `}` | |
| | t(n) = | 3n + 1 |

**t(n) ∈ O(n).**

# Complejidad de Algoritmos.
## Ejemplo.

| Paso | Algoritmo | Frecuencia |
|---|---|---|
| 1 | `void burbuja(int *vector, int n)` | 1 |
| | `{` | |
| | `  int i, j, aux;` | |
| 2 | `  for (i = 1; i <= n-1; i++)` | n |
| 3 | `    for (j = i+1; j <= n; j++)` | n n-1 |
| 4 | `      if (vector[i] > vector[j])` | n-1 n-1 |
| | `      {` | |
| 5 | `        aux = vector[i];` | n-1 n-1 |
| 6 | `        vector[i] = vector[j];` | n-1 n-1 |
| 7 | `        vector[j] = aux;` | n-1 n-1 |
| | `      }` | |
| | `}` | |
| | t(n) = | $5n^2 - 8n + 5$ |

**$t(n) \in O(n^2)$.**

# Complejidad de Algoritmos.
## Ejemplo.

| Paso | Algoritmo | Frecuencia |
|---|---|---|
| 1 | `void nuevo()` | 1 |
| | `{` | |
| | `  int x, t;` | |
| 2 | `  x = 1; t = 0;` | 2 |
| 3 | `  while (x < n)` | log n |
| | `  {` | |
| 4 | `    x = x * 2;` | log n - 1 |
| 5 | `    t = t + 1;` | log n - 1 |
| | `  }` | |
| | `}` | |
| | t(n) = | 1+ 3 log n |

**$t(n) \in O(\log n)$.**

# Complejidad de Algoritmos.
## Ejemplo. Cont.

- Calculando el valor de t se tiene:

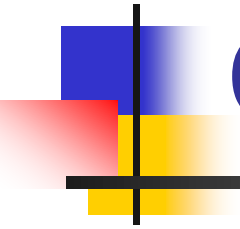
| n | t |
|---|---|
| 1 | 0 |
| 2 | 1 |
| 4 | 2 |
| 8 | 3 |
| 16 | 4 |
| 32 | 5 |

Por lo tanto:

$2^5 = 32$

$=> \quad 2^t = n$

$\log_2 n = t$

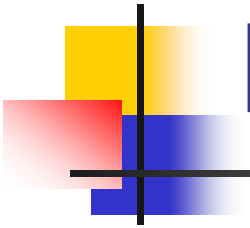


# Complejidad Algorítmica

- Solución de Recurrencia.
  - Método Iterativo.

# Solución de Recurrencias

- El tiempo de ejecución de los algoritmos recursivos se expresa normalmente con una recurrencia.
- Se presentara algunos métodos donde se muestra que a partir de estas ecuaciones de recurrencias, se puede determinar la complejidad algorítmica de estos algoritmos.



# Método Iterativo

- El método iterativo requiere, para obtener la complejidad algorítmica, de un poco de algebra.

# Método Iterativo
## Ejemplo. Búsqueda Binaria.

```
static int busqBinaria(double vector[], int i, int j,
   double x)
{
   int medio=(i+j)/2;
   if (i > j) return -1;
   else if (x == vector[medio])
       return medio;
   else if (x < vector[medio])
       return busqBinaria(vector,i,medio-1,x);
   else
       return busqBinaria(vector,medio+1,j,x);
}
```

# Método Iterativo
## Ejemplo. Búsqueda Binaria. Cont.

$$T(n) = \begin{cases} c_1 & si\ n = 1 \\ T(\frac{n}{2}) + c_2 & en\ caso\ contrario \end{cases}$$

$$T(n) = T(\frac{n}{2}) + c_2 \quad reemplazando\ n\ por\ \frac{n}{2}$$

$$= (T(\frac{n/2}{2}) + c_2) + c_2 = T(\frac{n}{4}) + 2 * c_2 \quad reemp.\ n\ por\ \frac{n}{4}$$

$$= (T(\frac{n}{8}) + c_2) + 2 * c_2 = T(\frac{n}{8}) + 3 * c_2$$

# Método Iterativo
## Ejemplo. Búsqueda Binaria. Cont.

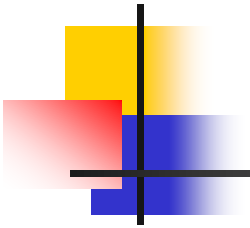
- La forma general será

$$T(n) = T(\frac{n}{2^k}) + k * c_2$$

- Este algoritmo termina cuando llegamos a T(1) donde toma el valor de 1, además $n/2^k = 1$ o $k = \log n$, reemplazando se tiene:

$$T(n) = T(1) + k * c_2 = 1 + c_2 * \log n$$

$$\therefore \quad T(n) \in O(\log n)$$



# Método Iterativo

- Ejemplo: Encontrar el mayor y el menor elemento de un vector
  - Aproximación directa
  - Aplicar Técnica Divide y Vencerás

# Método Iterativo

```
void directoMaxMin(int vector[], int n, int *max, int *min)
{
   int i;
   *max = *min = vector[1];
   for (i=2; i<=n; i++) {
      if (vector[i] > *max) *max = vector[i];
      if (vector[i] < *min) *min = vector[i];
   }
}
```

Requiere *2(n-1)* comparaciones de elementos!

