# Algoritmos y Complejidad

## Introducción

Pablo R. Fillottrani

Depto. Ciencias e Ingeniería de la Computación
Universidad Nacional del Sur

primer semestre 2024

# Introducción

- un **problema computacional** consiste en una caracterización de un conjunto de datos de entrada, junto con una especificación de la salida deseada en base a cada entrada
- un **algoritmo** es una secuencia bien determinada de acciones elementales que transforma los datos de entrada en datos de salida con el objetivo de resolver un problema computacional
  - para cada algoritmo es necesario aclarar cuáles son las operaciones elementales y cómo están representados los datos de entrada y de salida
  - en general se representarán los algoritmos por medio de un pseudocódigo informal
- un **programa** consiste en la especificación formal de un algoritmo por medio de un lenguaje de programación, de forma que pueda ser ejecutado por una computadora

- la **algoritmia** es el estudio sistemático del diseño y análisis de algoritmos.

**Problemas en la Algoritmia**

- **correctitud**
  - finalización
  - ¿resuelve el problema?
- **eficiencia**
  - cantidad de recursos
  - ¿se puede mejorar?
- **aproximación**
  - cálculo numérico
  - ¿se puede aproximar mejor?

- un problema computacional tiene una o más **instancias**, valores particulares para los datos de entrada, sobre las cuales se puede ejecutar un algoritmo para resolver el problema
- ejemplo: el problema computacional MULTIPLICACIÓN DE DOS NÚMEROS ENTEROS tiene por ejemplo las siguientes instancias: multiplicar 345 por 4653, multiplicar 2637 por 10000, multiplicar $-32341$ por 1, etc.
- un problema computacional **abarca** a otro problema computacional si las instancias del segundo pueden ser resueltas como instancias del primero en forma directa.
- ejemplo: MULTIPLICACIÓN DE UN ENTERO POR 352 es un problema computacional que es abarcado por el problema MULTIPLICACIÓN DE DOS NÚMEROS ENTEROS.

- es claro que para cada algoritmo la cantidad de recursos (tiempo, memoria) utilizados depende fuertemente de los datos de entrada. En general, la cantidad de recursos crece a medida que crece el tamaño de la entrada
- el análisis de esta cantidad de recursos no es viable de ser realizado instancia por instancia
- se introducen las funciones de cantidad de recursos en base al **tamaño de la entrada**. Este tamaño puede ser la cantidad de dígitos para un número, la cantidad de elementos para un arreglo, la cantidad de caracteres de una cadena, etc.
- en ocasiones es útil definir el tamaño de la entrada en base a dos o más magnitudes. Por ejemplo, para un grafo es frecuente utilizar la cantidad de nodos y la de arcos

- dado un algoritmo $A$, el **tiempo de ejecución** $t_A(n)$ de $A$ es la cantidad de pasos, operaciones o acciones elementales que debe realizar el algoritmo al ser ejecutado en una instancia de tamaño $n$
- el **espacio** $e_A(n)$ de $A$ es la cantidad de datos elementales que el algoritmo necesita al ser ejecutado en una instancia de tamaño $n$, sin contar la representación de la entrada ni de la salida
- estas definiciones son ambiguas (¿porqué?)

- ambigüedades de la definición de tiempo de ejecución:
  - No está claramente especificado cuáles son las operaciones o los datos elementales. Este punto quedará determinado en cada análisis en particular, dependiendo del dominio de aplicación
  - Dado que puede haber varias instancias de tamaño $n$, no está claro cuál de ellas es la que se tiene en cuenta para determinar la cantidad de recursos necesaria
- para resolver este último punto se definen distintos tipos de análisis de algoritmos

- tipos de análisis de algoritmos:
  - **análisis en el peor caso:** se considera el máximo entre las cantidades de recursos insumidas por todas las instancias de tamaño $n$
  - **análisis caso promedio:** se considera el promedio de las cantidades de recursos insumidas por todas las instancias de tamaño $n$
  - **análisis probabilístico:** se considera la cantidad de recursos de cada instancia de tamaño $n$ pesado por su probabilidad de ser ejecutada
  - **análisis en el mejor caso:** se considera el mínimo entre las cantidades de recursos insumidas por todas las instancias de tamaño $n$

