Algoritmos y Estructuras de Datos
Tema 3: Complejidad Algorítmica

Grado Imat. Escuela ICAI

Juan C. Aguí García

intro

- Un algoritmo debe funcionar en todos los casos del problema que manifiesta resolver.
- Para demostrar que un algoritmo es incorrecto, basta encontrar un ejemplo en el que no funcione.
- **Dominio de definición del algoritmo** $\rightarrow$ conjunto de casos que debe considerarse
- **Principio de invarianza:** Dos implementaciones distintas del mismo algoritmo no diferirán en su eficiencia[1] en más de alguna constante multiplicativa (10 veces más deprisa. . . etc).

COMILLAS ICAI

[1] Aunque no sabemos aún con precisión qué es la eficiencia

Casos Medios y Peor

- Se debe considerar el caso peor de un algoritmo, tamaño y condiciones, para los que el algoritmo requiera más tiempo
  - Hay que analizar el caso peor en procesos donde los tiempos de respuesta sean críticos (centrales nucleares, coche autoconducido, etc.) y para asegurar la terminación del mismo en tiempo razonable.
  - Por ejemplo, el tiempo requerido por el algoritmo de inserción, oscila entre n y n2, en función de los datos aportados al algoritmo
- Hay que evaluar el comportamiento medio de un algoritmo (A veces se denomina valor *amortizado*) donde el coste de operaciones particulares se ha da distribuir sobre varias llamadas al algoritmo.

COMILLAS ICAI

Part I

Notación Asintótica
The big O

ver https://www.youtube.com/watch?v=Q_1M2JaijjQ

Notación Asintótica

- Permite comparar de forma genérica y abstracta los recursos demandados por un algoritmo en función del volumen de datos a procesar
  - Independientemente de los datos concretos a procesar
  - Independientemente del lenguaje de programación
  - Independientemente del ordenador que lo ejecute
- Considerar dos recursos básicos:
  - Tiempo de ejecución
  - Consumo de memoria[2]
- La medida del volumen de los datos depende del problema:
  - Algoritmos de ordenación; número de registros a ordenar
  - Árboles binarios de búsqueda: número de nodos
  - Grafos: número de vértices, número de conexiones ? La suma ?

[2] Existen algoritmos *"out of core"* que funcionan con los datos en memoria y en almacenamiento externo

Cotas Asintóticas

- **Cota asintótica Superior (The big O).**
  Se dice que $f(n) = O(g(n))$ si:

$$\exists c, N_0 \mid \forall N \geq N_0 \;\; 0 \leq f(n) \leq c * g(N)$$

- **Cota asintótica Inferior (The big Ω)**
  Se dice que $f(n) = \Omega(g(n))$ si:

$$\exists c, N_0 \mid \forall N \geq N_0 \;\; 0 \leq c * g(n) \leq f(n)$$

- **Cota asintótica Superior e Inferior (The big Θ)**
  Se dice que $f(n) = \Theta(g(n))$ si:

$$\exists c_1, c_2, N_0 \mid \forall N \geq N_0 \;\; 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n)$$

Where N is the size indicator of the algorithm, and $N_0, c, c_1, c_2 > 0$. Si se dan las cotas inferior y superior, se puede derivar la existencia de c_1 y c_2 que demuestra la Cota Θ