# Método Iterativo
## Ejemplo. Aplic. Divide y Vencerás.

- Para n ≤ 2, hacer 1 comparación:
- Para n grande, dividir el conjunto en dos subconjuntos de tamaño menor y obtener el mayor y menor elemento de cada subconjunto.
- Comparar los mayores/menores elementos de los dos subconjuntos y determinar el mayor/menor de ambos.
- Repetir recursivamente.

# Método Iterativo
## Ejemplo. Aplic. Divide y Vencerás.

```
void MaxMin(int vector[], int i, int j, int *max, int *min)
{
  if (i == j) *max = *min = vector[i]; // Trivial
  else if (i == j-1)
       {//Pequeño
         if (vector[i] < vector[j]) {*max = vector[j];
                                     *min = vector[i];}
         else { *max = vector[i]; *min = vector[j]; }
       } else {// divide P en subproblemas.
                int medio=(i+j)/2, max1, min1;
                MaxMin(vector, i, medio, max, min);
                MaxMin(vector, medio+1, j, &max1, &min1);
                if (*max < max1) *max = max1;
                if (*min > min1) *min = min1;
              }
}
```

# Método Iterativo
## Ejemplo. Cont.

$$T(n) = \begin{cases} 5 & si\ n \leq 2 \\ 2T(\frac{n}{2}) + 7 & en\ caso\ contrario \end{cases}$$

# Método Iterativo
## Ejemplo. Cont.

$$T(n) = 2T(\frac{n}{2}) + 7 \quad reemplazando\ n\ por\ \frac{n}{2}$$

$$= 2(2T(\frac{n}{4}) + 7) + 7$$

$$= 4T(\frac{n}{4}) + (2*7 + 7) \quad reemplazando\ n\ por\ \frac{n}{4}$$

$$= 4(2T(\frac{n}{8}) + 7) + (2*7 + 7)$$

$$= 8T(\frac{n}{8}) + (4*7 + 2*7 + 7)$$

$$S_n = \frac{a(1-r^n)}{1-r} = \frac{7(1-(2)^n)}{1-2} = 7(2^n - 1)$$

# Método Iterativo
## Ejemplo. Cont.

- La forma general será

$$T(n) = 2^k T(\frac{n}{2^k}) + 7(2^k - 1)$$

- Este algoritmo termina cuando llegamos a T(1) donde toma el valor de 5, además $n/2^k = 1$ o $n = 2^k$, reemplazando se tiene:

$$T(n) = 2^k T(1) + 7(2^k - 1) = 5n + 7(n-1) = 12n - 7$$

$$\therefore \quad T(n) \in O(n)$$

# Método Iterativo

## Ejemplo.

```
void DC(int vector[], int n)
{
  int j;
  if (n <= 1) return;
  else{
    for (j = 1; j <= 8; j++)
      DC(vector, n/2);
    for (j = 1; j <= n*n*n; j++)
      vector[j] = 0;
  }
}
```

# Método Iterativo
## Ejemplo. Cont.

$$T(n) = \begin{cases} 1 & si\ n \leq 1 \\ 8T(\frac{n}{2}) + n^3 & en\ caso\ contrario \end{cases}$$

# Método Iterativo
## Ejemplo. Cont.

$$T(n) = 8T(\frac{n}{2}) + n^3 \quad reemplazando\ n\ por\ \frac{n}{2}$$

$$= 8(8T(\frac{n}{4}) + (\frac{n}{2})^3) + n^3$$

$$= 64T(\frac{n}{4}) + 2n^3 \quad reemplazando\ n\ por\ \frac{n}{4}$$

$$= 64(8T(\frac{n}{8}) + (\frac{n}{4})^3) + 2n^3$$

$$= 512T(\frac{n}{8}) + 3n^3$$

# Método Iterativo
## Ejemplo. Cont.

- La forma general será

$$T(n) = (2^k)^3 T(\frac{n}{2^k}) + kn^3$$

- Este algoritmo termina cuando llegamos a T(1) donde toma el valor de 1, además $n/2^k=1$ o $n=2^k$ *o* $k = \log n$, reemplazando se tiene:

$$T(n) = T(1) + (\log n)n^3 = n^3 + n^3 \log n$$

$$\therefore \quad T(n) \in O(n^3 \log n)$$

# Método Iterativo
## Ejemplo.

```
void DC(int vector[], int n)
{
  int j;
  if (n <= 1) return;
  else{
    DC(vector, n/2);
    for (j = 1; j <= n*n*n; j++)
      vector[j] = 0;
  }
}
```

