# Análisis y Diseño de Algoritmos

## Análisis de Programas Recursivos

**Arturo Díaz Pérez**

- Introducción
- Programas Recursivos
- Análisis de Funciones Recursivas
- Relaciones de recurrencia para evaluar programas recursivos



## Introducción

- Las reglas generales de análisis nos permiten analizar programas y algoritmos con estructuras de control convencionales

```
void Burbuja( int A[], int n)
{
   int  i, j, temp;
   for( i = 0; i < n-1; i++ )
      for( j = n-1; j >= i+1; j-- )
         if( A[j-1] > A[j] ) {
            temp = A[j-1];                          O(n)
            A[j-1] = A[j];        O(1)      O(n)
            A[j] = temp;
         }
}
```

## Programas Recursivos

- En muchas ocasiones es “mejor” escribir algoritmos (programas) recursivos
  - Claridad
  - Espacio
  - Elegancia
- Durante la década de los 70’s existían cursos sobre programación recursiva
- La programación recursiva se utiliza ampliamente en los lenguajes funcionales y programación lógica
  - LISP
  - Prolog
  - Mathematica
  - Maple

*Análisis y Diseño de Algoritmos* *Análisis-3*

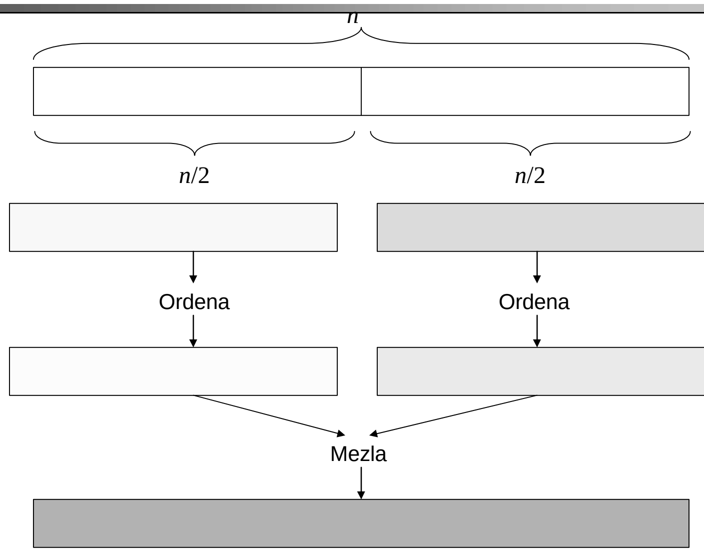
## Ejemplo: Factorial de *n*

$$n! = n\,(n-1)!, \qquad \text{Si } n > 0$$
$$n! = 1 \qquad \text{Si } n = 0$$

```
long  factorial( long n )
{
   if( n <= 0 )
      return 1;
   else
      return  n*factorial(n-1);
}
```

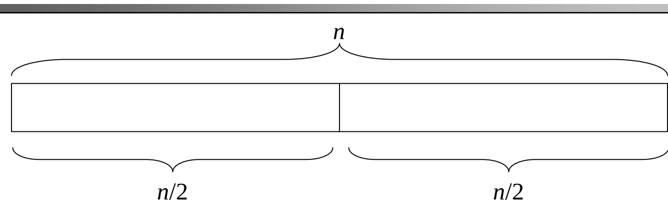
*Análisis y Diseño de Algoritmos* *Análisis-4*

## MergeSort






## MergeSort

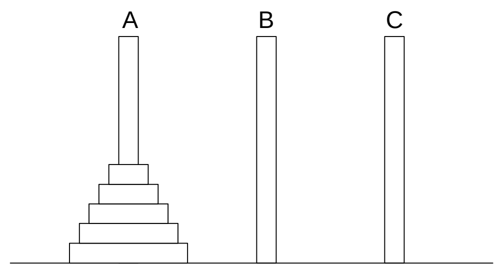



```
void Sort( int A, int n )
{
   int    A1[], A2[];

   if( n > 1 ) {
      /* Divide a A en dos mitades, A1 y A2,
         cada una de longitud n/2   */
      Sort( A1, n/2 );
      Sort( A2, n/2 );
      Mezcla( A1, A2, A);
   }
}
```