- nos concentraremos en general a analizar el peor caso, debido a que
    - constituye una cota superior al total de los recursos insumidos por el algoritmo. Conocerla nos asegura que no se superará esa cantidad
    - para muchos algoritmos, el peor caso es el que ocurre más seguido
    - debido al uso de la notación asintótica, el caso promedio o probabilístico es muchas veces el mismo que el peor caso
    - no se necesita conocer la distribución de probabilidades para todas las instancias de un mismo tamaño, como sería necesario en el análisis probabilístico
    - en la mayor parte de los casos, es más fácil de analizar matemáticamente

- se considerará entonces que un algoritmo es **más eficiente** que otro para resolver el mismo problema si su tiempo de ejecución (o espacio) en el peor caso tiene un crecimiento menor

# MULTIPLICACIÓN DE DOS NÚMEROS ENTEROS

## Algoritmos:

### Clásico

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | 9 | 8 | 1 |
| × | | | 1 | 2 | 3 | 4 |
| | | | 3 | 9 | 2 | 4 |
| | | 2 | 9 | 4 | 3 | |
| | 1 | 9 | 6 | 2 | | |
| | 9 | 8 | 1 | | | |
| 1 | 2 | 1 | 0 | 5 | 5 | 4 |

### A la inglesa

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | 9 | 8 | 1 |
| × | | | 1 | 2 | 3 | 4 |
| | 9 | 8 | 1 | | | |
| | 1 | 9 | 6 | 2 | | |
| | | 2 | 9 | 4 | 3 | |
| | | | 3 | 9 | 2 | 4 |
| 1 | 2 | 1 | 0 | 5 | 5 | 4 |

### A la rusa

| | | |
|---|---|---|
| 981 | 1234 | 1234 |
| 490 | 2468 | |
| 245 | 4936 | 4936 |
| 122 | 9872 | |
| 61 | 19744 | 19744 |
| 30 | 39488 | |
| 15 | 78976 | 78976 |
| 7 | 157952 | 157952 |
| 3 | 315904 | 315904 |
| 1 | 631808 | 631808 |
| | | 1210554 |

# ORDENAR UN ARREGLO

Algoritmo: Ordenamiento por Inserción

```
FOR j ::= 2 TO n
    x ::= A[j]
    i ::= j-1
    WHILE i> 0 and A[i]>x
        A[i+1] ::= A[i]
        i ::= i-1
    ENDWHILE
    A[i+1] ::= x
ENDFOR
```

| costo | veces |
|---|---|
| $c_1$ | $n$ |
| $c_2$ | $n-1$ |
| $c_3$ | $n-1$ |
| $c_4$ | $\sum_{j=1}^{n-1} t_j$ |
| $c_5$ | $\sum_{j=1}^{n-1}(t_j-1)$ |
| $c_6$ | $\sum_{j=1}^{n-1}(t_j-1)$ |
| $c_8$ | $n-1$ |

# ORDENAR UN ARREGLO

Algoritmo: Ordenamiento por Inserción

- Costo de la ejecución del algoritmo:

$$
\begin{aligned}
T_I(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=1}^{n-1} t_j + c_5 \sum_{j=1}^{n-1} (t_j - 1) + \\
&\quad + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_8(n-1) \\
&= (c_1 + c_2 + c_3 + c_8)n - (c_2 + c_3 + c_8) + (c_4 + c_5 + c_6) \sum_{j=1}^{n-1} t_j - (c_5 + c_6) \sum_{j=1}^{n-1} 1
\end{aligned}
$$

- demasiado complicado para ser útil. No es posible simplificar
- aplicando la notación asintótica $\Theta(\cdot)$ podemos decir que $T_I(n) \in \Theta(n^2)$ y comparar fácilmente con otros algoritmos

# Función característica de un ciclo

- la función característica de un ciclo es una función, en los datos del ciclo, que es decreciente y que siempre tiene valor positivo o O
- permite acotar la cantidad de iteraciones de ese ciclo y de esa forma obtener una cota superior al tiempo de ejecución
- también asegura que el ciclo termina