Cotas Asintóticas: Visual

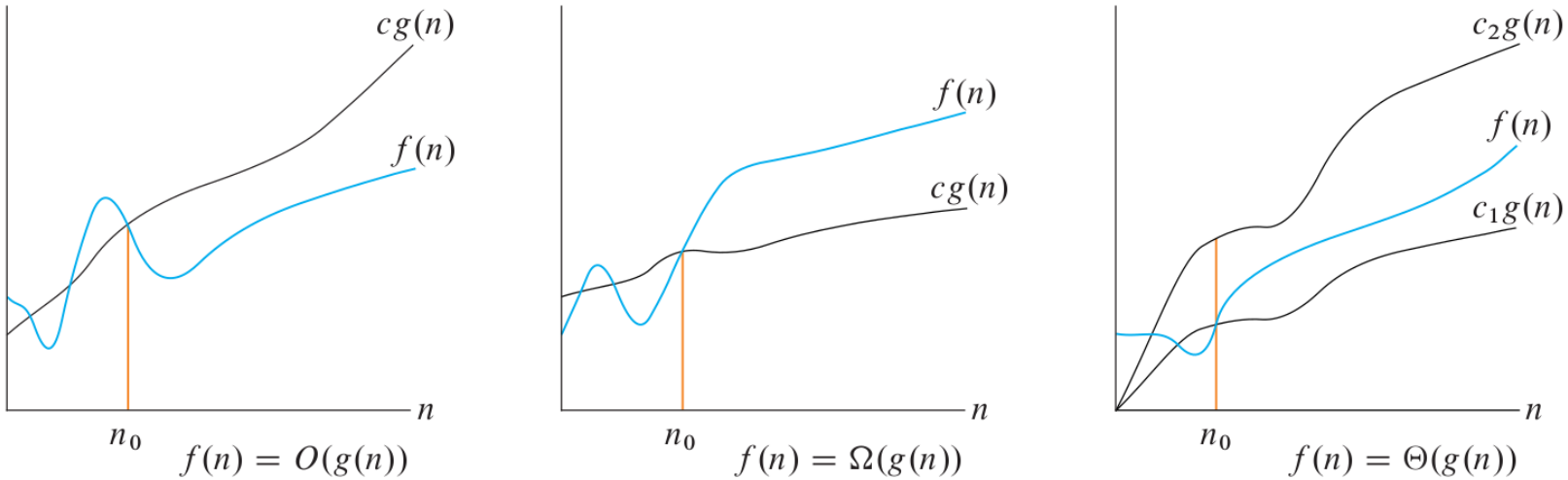


Figure: Visualización Cotas Asintóticas ©Cormen et al.

- Sólo nos preocupa el **"comportamiento asíntotico"** esto es, para n grandes. En valores pequeños de n, esto es algoritmos ejecutando sobre datos pequeños, pueden ocurrir cosas raras, que no afectan a nuestras conclusiones.
- Las cotas Superior (**O**), y la cota Inferior (**Ω**) no presuponen una evolución **paralela**, pudiendo crecer, sin embargo, $f(n)$ y $g(n)$ **a ritmos muy dispares.**
- Sólo la cota Inferior y Superior asegura un orden de crecimiento asintótico parejo entre el coste del algoritmo $f(n)$ y la función $g(n)$ a expensas de constantes fijas.