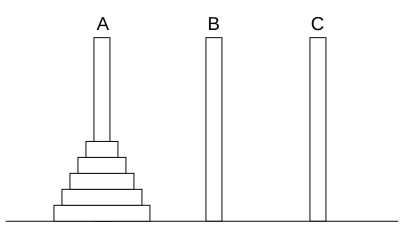
# Las Torres de Hanoi

- Las torres de Hanoi
  - Se tienen 3 postes, A, B, y C. Inicialmente el poste A tiene apilados un número de discos, iniciado con el diámetro más grande en el fondo hasta el más pequeño en el tope.
  - El problema es mover los discos de un poste a otro, uno a la vez, sin colocar nunca un disco de diámetro mayor sobre uno de diámetro menor.






# Las Torres de Hanoi




```
struct Poste P[3];

void Hanoi( int n, int A, int B, int C )
{
   if(n > 1){
      Hanoi(n-1, A, C, B);
      MueveDisco(A, B);
      Hanoi(n-1, C, B, A);
   } else
      MueveDisco(A,B);
}
```

¿Cómo analizar algoritmos o programas recursivos?



## Análisis de Funciones Recursivas

- Análisis
  - Con cada procedimiento recursivo se asocia una función de tiempo desconocido $T(n)$, donde $n$ mide el tamaño de los argumentos al procedimiento
  - Se puede obtener una recurrencia para $T(n)$, esto es una ecuación para $T(n)$ en términos de $T(k)$ para varios valores de $k$
    - $T(n) = \ldots T(k) \ldots$

## Análisis de Funciones Recursivas

```
long factorial( long n )
{
  if( n <= 0 )
    return 1;
  else
    return n*factorial(n-1);
}
```

$$T(n) = \begin{cases} c + T(n-1) & si \quad n > 0 \\ d & si \quad n \le 0 \end{cases}$$

- Si $n > 2$, se tiene que
  - $T(n) = c + T(n\text{-}1) = c + c + T(n\text{-}2) = 2c + T(n\text{-}2)$
- Si $n > 3$
  - $T(n) = 2c + T(n\text{-}2) = 2c + c + T(n\text{-}3) = 3c + T(n\text{-}3)$
- En general, si $n > i$
  - $T(n) = ic + T(n\text{-}i)$
- Finalmente, si $i = n\text{-}1$
  - $T(n) = (n\text{-}1)c + T(1) = (n\text{-}1)c + d$
- de aquí se concluye que $T(n)$ es un $O(n)$

*Análisis y Diseño de Algoritmos* *Análisis*-11

## MergeSort

```
void Sort( int A, int n )
{
  int  A1[], A2[];

  if( n > 1 ) {
    /* Divide a A en dos mitades, A1 y A2,
       cada una de longitud n/2  */
    Sort( A1, n/2 );
    Sort( A2, n/2 );
    Mezcla( A1, A2, A);
  }
}
```

$$T(n) = \begin{cases} c_1 & si \quad n \le 1 \\ 2T(\frac{n}{2}) + c_2 n & si \quad n > 1 \end{cases}$$

*Análisis y Diseño de Algoritmos* *Análisis*-12

## Resolución de Recurrencias

- Estrategias para resolver una recurrencia
  - Adivinar una solución
  - Transformar a alguna expresión con solución conocida
  - Calcular la fórmula cerrada
    - Expandiendo la recurrencia

*Análisis y Diseño de Algoritmos* *Análisis*-13

## Adivinando una Solución

$$T(n) = \begin{cases} c_1 & si \quad n \le 1 \\ 2T(\frac{n}{2}) + c_2 n & si \quad n > 1 \end{cases}$$