# Método Iterativo
## Ejemplo. Cont.

$$T(n) = \begin{cases} 1 & si\ n \leq 1 \\ T(\frac{n}{2}) + n^3 & en\ caso\ contrario \end{cases}$$

# Método Iterativo
## Ejemplo. Cont.

$$T(n) = T(\frac{n}{2}) + n^3 \quad reemplazando\ n\ por\ \frac{n}{2}$$

$$= (T(\frac{n}{4}) + (\frac{n}{2})^3) + n^3$$

$$= T(\frac{n}{4}) + (1+\frac{1}{8})n^3 \quad reemplazando\ n\ por\ \frac{n}{4}$$

$$= (T(\frac{n}{8}) + (\frac{n}{4})^3) + (1+\frac{1}{8})n^3$$

$$= T(\frac{n}{8}) + (1+\frac{1}{8}+\frac{1}{8^2})n^3$$

$$S_n = \frac{a(1-r^n)}{1-r} = \frac{1-(1/8)^n}{1-1/8} = \frac{8}{7}(1-(\frac{1}{8})^n)$$

# Método Iterativo
## Ejemplo. Cont.

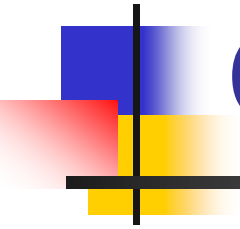
- La forma general será

$$T(n) = T(\frac{n}{2^k}) + \frac{8}{7}(1 - \frac{1}{(2^k)^3})n^3$$

- Este algoritmo termina cuando llegamos a T(1) donde toma el valor de 1, además $n/2^k = 1$ o $n = 2^k$, reemplazando se tiene:

$$T(n) = T(1) + \frac{8}{7}(1 - \frac{1}{n^3})n^3 = 1 + \frac{8}{7}n^3 - \frac{8}{7} = \frac{8}{7}n^3 - \frac{1}{7}$$

$$\therefore \quad T(n) \in O(n^3)$$



# Complejidad Algorítmica

- Solución de Recurrencia.
  - Teorema Maestro.

# Teorema Maestro (Master Theorem)

- Se usa para resolver recurrencias del tipo $T(n) = a\,T(n/b) + f(n)$, donde $f(n)$ es una función asintóticamente positiva y las constantes $a \geq 1$, $b > 1$.
- Este teorema expresa que el problema se divide en $a$ sub problemas de tamaño $n/b$.
- Los $a$ sub problemas se resuelven recursivamente en tiempo $T(n/b)$.
- El costo de dividir el problema y combinar sus soluciones es descrito por $f(n)$.

# Teorema Maestro (Master Theorem)

- Sean a≥1, b>1 constantes, *f(n)* una función y *T(n)* definida para enteros no negativos por la recurrencia:

$$T(n) = aT(n/b) + cn^k$$

Donde n/b es round(n/b). Entonces *T(n)* puede ser acotada asintóticamente por:

# Teorema Maestro (Master Theorem)

$$T(n) \in O(n^{\log_b a}) \; si \; a > b^k$$

$$T(n) \in O(n^k \log n) \; si \; a = b^k$$

$$T(n) \in O(n^k) \; si \; a < b^k$$

# Teorema Maestro (Master Theorem)
## Ejemplo. Búsqueda Binaria.

```
int busqBinaria(int vector[], int i, int j, int x)
{
   if (i == j) return i;
   else
   {
      int medio=(i+j)/2;
      if (x <= vector[medio])
         busqBinaria(vector,i,medio,x);
      else
         busqBinaria(vector,medio+1,j,x);
   }
}
```

# Teorema Maestro (Master Theorem)

## Ejemplo. Búsqueda Binaria. Cont.

$$T(n) \in O(n^{\log_b a}) \; si \; a > b^k$$

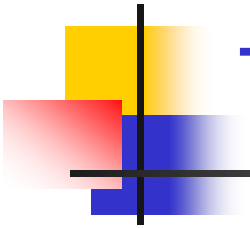
$$T(n) \in O(n^k \log n) \; si \; a = b^k$$

$$T(n) \in O(n^k) \; si \; a < b^k$$

$$T(n) = \begin{cases} 1 & si \; n = 1 \\ T(n/2) + 3 \; en \; caso \; contrario \end{cases}$$

$$a = 1, \quad b = 2, \quad k = 0 \quad \Rightarrow \quad 1 = 2^0$$

$$\therefore \quad T(n) \in O(n^k \log n) = O(n^0 \log n) = O(\log n)$$



# Teorema Maestro (Master Theorem)

- Ejemplo: Encontrar el mayor y el menor elemento de un vector
  - Aplicar Técnica Divide y Vencerás