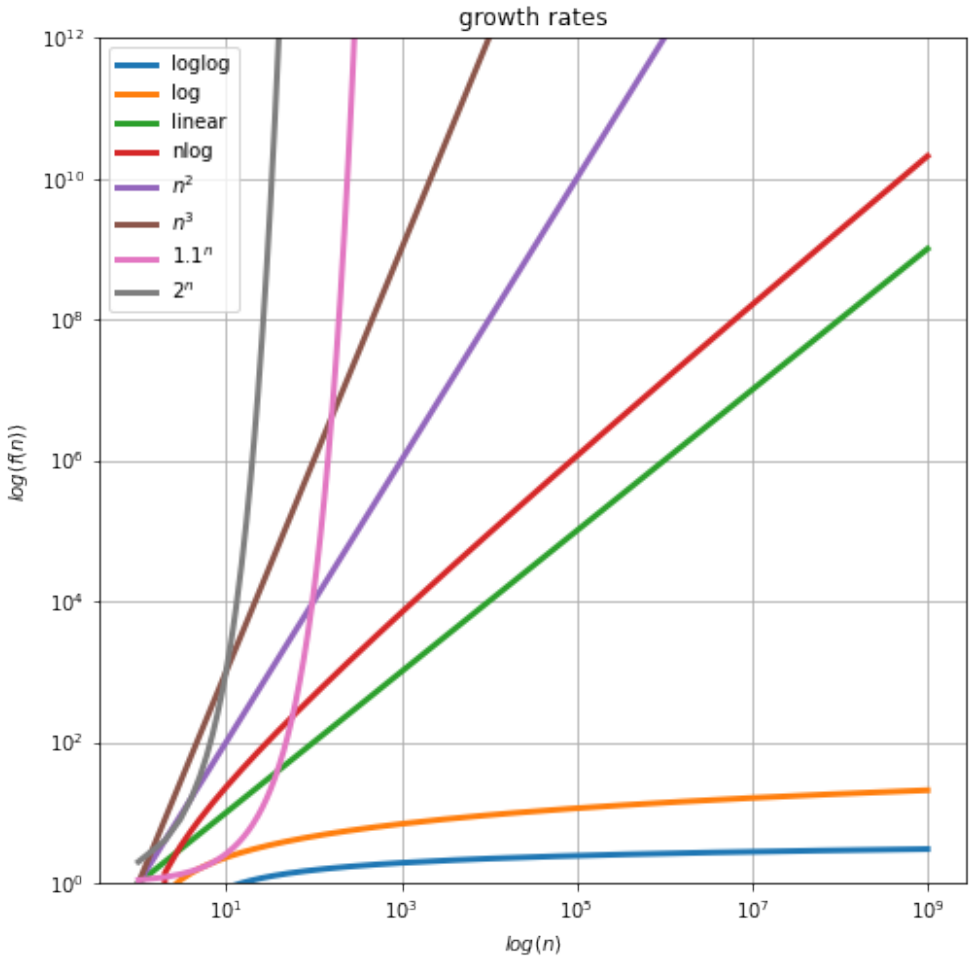
Rangos de Crecimiento

A partir de un n suficientemente grande se cumple que:

$$\log\log n \; < \; \log n \; < \; n \; < \; n * \log n \; < \; n^2 \; < \; n^3 \; < \; c^n$$

Diferentes Rangos:

1. Logarithmic
   (log, loglog)
2. Polynomial
   (n^a; $a > 0$)
3. Pol-logarithmic
   ($n^a * log^b(n)$; $a, b > 0$)
4. Exponential
   (c^n; $c > 1$)



COMILLAS ICAI

El Orden de Cremiento es importante

Para un tamaño realista de datos, el orden de crecimiento de un algoritmo marca la diferncia y su usabilidad práctica, o no
Hay:

- 10^{80} átomos en el universo.
- $4,4 \times 10^{17}$ segundos, desde el BigBang

No es difícil pensar algoritmos que sobrepasan estos límites. Si para $n = 1$ el tiempo es de $10\mu s$, para $n = 1,000,000$ los tiempos de cálculo esperables son:

logarithmic	60 ms
linear	10 segundos
square	115 días
cubic	3,174 siglos
exponential	mayor que bigbang

Propiedades Notación Asintótica

$$c * \mathbf{O}(f(N)) = \mathbf{O}(f(N)) \qquad c > 0 \ cte \qquad (1)$$

$$\mathbf{O}\left(f(N) + g(N)\right) = max\left(\mathbf{O}(f(N)), \mathbf{O}(g(N))\right) \qquad (2)$$

$$\mathbf{O}\left(f(N)\right) + \mathbf{O}(g(N)) = \mathbf{O}\left(f(N) + g(N)\right) \qquad (3)$$

$$\mathbf{O}\left(f(N)\right) \times \mathbf{O}\left(g(N)\right) = \mathbf{O}\left(f(N) \times g(N)\right) \qquad (4)$$

$$\mathbf{O}\big(\mathbf{O}f(N)\big) = \mathbf{O}\big(f(N)\big) \qquad (5)$$

(1) Órdenes son independientes de constantes

(2) Órdenes de suma de dos funciones es el máximo de los respectivos órdenes

(3) Suma de órdenes igual al orden de la suma de las funciones[3]

(4) Orden del producto igual al producto de los respectivos órdenes

(5) Orden del Orden equivale al Orden

COMILLAS ICAI

[3] ???

Propiedades Notación Asintótica: Ejemplos

$$3 * \mathbf{O}(N^3) = \mathbf{O}(N^3)$$
$$\mathbf{O}(N^2 + N) = max(\mathbf{O}(N^2), \mathbf{O}(N)) = \mathbf{O}(N^2)$$
$$\mathbf{O}(N) + \mathbf{O}(N) = \mathbf{O}(N + N) = 2 * \mathbf{O}(\mathbf{N}) = \mathbf{O}(\mathbf{N})$$
$$\mathbf{O}(N) * \mathbf{O}(N) = \mathbf{O}(N * N) = O(N^2)$$
$$\mathbf{O}(\mathbf{O}(N)) = \mathbf{O}(N)$$
$$\mathbf{O}(N + N \log N) = \mathbf{O}(N \log N)$$
$$\mathbf{O}(N^2 + 4 * N \log N) = \mathbf{O}(N^2)$$
$$\mathbf{O}(100 + N/2 + 4 * N \log N) = \mathbf{O}(N \log N)$$