- Supongamos que $T(n) \le a\, n \log n + b$ para algunas constantes $a$ y $b$
- Probemos por inducción, que $T(n) \le an \log n + b$ (*)
  - a) Para $n = 1$, $c_1 \le a\log 1 + b \quad \Rightarrow \quad c_1 \le b$

*Análisis y Diseño de Algoritmos* *Análisis*-14

## Adivinando una Solución

- b) Supongamos que $T(n) \leq an \log n + b$ (*) es válida para todo $k < n$.
- Demostremos que se cumple también para $n$
  - Se tiene que $T(n) \leq 2\, T(n/2) + c_2 n$
  - Entonces
    - $T(n) \leq 2[a(n/2)\log(n/2) + b] + c_2 n \quad \Rightarrow$
    - $T(n) \leq an \log n - an \log 2 + 2b + c_2 n \Rightarrow$
    - $T(n) \leq an \log n - an + c_2 n + 2b$
  - Por un lado,
    - Si $a \geq c_2 + b \quad \Rightarrow -an \leq -c_2 n - bn$, y
    - Si $n \geq 1 \qquad \Rightarrow -bn \leq -b \Rightarrow -c_2 n - bn \leq -c_2 n - b$
  - Entonces, si $a \geq c_2 + b$ y $n \geq 1$, se tiene que $-an + c_2 n + b \leq 0$
  - Por lo tanto, $T(n) \leq an \log n + b$
- Luego entonces, si $b \geq c_1$ y $a \geq c_2 + b$ se tiene que $T(n) \leq an \log n + b \quad \forall n \geq 1$

De aquí, $T(n)$ es un $O(n\log n)$

*Análisis y Diseño de Algoritmos* *Análisis*-15

## Observaciones

- Si se supone que $T(n)$ es un $O(f(n))$ y al intentar probar por inducción que $T(n) \leq cf(n)$ no se tiene éxito, no se sigue de esto que $T(n)$ no es un $O(f(n))$.
  - En algunos casos una prueba por inducción de la forma
    - $T(n) \leq cf(n) - 1$

    puede tener éxito.

*Análisis y Diseño de Algoritmos* *Análisis*-16

## Ejemplo

- Considere la recurrencia
  - $T(n) = T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) + 1$
  - Adivinamos que la solución es $O(n)$.
  - Tratamos de mostrar que $T(n) \leq cn$, para una constante $c$ apropiada.
  - $T(n) \leq c\lfloor n/2 \rfloor + \lceil n/2 \rceil) + 1$
  - $= cn + 1$.
  - Lo cual no implica que $T(n) \leq cn$.
  - Tratemos ahora con $T(n) \leq cn - b$, para constantes $c$ y $b \geq 0$.
  - $T(n) \leq (c\lfloor n/2 \rfloor - b) + (c\lceil n/2 \rceil - b) + 1$
  - $= cn - 2b + 1$.
  - $\leq cn - b$, siempre que $b \geq 1$.
  - Lo cual no implica que $T(n) \leq cn$

*Análisis y Diseño de Algoritmos* — *Análisis*-17

## Cambio de Variables

- Considere ahora la recurrencia
  - $T(n) = 2T(\lceil \sqrt{n} \rceil) + \log_2 n$
  - Hagamos $m = \log_2 n$, entonces, $n = 2^m$ y
    - $T(2^m) = 2T(\lceil 2^{m/2} \rceil) + m$
  - Esto es, $S(m) = 2S(\lceil m/2 \rceil) + m$
  - Por lo tanto, $S(m) = O(m\log m)$.
  - De aquí, $T(n) = O(\log n \log\log n)$.