# MÁXIMO COMÚN DIVISOR

Algoritmo: Algoritmo de Euclides

```
Function EUCLIDES(m,n)
   WHILE m>0
      temp ::= m
      m ::= n mod m
      n ::= temp
   ENDWHILE
   RETURN n
```

- se puede observar las propiedades:
  - $n_i = m_{i-1}$, $m_i = n_{i-1} \bmod m_{i-1}$ siempre que $i \geq 1$
  - $n_i \geq m_i$ siempre que $i > 1$
  - para todo $n, m$ tal que $n \geq m$ vale $n \bmod m < n/2$
  - $n_i = m_{i-1} = n_{i-2} \bmod m_{i-2} < n_{i-2}/2$ si $i > 2$
- luego, en dos iteraciones $n_0$ se reduce a menos de la mitad; en cuatro a menos del cuarto; etc. Como $m_i > 0$ entonces no puede haber más de $2 \log_2 n_0$ iteraciones.

# Algoritmo de Euclides

- la función característica es entonces de $O(\log_2 n)$, pero esto es solo una cota en la cantidad de iteraciones
- para obtener una cota en tiempo debemos:
  1. multiplicar esa cota por el tiempo de ejecución de cada bucle, en este caso $O(1)$
  2. expresar la función en base a la longitud de la entrada (cantidad de dígitos de $n$) y no de su valor ($n$)

# Algoritmo de Euclides

- se $d$ la cantidad de dígitos de $n$. Si $n$ esta en notación decimal, $d = \lceil \log_{10} n \rceil$
- esto implica que $n \in \Theta(10^d)$, y que por lo tanto $T_{Euc} \in \Theta(\log_2 10^d) = O(d * \log_2 10) = O(d)$
- Euclides es un algoritmo lineal Ejercicio: comparar con el algoritmo que se enseña en la primaria
- tener cuidado en todos los algoritmos numéricos hay que expresar el análisis del tiempo y del espacio en función de la cantidad de dígitos y no de su valor

# Sentencias barómetro

- esta técnica se usa a fin de simplificar el análisis o de obtener cotas más precisas de los tiempos de ejecución de ciclos
- se identifican determinadas sentencias clave dentro de los ciclos, que se denominan barómetros, y se cuentan cuántas veces se ejecutan estas sentencias en el total de las iteraciones, no en cada iteración individual

# ORDENAR UN ARREGLO

Algoritmo: Ordenamiento por Cubículos (enteros hasta $s$)

```
array U[1..s] ::= 0                        Θ(n)
FOR i ::= 1 TO n
   k ::= T[i]; U[k]++                      Θ(1)
ENDFOR
i ::= 0
FOR k ::= 1 TO s
   WHILE U[k] != 0                         barómetro
     T[i++] ::= k; U[k]- -
   ENDWHILE
ENDFOR
```

- el barómetro se ejecuta $U[k]_0 + 1$ veces por cada $k$
- luego el tiempo total es: $\sum_{k=1}^{s}(U[k]_0 + 1)\Theta(1)$
- y vale

$$\begin{aligned} T(n) &= \Theta(n) + \Theta(1) + \sum_{k=1}^{s}(U[k]_0 + 1)\Theta(1) = \\ &= \Theta(n) + \sum_{k=1}^{s} U[k]_0\Theta(1) + \sum_{k=1}^{s}\Theta(1) \\ &= \Theta(n) + \Theta(n) + \Theta(s) \in \Theta(\max(n, s)) \end{aligned}$$

- el problema de este algoritmo es el límite máximo de los números a utilizar, y el espacio de memoria auxiliar

# NÚMEROS DE FIBONACCI

$$F_n = \begin{cases} i & \text{si } i = 0 \text{ o } i = 1 \\ F_{n-1} + F_{n-2} & \text{si } i \geq 2 \end{cases}$$

Algoritmo: naïve, recursivo

```
function FIB1(n)
  IF n<2
      return n
  ELSE
      return Fib1(n-1)+Fib1(n-2)
```