Part II

Análisis de la Complejidad

Análisis de la complejidad: 1

Cálculo complejidad de un algoritmo:

- Evaluar la complejidad de las operaciones que lo componen
- Obtener una expresión de la complejidad genérica que indique cómo aumenta la misma con el volumen de los datos a procesar
  - Normalmente se utiliza Notación asintótica
  - Se suele distinguir distintos casos
- Dos tipos de análisis diferentes:
  - Algoritmos no recursivos
  - Algoritmos recursivos (Cálculo de recurrencias)[4]

[4] Ver cap 4 de Cormen's Book

Complejidad de tiempo usuales

$\mathbf{O}(1)$: complejidad constante

Ejecución invariante ante el cambio en número de datos.
(Situación ideal !)

$\mathbf{O}(\log(N))$: complejidad Logarítmica

Ejecución escala con el logaritmo del número de datos.

- Es una situación muy deseable correspondiéndose con muchos de los algoritmos óptimos.
- El orden es independiente de la base del logaritmo utilizado:
  $\log_x(N) = \log_x(b) * \log_b(N))$

Aparece en casos de iteración o recursión no estructura, o bucles cuyo índice evoluciona de forma multiplicativa

Complejidad de tiempo usuales (2)

$\mathbf{O}(N)$: complejidad Lineal

Ejecución escala linealmente del número de datos.

- Es una situación muy frecuente

Aparece en bucle simple cuando la complejidad de las operaciones interiores es constante o en algoritmos con recursión estructural (iteraciones con evolución no multiplicativa de los índices)

$\mathbf{O}(N \log(N))$: complejidad Lineal-Logarítmica

Ejecución escala como el producto del tamaño de los datos y su logaritmo

- No es una situación infrecuente

Aparece en base a la combinación de una iteración lineal compuesta con un subalgoritmo de complejidad logarítmica en base a iteración o recursión no estructural

COMILLAS ICAI

Complejidad de tiempo Polinómicas

$\mathbf{O}(N^2)$: complejidad cuadrática

Ejecución escala con el cuadrado del número de datos.
Aparece en bucles anidados con iteraciones simples. Ej. Proceso de imágenes

$\mathbf{O}(N^3)$: complejidad Cúbica

Ejecución escala con el cubo del número de datos.
Aparece en bucles anidados con iteraciones simples. Ej. Multiplicación de Matrices

$\mathbf{O}(N^k)$: complejidad polinómica

Ejecución escala con la potencia del número de datos.
Bucles anidados en general.

COMILLAS ICAI

Complejidad de tiempo exponencial

$\mathbf{O}(c^N)$ $c > 1$: complejidad exponencial

Ejecución escala como una constante elevado a N
Aparece en algoritmos del tipo **Divide y Vencerás** del tipo
$T(N) = a * T(N/b) + f(N)^a$
o procesos de forma recursiva del tipo
$T(N) = a * T(N-1) + b * T(N-2) + ... + f(N)$

Son algoritmos muy poco eficientes que por su explosión exponencial, *"escalan mal"*
Ejemplos: Cálculo de números de Fibonacci, Problemas de planificación temporal.

---

[a] este tipo de algoritmo también puede dar lugar a complejidades del tipo polinómico, o pol-logarítmico

Evaluando el Orden de complejidad de un algoritmo

En un algoritmo, es un conjunto de operaciones básicas, asignaciones, condicionales de Flujo, iteraciones, y llamadas a otras funciones y llamadas recursivas

- **Operaciones básicas (+/-,*,/, operaciones matemáticas, etc...)** . En tanto operen sobre datos atómicos (no vectores) son rápidas y esencialmente invariante con los datos
  Tomaremos $\Rightarrow \mathbf{O}(1)$
- **Asignaciones de variables** $(=)$. Su coste sólo depende del tipo de datos y normalmente sólo involucra la asignación de punteros a memoria.
  Tomaremos $\Rightarrow \mathbf{O}(1)$
- **Secuencias de Operaciones** El coste de ejecución es la suma de los costes de cada operación. Aplicando los criterios de la suma:
  $\Rightarrow \mathbf{O}[I_1, I_2, ...I_m] = max[\mathbf{O}(I_1), \mathbf{O}(I_2), ..., \mathbf{O}(I_m)]$