*Análisis y Diseño de Algoritmos* — *Análisis*-18

## Observaciones

- Si se supone que $T(n)$ es un $O(f(n))$ y al intentar probar por inducción que $T(n) \leq cf(n)$ no se tiene éxito, no se sigue de esto que $T(n)$ no es un $O(f(n))$.
  - En algunos casos una prueba por inducción de la forma
    - $T(n) \leq cf(n) - 1$

    puede tener éxito.
- Al adivinar que $T(n)$ es un $O(f(n))$ no se garantiza que $f(n)$ es la menor cota superior al rango de crecimiento de $T(n)$. Puede existir alguna otra solución con un índice de crecimiento menor.

*Análisis y Diseño de Algoritmos* *Análisis*-19

## Técnica General

- Supongamos que se parte de la ecuación de recurrencia siguiente:

$$T(1) = c$$
$$T(n) \leq g(T(\tfrac{n}{2}), n), \qquad n > 1$$

- Supongamos además que se adivina la solución representada por la función:

$$f(a_1, \ldots, a_l, n)$$

  la cual depende de los parámetros $a_1, a_2, \ldots, a_l$ y de $n$.
- Entonces, se tendría que probar que

$$T(n) \leq f(a_1, \ldots, a_l, n)$$

  para todo $n > n_0$ y para algunos valores de $a_1, a_2, \ldots, a_l$.

*Análisis y Diseño de Algoritmos* *Análisis*-20

## Técnica General

- Así, se debe satisfacer que

$$f(a_1,\ldots,a_l,1) \geq c \qquad (i)$$
$$f(a_1,\ldots,a_l,n) \geq g(f(a_1,\ldots,a_l,\tfrac{n}{2}),n) \qquad (ii)$$

- Así se obtendría que

$$T(n) \leq g(f(a_1,\ldots a_l,\tfrac{n}{2}),n)$$

y de aquí que

$$T(n) \leq f(a_1,\ldots,a_l,n)$$

*Análisis y Diseño de Algoritmos* *Análisis*-21

## Expandiendo Recurrencias

- Si no se puede adivinar una solución o no se está seguro de que se ha encontrado la mejor cota superior para $T(n)$, se puede tratar de expandir la ecuación de recurrencia.
- Para el ejemplo, si $n \geq 2$,

$$T(n) \leq 2T\left(\tfrac{n}{2}\right) + c_2 n$$

- Si $n \geq 4$,

$$T\left(\tfrac{n}{2}\right) \leq 2T\left(\tfrac{n}{4}\right) + c_2\,\tfrac{n}{2}$$

- Entonces

$$T(n) \leq 2\left[2T\left(\tfrac{n}{4}\right) + c_2\,\tfrac{n}{2}\right] + c_2 n = 4T\left(\tfrac{n}{4}\right) + 2c_2 n$$

- Similarmente, si $n \geq 8$,

$$T(n) \leq 4\left[2T\left(\tfrac{n}{8}\right) + c_2\,\tfrac{n}{4}\right] + 2c_2 n$$

$$T(n) \leq 8T\left(\tfrac{n}{8}\right) + 3c_2 n$$

*Análisis y Diseño de Algoritmos* *Análisis*-22

## Expandiendo Recurrencias

- Se puede ver que si $n \geq 2^i$, se tiene que

$$T(n) \leq 2^i T\left(\frac{n}{2^i}\right) + ic_2 n$$

- Suponiendo que $n = 2^k$, este proceso termina tan pronto como se obtiene $T(1)$ en la parte derecha. Así

$$T(n) \leq 2^k T(1) + kc_2 n$$

- Ya que $2^k = n \Rightarrow \log_2 n = k$. Como $T(1) \leq c_1$, entonces se cumple que

$$T(n) \leq c_1 n + c_2 n \log n$$

- De aquí que $T(n)$ es un $O(n \log n)$

*Análisis y Diseño de Algoritmos* *Análisis*-23

## Solución General

- Suponga que un problema de tamaño $n$ se divide en $a$ subproblemas de tamaño $n/b$
  - se asume que un problema de tamaño 1 toma una unidad de tiempo
  - el tiempo para juntar las soluciones de los subproblemas para resolver el problema de tamaño $n$ toma un tiempo $d(n)$
  - Ecuación de recurrencia

$$T(1) = 1$$
$$T(n) = aT\left(\frac{n}{b}\right) + d(n)$$

- esta ecuación sólo se aplica a $n$'s que son potencias enteras de $b$.
  - Si $T(n)$ es suave, se obtiene una cota superior en $T(n)$ para aquellos valores de $n$ (potencias enteras de $b$),
  - Dice como crece $T(n)$ en general
- $d(n)$ es arbitraria, por eso la ecuación de recurrencia se expresa en forma exacta y no como una inecuación