# NÚMERO DE FIBONACCI

Algoritmo: naïve, recursivo

- definiendo el tiempo de ejecución del algoritmo anterior, se obtiene la siguiente recurrencia

$$T_{FIB1}(n) = \begin{cases} a & \text{si } n < 2 \\ T_{FIB1}(n-1) + T_{FIB1}(n-2) + b & \text{si } n \geq 2 \end{cases}$$

- asi como está, esta función tampoco es útil para analizar y comparar el algoritmo

# NÚMERO DE FIBONACCI

Algoritmo: primer algoritmo iterativo (sumas y restas son operaciones elementales)

| Código | costo | veces |
|---|---|---|
| `Function FIB2(n)` | | |
| `i ::= 1; j ::= 0` | $b$ | $1$ |
| `FOR k ::= 1 TO n` | $c_1$ | $\sum_{k=1}^{n+1} 1$ |
| `j ::= i+j` | $c_2$ | $\sum_{i=1}^{n} 1$ |
| `i ::= j-i` | $c_3$ | $\sum_{i=1}^{n} 1$ |
| `ENDFOR` | | |
| `RETURN j` | $d$ | $1$ |

- calculando el tiempo de ejecución se tiene

$$
\begin{aligned}
T_{FIB2}(n) &= b + c_1 + \sum_{k=1}^{n} (c_1 + c_2 + c_3) + d = \\
&= (b + d) + (c_1 + c_2 + c_3)n \in \Theta(n)
\end{aligned}
$$

- si la suma y la resta no son operaciones elementales (operan sobre números muy grandes) entonces

| | costo | veces |
|---|---|---|
| `Function FIB2(n)` | | |
| `i ::= 1; j ::= 0` | $b$ | $1$ |
| `FOR k ::= 1 TO n` | $c_1$ | $\sum_{k=1}^{n} 1$ |
| `j ::= i+j` | $c_2 * tamaño(j)$ | $\sum_{i=1}^{n} 1$ |
| `i ::= j-i` | $c_3 * tamaño(j)$ | $\sum_{i=1}^{n} 1$ |
| `ENDFOR` | | |
| `RETURN j` | $d$ | $1$ |

- resultando

$$
\begin{aligned}
T_{FIB2}(n) &= b + \sum_{k=1}^{n} (c_1 + (c_2 + c_3) * tamaño(j)) + d = \\
&\leq (b+d) + \sum_{k=1}^{n} (c_1 + (c_2 + c_3) * tamaño(F_k)) \\
&\leq (b+d) + nc_1 + \sum_{k=1}^{n} dk \ \text{ por } \ tamaño(F_k) \in \Theta(k) \\
&= (b+d) + nc_1 + d\frac{n(n+1)}{2} = \\
&= (b+d) + nc_1 + \frac{d}{2}n^2 + \frac{d}{2}n \ \in O(n^2)
\end{aligned}
$$

- es importante observar que $n$ es el valor de la entrada y no la longitud de la entrada Ejercicio: transformar este tiempo de ejecución a una función de la longitud de la entrada

# NÚMERO DE FIBONACCI

Algoritmo: algoritmo iterativo simple

- para analizar el $O(\cdot)$ se tiene:

$$T_{FIB2}(n) \quad = \quad b + \sum_{k=1}^{n} (c_1 + c_2 + c_3) + d \leq fn$$

- luego $T_{FIB2}(n) \in O(n)$
- para analizar el $\Omega(\cdot)$ se tiene:

$$T_{FIB2}(n) \quad = \quad b + \sum_{k=1}^{n} (c_1 + c_2 + c_3) + d \geq \sum_{k=1}^{n} (c_1 + c_2 + c_3) \geq n$$

- luego $T_{FIB2}(n) \in \Omega(n)$, y por lo tanto también $T_{FIB2}(n) \in \Theta(n)$

- si la suma y la resta no son operaciones elementales (operan sobre números muy grandes) entonces

```
Function FIB2(n)
    i ::= 1; j ::= 0
    FOR k ::= 1 TO n
        j ::= i+j
        i ::= j-i
    ENDFOR
    RETURN j
```

| costo | veces |
|---|---|
| $b$ | $1$ |
| $c_1$ | $\sum_{k=1}^{n} 1$ |
| $c_2 * tamaño(j)$ | $\sum_{i=1}^{n} 1$ |
| $c_3 * tamaño(j)$ | $\sum_{i=1}^{n} 1$ |
| $d$ | $1$ |