Evaluando el Orden de complejidad de un algoritmo II

- **Llamadas a funciones** T(llamada) = T(ejecutar función)
  Se puede despreciar el tiempo de paso de argumentos si se pasan por referencia
- **Instrucciones Condicionales**. Hay que considerar el tiempo de evaluacion de las condiciones (normalmente $\sim \mathbf{O}(1)$) más el tiempo de cada posible rama de ejecución

$$\mathbf{O}\left(\begin{array}{|l|}\hline \textbf{if Cond:} \\ \qquad\qquad B1 \\ \text{else:} \\ \qquad\qquad B2 \\ \textbf{End if} \\ \hline \end{array}\right) \Rightarrow \quad max[\mathbf{O}(Cond), \mathbf{O}(B1), \mathbf{O}(B2)]$$

Evaluando el Orden de complejidad de un algoritmo III

- **Instrucciones iteración:** `FOR: n_iter` o `WHILE: n_iter` con diversos niveles de anidamiento, y aplicado a un bloque de instrucciones depende del iterador que a su vez, depende de los datos (peor caso y mejor caso)
  Tomaremos $\Rightarrow \mathbf{O}(N^k * \log^j N)$ donde k es el número de anidamientos con iteración estructural y j el número de anidamientos con recursion no estructural
- **Llamadas a funciones:** T(llamada) = T(ejecutar función)
  Se puede despreciar el tiempo de paso de argumentos si se pasan por referencia
- **Llamadas Recursivas:** Es necesario evaluar el **Branching Factor** y la **profundidad del árbol de recursion** para evaluar el Orden del Algoritmo (lo veremos más adelante)

Complejidad Polinómica: Ejemplo Multiplicación de Matrices

Sea A y B matrices de dimensiones $n \times m$ y $m \times k$ respectivamente:
Entonces al algoritmo básico de multiplicación: $C = A \times B$ se define como:

$$c_{i,j} = \sum_{s=0}^{s=m-1} a_{i,s} * b_{s,j}$$

Analiza:

- Dónde están las operaciones costosas ??
- Cuántas veces se ejecutan?

La complejidad es $n \times m \times k \sim n^3$[a]

[a] Existen algoritmos que llegan a complejidad $\sim \mathbf{O}(n^{2,7})$ Siendo una operación clave, cualquier mejora es importante !!

```
function MatMult(A,B)
    n, m ←Dimensions of A
    p, k ←Dimensions of B
    if  p != m then
        RaiseError (Matrices No congruentes)
    else
        C ← MatAlloc(n,k)
        for i in range(n) do
            for j in range(k) do
                c_{i,j} ← 0
                for s in range(p) do
                    c_{i,j} ← c_{i,j} + a_{i,s} * bs, j
                end for
            end for
        end for
    end if
    return C
end function
```

Complejidad Logarítmica: Ejemplo Multiplicación a la Russe

Multiplicacion a la rusa

consiste en multiplicar sucesivamente por 2 el multiplicando y dividir por 2 el multiplicador hasta que el multiplicador tome el valor 1. En el proceso, se suman todos los multiplicandos correspondientes a los multiplicadores impares. Dicha suma es el producto de los dos números

Cuál es la complejidad ?

- Cuántos Loops hay ?
  - → Uno, el `While`
- Cuántas veces se ejecuta ?
  - → la mitad de $log_2(m)$
- Cual es el coste de cada ejecución ?
  - → Suma de un número de $log_2(n)$ digitos más dos bit-shifts

⇒ $\log_2(n) \times \log_2(m)$

function RusMult(n,m)
  $res \leftarrow 0$
  while $m \geq 1$ **do**:
    if m is odd **then**
      $res \leftarrow res + n$
      $m \leftarrow \lfloor m/2 \rfloor$ ▷ bit Shift right
      $n \leftarrow n + n$ ▷ bit Shift Left
    end if
  end while
  return res
end function

Complejidad Logarítmica: Binary Search

Buscando en una lista ordenada por el método de la bisección.