*Análisis y Diseño de Algoritmos* *Análisis*-24

## Solución General, cont.

$$T(1) = 1$$
$$T(n) = aT\left(\tfrac{n}{b}\right) + d(n)$$

$$\begin{aligned}
T(n) &= aT\left(\frac{n}{b}\right) + d(n) \\
&= a\left[aT\left(\frac{n}{b^2}\right) + d\left(\frac{n}{b}\right)\right] + d(n) = a^2T\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d(n) \\
&= a^2\left[aT\left(\frac{n}{b^3}\right) + d\left(\frac{n}{b^2}\right)\right] + ad\left(\frac{n}{b}\right) + d(n) = aT^3\left(\frac{n}{b^3}\right) + a^2d\left(\frac{n}{b^2}\right) + ad\left(\frac{n}{b}\right) + d \\
&= \cdots\cdot \\
&= a^iT\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j d\left(\frac{n}{b^j}\right)
\end{aligned}$$

*Análisis y Diseño de Algoritmos* *Análisis*-25

## Solución General, cont.

$$T(n) = a^iT\left(\frac{n}{b^i}\right) + \sum_{j=0}^{i-1} a^j d\left(\frac{n}{b^j}\right)$$

- Suponiendo que $n = b^k$, se puede usar el hecho que

$$T\left(\frac{n}{b^k}\right) = T(1) = 1$$

- Haciendo $i = k$, se obtiene

$$T(n) = a^k + \sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right)$$

  - Ya que $k = \log_b n$ , entonces,

$$a^k = a^{\log_b n} = n^{\log_b a}$$

  - Por lo tanto,

$$T(n) = n^{\log_b a} + \sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right)$$

*Análisis y Diseño de Algoritmos* *Análisis*-26

## Solución General, cont.

$$T(n) = n^{\log_b a} + \sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right)$$

- Cuando $a$ es grande, es decir, se divide el problema en más subproblemas, el exponente es mayor
- Cuando $b$ es mayor, es decir, el tamaño de cada subproblema es menor, el exponente será menor.
- El término $a^k$ o $n^{\log_b a}$ se le conoce como la <u>*solución homogénea*</u>. Esta es la solución exacta cuando $d(n)$, la función conductora, es 0 para todos los $n$.
  - La solución homogénea representa el costo de resolver todos los subproblemas cuando ellos se combinan sin costo alguno.

*Análisis y Diseño de Algoritmos* *Análisis*-27

## Solución General, cont.

$$T(n) = n^{\log_b a} + \sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right)$$

- El término $\sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right)$ representa el costo de crear los subproblemas y combinar sus resultados, éste se denomina la <u>*solución particular*</u>.
- Si la solución homogénea es mayor que la función conductora, entonces, la solución particular tiene el mismo índice de crecimiento que la solución homogénea.

*Análisis y Diseño de Algoritmos* *Análisis*-28

## Solución General, cont.

$$\boxed{T(n) = n^{\log_b a} + \sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right)}$$

- Si la función conductora crece más rápido que la solución homogénea por algún factor $n^{\varepsilon}$, para algún $\varepsilon > 0$, entonces la solución particular tiene el mismo índice de crecimiento que la función conductora.

- Si la función conductora tiene el mismo índice de crecimiento que la solución homogénea, o crece más rápido acotada por $\log^k n$ para algún entero $k$, entonces, la solución particular crece en el orden de $\log n$ veces la función conductora.