- resultando

$$
\begin{aligned}
T_{FIB2}(n) &= b + \sum_{k=1}^{n} (c_1 + (c_2 + c_3) * tamaño(j)) + d = \\
&\leq (b+d) + \sum_{k=1}^{n} (c_1 + (c_2 + c_3) * tamaño(F_k)) \\
&\leq (b+d) + nc_1 + \sum_{k=1}^{n} dk \ \text{ por } \ tamaño(F_k) \in \Theta(k) \\
&= (b+d) + nc_1 + d\frac{n(n+1)}{2} = \\
&= (b+d) + nc_1 + \frac{d}{2}n^2 + \frac{d}{2}n \in O(n^2)
\end{aligned}
$$

# NÚMERO DE FIBONACCI

## Algoritmo: algoritmo iterativo complejo

```
function FIB3(n)
   i ::= 1; j,k ::= 0; h ::= 1
   WHILE n>0
      IF n es impar
         t ::= j*h
         j ::= i*h+j*k+t
         i ::= i*k+t
      ENDIF
      t ::= h^2; h ::= 2*k*h+t
      k ::= k^2+t
      n ::= n div 2
   ENDWHILE
```

# NÚMERO DE FIBONACCI

Algoritmo: algoritmo iterativo complejo

- no analizaremos la correctitud del algoritmo
- calculando el tiempo de ejecución se tiene

$$T_{FIB3}(n) = c1 + \sum_{k=1}^{\log n} c2 + \sum_{k=1}^{\log n} c3$$

- no sólo estamos suponiendo que sumas y restas se computan en tiempo constante (como en FIB2), sino también productos y cuadrados

# Notación Asintótica Condicional

- muchos algoritmos son más fáciles de analizar si se restringe la atención a instancias cuyos tamaños satisfacen determinadas condiciones

$$\begin{aligned} O(g(n) \mid P(n)) = \{ \quad & f(n) : \exists c \in \mathbf{R}^{+} \exists n_0 \in \mathbf{N}, \text{ tal que} \\ & f(n) \leq cg(n) \text{para todo } n \geq n_0 \\ & \text{siempre que } P(n) \ \} \end{aligned}$$

- por ejemplo, $t(n) \in \Theta(n^2 \mid n = 2^k)$ significa que si $n$ es potencia de 2 entonces $t(n) \in \Theta(n^2)$
- nada se está afirmando sobre $t(n)$ si $n$ no es potencia de 2

# Regla de las Funciones de Crecimiento Suave

- sirve para extender lo analizado condicionalmete a todos los tamaños de entrada

**Teorema 1**

*Sea $f : \mathbf{N} \longrightarrow \mathbf{R}^+$ una función de crecimiento suave, y $t : \mathbf{N} \longrightarrow \mathbf{R}^+$ una función eventualmente no decreciente. Luego siempre que $t(n) \in \Theta(f(n) \mid n = b^k)$ para algún entero $b \geq 2$, entonces $t(n) \in \Theta(f(n))$*

### Definición

*una función* $f(n) : \mathbf{N} \longrightarrow \mathbf{R}^{+}$ *es eventualmente no decreciente si existe* $n_0 \in \mathbf{N}$ *tal que para todo* $n \geq n_0$ *vale* $f(n) \leq f(n+1)$

### Definición

*una función* $f(n) : \mathbf{N} \longrightarrow \mathbf{R}^{+}$ *es de crecimiento suave si existe* $b \in \mathbf{N}, b \geq 2$ *tal que* $f(n)$ *es eventualmente no decreciente y* $f(bn) \in O(f(n))$

- la mayoría de las funciones que se encuentran son de crecimiento suave: $\log n$, $n$, $n \log n$, $n^2$, o cualquier polinomio con coeficiente principal positivo
- funciones tales como $n^{\log n}$, $2^n$ o $n!$ no son de crecimiento suave
- reglas análogas también son válidas para $O(\cdot)$ y $\Omega(\cdot)$