Decide en qué mitad está, comparando con el pivote central, y llama de forma recursiva sobre la mitad "candidata"

Search for 0.69 Using Binary Search

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
0,06	0,07	0,10	0,16	0,25	0,36	0,40	0,43	0,45	0,47	0,48	0,53	0,54	0,55	0,58	0,59	0,65	0,69	0,73	0,76	0,79	0,88	0,88	0,90	0,91	0,96
												»	0	1	2	3	4	5	6	7	8	9	10	11	12
												13	0,55	0,58	0,59	0,65	0,69	0,73	0,76	0,79	0,88	0,88	0,90	0,91	0,96
												13	0	1	2	3	4	5	«						
													0,55	0,58	0,59	0,65	0,69	0,73							
															»	0	1	2							
															13	0,65	0,69	0,73							
																	'==								
																	17								

- Cuál es el número máximo de llamadas recursivas, "o mitades" ??
  - → $log_2(N)$
- Cuál es el coste en cada nivel ??
  - → Una comparación y una división para hacer la nueva mitad $\mathbf{O}(1)(N)$
- Cuál es la complejidad resultante ?
  - → $\log_2(N) * \mathbf{O}(1) = \mathbf{O}(\log N)$

Binary Search

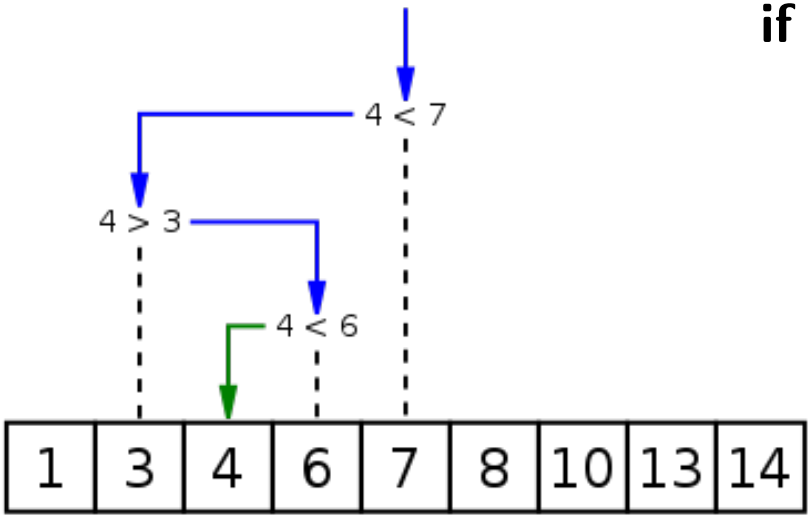


Figure: Visualización Binary Search ©Wikipedia

```
function BinSearch(array, low, high,x)
    if high ≥ low then
        mid ← ⌊(high + low)/2⌋
        if array[mid] = x then
            return mid
        else if array[mid] > x then
            return BinSearch(arr, low, mid - 1, x)
        else
            return BinSearch(arr, mid+1,high, x)
        end if
    else
        return -1
    end if
end function
```

Complejidad equivalente al número de saltos $\sim log(N)$ contenidos...

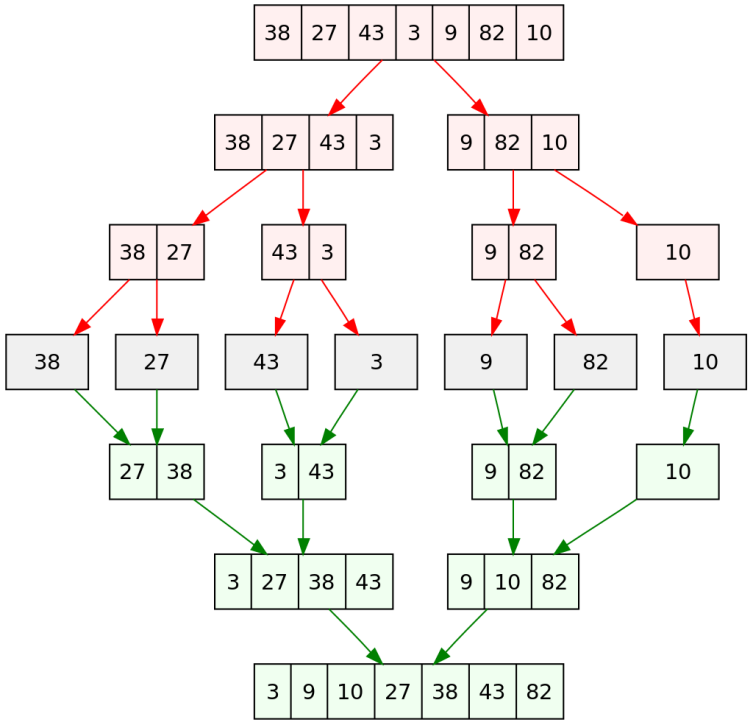
Complejidad Lineal-Logarítmica: Ejemplo MergeSort

MergeSort

Divide la secuencia en dos, sobre los que invoca el Merge_sort de forma recursiva, para luego realizar un merge de la secuencias previamente ordenadas

Cuál es la complejidad ?