*Análisis y Diseño de Algoritmos* *Análisis*-29

## Solución General, cont.

$$\boxed{T(n) = n^{\log_b a} + \sum_{j=0}^{k-1} a^j d\left(b^{k-j}\right)}$$

- Es importante reconocer que cuando se busca mejorar un algoritmo se debe tener en cuenta si la solución homogénea es mayor que la función conductora.

- Si la solución homogénea es mayor, no ayuda el encontrar una manera más rápida para dividir y combinar los subproblemas
  - Se debe tratar de dividir un problema en un menor número de subproblemas o en problemas de menor tamaño.

- Si la función conductora excede a la solución homogénea, entonces, se debe tratar de decrementar la función conductora.

*Análisis y Diseño de Algoritmos* *Análisis*-30

## Funciones Multiplicativas

- Se dice que una función $f$ sobre los enteros es *multiplicativa* si $f(x,y) = f(x)f(y)$ para todos los enteros positivos
  - Ejemplo: la función $f(n) = a^n$ es multiplicativa ya que $(xy)^a = x^a y^a$
- Si la función conductora, $d(n)$, es multiplicativa, entonces, $d(b^{k-j}) = (d(b))^{k-j}$
- La solución particular se puede expresar como

$$\sum_{j=0}^{k-1} a^j (d(b))^{k-j} = d(b)^k \sum_{j=0}^{k-1} \left[\frac{a}{d(b)}\right]^j$$

$$= d(b)^k \frac{\left[\frac{a}{d(b)}\right]^k - 1}{\frac{a}{d(b)} - 1}$$

$$= \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}$$

*Análisis y Diseño de Algoritmos* — *Análisis*-31

## Funciones Multiplicativas

$$\sum_{j=0}^{k-1} a^j (d(b))^{k-j} == \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}$$

Si $a > d(b)$, entonces, la solución particular es $O(a^k)$, esto es, $O\left(n^{\log_b a}\right)$, pues $k = \log_b n$.

En este caso la solución particular y homogénea son las mismas, y dependen sólo en $a$ y $b$, y no en la función conductora $d$. Así, para mejorar el tiempo de ejecución se debe decrementar $a$ o incrementar $b$.

*Análisis y Diseño de Algoritmos* — *Análisis*-32

## Funciones Multiplicativas, cont.

$$\sum_{j=0}^{k-1} a^j (d(b))^{k-j} == \frac{a^k - d(b)^k}{\frac{a}{d(b)} - 1}$$

- Si $a < d(b)$, entonces, la solución particular es $O\left(d(b)^k\right)$ o equivalentemente $O\left(n^{\log_b d(b)}\right)$.
  - En este caso, la solución particular excede a la homogénea y para mejorar el tiempo de ejecución, además de considerar $a$ y $b$, se tiene que tomar en cuenta la función conductora, $d(n)$.
- Un caso especial es cuando $d(n) = n^\alpha \Rightarrow d(b) = b^\alpha$.
- Por lo tanto, $\log_b b^b = \alpha$. Así, la solución particular es $O(n^\alpha)$ ú $O(d(n))$.

*Análisis y Diseño de Algoritmos* *Análisis*-33

## Funciones Multiplicativas, cont.

- Si $a = d(b)$,

$$\sum_{j=0}^{k-1} a^j (d(b))^{k-j} = d(b)^k \sum_{j=0}^{k-1} \left[\frac{a}{d(b)}\right]^j$$

$$= d(b)^k \sum_{j=0}^{k-1} 1 = d(b)^k k$$

$$= n^{\log_b d(b)} \log_b n = n^{\log_b a} \log_b n$$

•Ya que $a = d(b)$, la solución particular es $\log_b n$ veces la solución homogénea.

• En el caso en que $d(n) = n^\alpha$, se puede expresar la solución particular como

$$n^{\log_b b^a} \log_b n = n^a \log_b n$$

*Análisis y Diseño de Algoritmos* *Análisis*-34