# Teorema Maestro (Master Theorem)
## Ejemplo. Aplic. Divide y Vencerás. Cont.

```
void MaxMin(int vector[], int i, int j, int *max, int *min)
{
  if (i == j) *max = *min = vector[i]; // Trivial
  else if (i == j-1)
       {//Pequeño
         if (vector[i] < vector[j]) {*max = vector[j];
                                     *min = vector[i];}
         else { *max = vector[i]; *min = vector[j]; }
       } else {// divide P en subproblemas.
                int medio=(i+j)/2, max1, min1;
                MaxMin(vector, i, medio, max, min);
                MaxMin(vector, medio+1, j, &max1, &min1);
                if (*max < max1) *max = max1;
                if (*min > min1) *min = min1;
              }
}
```

# Teorema Maestro (Master Theorem)
## Ejemplo. Aplic. Divide y Vencerás. Cont.

$$T(n) \in O(n^{\log_b a}) \; si \; a > b^k$$
$$T(n) \in O(n^k \log n) \; si \; a = b^k$$
$$T(n) \in O(n^k) \; si \; a < b^k$$

$$T(n) = \begin{cases} 5 & si \; n \le 2 \\ 2T(n/2) + 7 & en \; caso \; contrario \end{cases}$$

$$a = 2, \quad b = 2, \quad k = 0 \quad \Rightarrow \quad 2 > 2^0$$

$$\therefore \quad T(n) \in O(n^{\log_b a}) = O(n^{\log_2 2}) = O(n)$$

# Teorema Maestro (Master Theorem)

## Ejemplo.

$$T(n) \in O(n^{\log_b a}) \; si \; a > b^k$$

$$T(n) \in O(n^k \log n) \; si \; a = b^k$$

$$T(n) \in O(n^k) \; si \; a < b^k$$

$$T(n) = T(2n/3) + 1$$

$$a = 1, \quad b = 3/2, \quad k = 0 \quad \Rightarrow \quad 1 = (3/2)^0$$

$$\therefore \quad T(n) \in O(n^k \log n) = O(n^0 \log n) = O(\log n)$$

# Teorema Maestro (Master Theorem)

## Ejemplo.

```
void DC(int vector[], int n)
{
  int j;
  if (n <= 1) return;
  else{
    DC(vector, n/2);
    for (j = 1; j <= n*n*n; j++)
      vector[j] = 0;
  }
}
```

# Teorema Maestro (Master Theorem)

## Ejemplo.

$$T(n) \in O(n^{\log_b a}) \; si \; a > b^k$$

$$T(n) \in O(n^k \log n) \; si \; a = b^k$$

$$T(n) \in O(n^k) \; si \; a < b^k$$

$$T(n) = \begin{cases} 1 & si \; n \leq 1 \\ T(n/2) + n^3 & en \; caso \; contrario \end{cases}$$

$$a = 1, \quad b = 2, \quad k = 3 \quad \Rightarrow \quad 1 < 2^3$$

$$\therefore \quad T(n) \in O(n^k) = O(n^3)$$

# Teorema Maestro (Master Theorem)
## Ejemplo.

```
void DC(int vector[], int n)
{
  int j;
  if (n <= 1) return;
  else{
    for (j = 1; j <= 8; j++)
      DC(vector, n/2);
    for (j = 1; j <= n*n*n; j++)
      vector[j] = 0;
  }
}
```

# Teorema Maestro (Master Theorem)

## Ejemplo.

$$T(n) \in O(n^{\log_b a}) \; si \; a > b^k$$

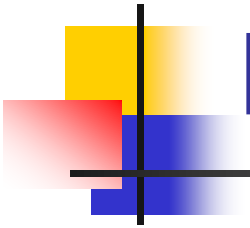
$$T(n) \in O(n^k \log n) \; si \; a = b^k$$

$$T(n) \in O(n^k) \; si \; a < b^k$$

$$T(n) = \begin{cases} 1 & si \; n \leq 1 \\ 8T(n/2) + n^3 & en \; caso \; contrario \end{cases}$$

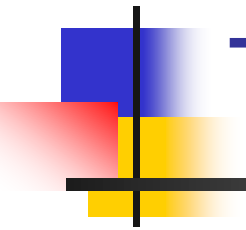
$$a = 8, \quad b = 2, \quad k = 3 \quad \Rightarrow \quad 8 = 2^3$$

$$\therefore \quad T(n) \in O(n^k \log n) = O(n^3 \log n)$$



# Bibliografía


- Fundamentos de Programación, Jorge Teran Pomier, 2006.
- Fundamentals of Data Structures, Ellis Horowitz, 1981.



# Taller de Programación

Gracias