- Fase de división
  - $\rightarrow$ $log_2(N) \times \mathbf{O}(1)$
- Cuántos niveles hay el el arbol durante el merge ?
  - $\rightarrow$ $log_2(N)$
- Cuál es el coste de cada uno de ellos
  - $\rightarrow$ $\mathbf{O}(N)$
- Cuál es la complejidad resultante ?
  - $\rightarrow$ $\log_2(N) * (\mathbf{O}(1) + \mathbf{O}(N))$
  - $\rightarrow$ $\mathbf{O}(N * \log N)$



Complejidad de tiempo Exponenciales

$\mathbf{O}(2^N)$: complejidad exponencial

Explosión combinatoria

Aparece en algoritmos con recursión en que hay dos llamadas o más por cada dato de entrada

La base es frecuentemente 2, pero puede ser cualquier constante > 1

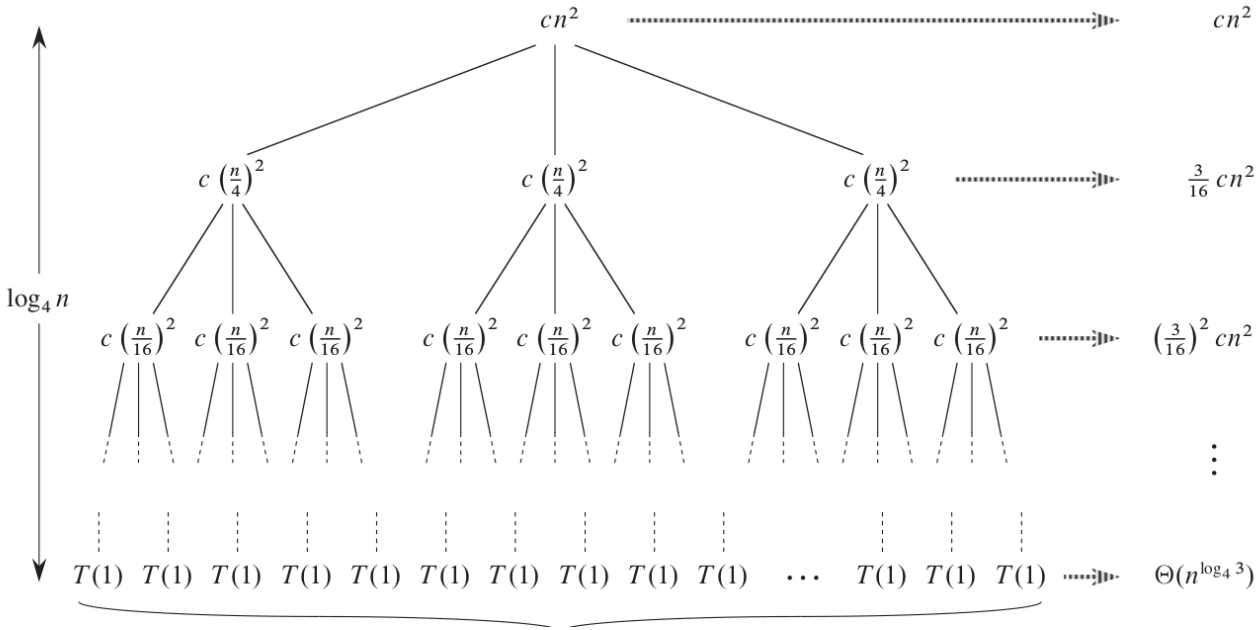
- Cada llamada genera, recursivamente 2 llamadas.
- La profundidad del arbol de recursión es n
- $\Rightarrow$ **La complejidad es por tanto** $\sim \mathbf{O}(2^n)$

```
function FIBONACCI(n)
    if n ≤ 1 then
        return n
    else
        return FIBONACCI(n-1) +
               FIBONACCI(n-2)
    end if
end function
```

Complejidad de Algoritmos Recursivos: Método del Árbol de Recursión

1. Dibuja el árbol de recursion definido por el algoritmo
2. En el árbol de recursión, cada nodo representa el coste de un subproblema aislado dentro del árbol. Evalúa el coste de ejecución del nodo.
3. Sumando los costes de cada nodo, en cada nivel de recursión y evaluando la profundidad del árbol, podemos estimar el coste total de la recursión