# Ejemplo

- si $t(n)$ es

$$t(n) = \begin{cases} a & \text{si } n = 1 \\ 4t(\lceil n/2 \rceil) + bn & \text{sino} \end{cases}$$

- entonces es fácil probar (usando los métodos de resolución de recurrencias que se verán) que $t(n) = (a+b)n^2 - bn$ si $n = 2^k$, ie $t(n) \in \Theta(n^2 \mid n = 2^k)$
- luego, como $n^2$ es una función de crecimiento suave y $t(n)$ es eventualmente no decreciente (¿porqué?) se puede aplicar la regla de las funciones de crecimiento suave y concluir que $t(n) \in \Theta(n^2)$ para todo $n$

# NÚMERO DE FIBONACCI

Algoritmo: algoritmo iterativo complejo

$$T_{FIB3}(n) \quad = \quad c1 + \sum_{k=1}^{\log n} c2 + \sum_{k=1}^{\log n} c3$$

- si $n = 2^k$ entonces

$$T_{FIB3}(n) \quad = \quad c1 + \sum_{k=1}^{\log n} c3 = d \log n$$

- y entonces $T_{FIB3}(n) \in \Theta(\log n \mid n = 2^k)$

- si $n = 2^k - 1$ entonces

$$T_{FIB3}(n) = c1 + \sum_{k=1}^{\log n} (c2 + c3)$$

- estos casos se pueden obviar aplicando la regla de las funciones de crecimiento suave
- como $T_{FIB3}(n)$ es eventualmente no decreciente, y $\log n$ es una función de crecimiento suave, entonces podemos afirmar que $T_{FIB3}(n) \in \Theta(\log n)$

# Comparación de los tres algoritmos para NUMERO DE FIBONACCI

- implementando los algoritmos en una máquina determinada, y con las herramientas que se introducirán se puede establecer:

$$
\begin{aligned}
T_{FIB1}(n) &\approx ((1+\sqrt{5})/2)^{n-20} \text{ segundos} \\
T_{FIB2}(n) &\approx 15n \text{ microsegundos} \\
T_{FIB3}(n) &\approx 1/4 \log n \text{ milisegundos}
\end{aligned}
$$

# Comparación de los tres algoritmos para NUMERO DE FIBONACCI

| n | 10 | 20 | 30 | 50 | 100 | 10.000 | $10^6$ | $10^8$ |
|---|---|---|---|---|---|---|---|---|
| *Fib*1 | 8 mseg | 1 seg | 2 min | 21 días | $10^9$ años | | | |
| *Fib*2 | 1/6 mseg | 1/3 mseg | 1/2 mseg | 4/4 mseg | 1,5 mseg | 150 mseg | 15 seg | 25 min |
| *Fib*3 | 1/3 mseg | 2/5 mseg | 1/2 mseg | 1/2 mseg | 1/2 mseg | 1 mseg | 1,5 mseg | 2 mseg |

- estos son tiempos absolutos, dependientes de una implementación y del hardware subyacente
- sin embargo, la relación entre estos tiempos se mantendrá cambiando implementación y/o hardware

# Resumen 1

- para acelerar el análisis de los algoritmos, es necesario usar herramientas que simplifiquen las funciones y permitan una rápida comparación: las distintas notaciones asintóticas
- conocer las características computacionales de las estructuras de datos que se usan en cada algoritmo es fundamental para analizar los algoritmos
- el análisis de los ciclos o iteraciones requiere entender muy bien qué es lo que se realiza en cada iteración, para poder determinar la función característica o el barómetro
- las recurrencias son funciones recursivas que surgen del análisis de algoritmos recursivos. Más adelante se verá cómo transformarlas eliminando la recursividad de su definición

# Resumen 2

- no interesa tanto nivel de detalle como para individualizar el costo de cada sentencia. Además, esto haría el análisis dependiente del lenguaje de programación y de la plataforma de ejecución
- es importante aceptar el **principio de invarianza**, que establece que un mismo algoritmo puede ser implementado con diferencia de factores constantes en distintos lenguajes o plataformas
- no es superfluo buscar la eficiencia, puede significar la diferencia entre obtener las soluciones del problema o no. El avance del hardware no tiene tanto impacto como el avance en los algoritmos