Tomemos una recursión del tipo: $T(N) = 3T(\lfloor n/4 \rfloor) + \Theta(n^2)$ y veamos el árbol de recursión:



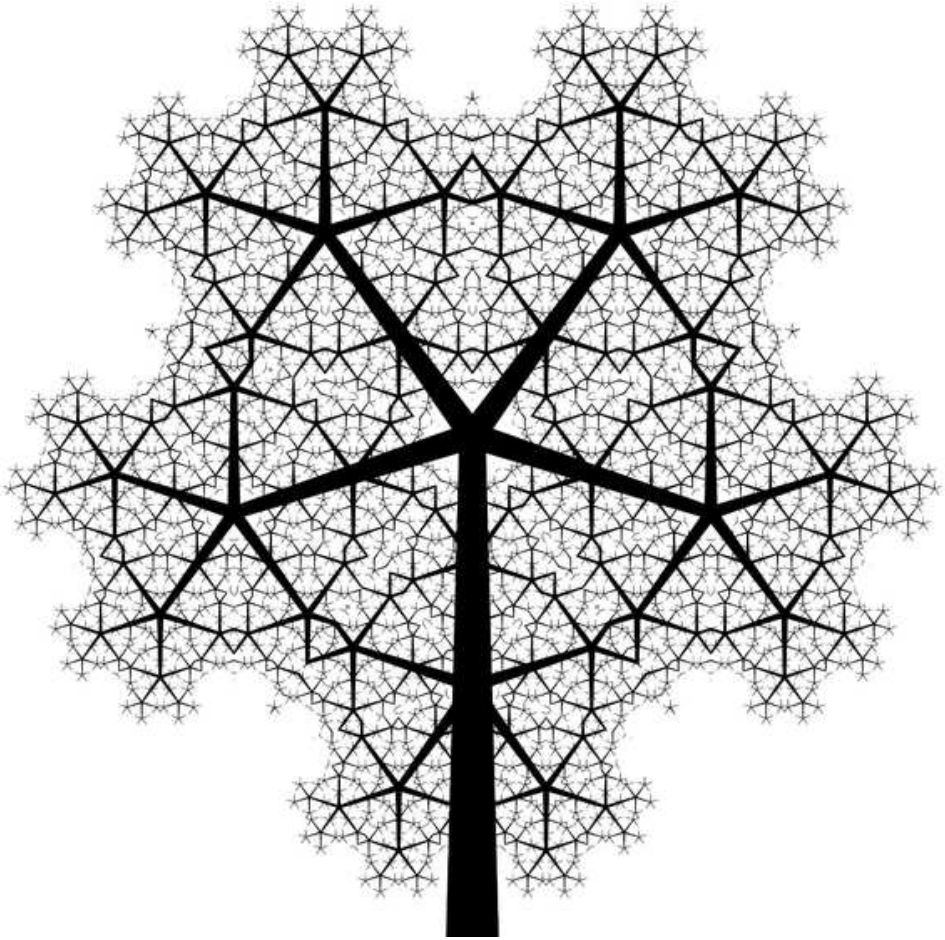
Complejidad de Algoritmos Recursivos: Método del Árbol de Recursión (cont)

- La profundidad del árbol es de $log_4 n$ en base al **branching factor**
- En el nivel final, hay $3^{log_4 n} = n^{log_4 3}$ nodos con coste unitario $\mathbf{O}(1)$
- En los niveles intermedios los costes son de $cn^2 + 3/16cn^2 + (3/16)^2 cn^2 + ... + (3/16)^{\log_4(n-1)} cn^2$

En consecuencia

$$\begin{aligned} T(N) &= \sum_{i=0}^{\log_4 n-1} (3/16)^i cn^2 + \Theta(n^{log_4 3}) \\ &< \sum_{i=0}^{\infty} (3/16)^i cn^2 + \Theta(n^{log_4 3}) \\ &= \frac{1}{1-3/16} cn^2 + \Theta(n^{log_4 3}) \\ &= \frac{16}{3} cn^2 + \Theta(n^{log_4 3}) = \mathbf{O}(n^2) \end{aligned}$$

Eof Tema 3: Gracias !



Nature is highly fractal and recursive...

COMILLAS ICAI