

Algoritmos y Complejidad

Introducción

Pablo R. Fillottrani

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

Primer Cuatrimestre 2017



Introducción

Algoritmos y Algoritmia

Problemas e instancias

Tipos de análisis de eficiencia

Algunos ejemplos simples



- ▶ 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

$$\begin{array}{r} \times \\ 1234 \\ 2943 \\ 1962 \\ 981 \\ \hline 1210554 \end{array}$$

A la inglesa

$$\begin{array}{r} \times \\ 1234 \\ 2943 \\ 1962 \\ 981 \\ \hline 1210554 \end{array}$$

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
<hr/>		
		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
- ▶ veremos más adelante como tratar estas funciones



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**

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

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



NÚMERO DE FIBONACCI

Algoritmo: algoritmo iterativo simple

	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	$\sum_{i=1}^n 1$
i ::= j-i	c_3	$\sum_{i=1}^n 1$
ENDFOR		
return j	d	1

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



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**



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:

$$T_{FIB1}(n) \approx ((1 + \sqrt{5})/2)^{n-20} \text{ segundos}$$

$$T_{FIB2}(n) \approx 15n \text{ microsegundos}$$

$$T_{FIB3}(n) \approx 1/4 \log n \text{ milisegundos}$$



Comparación de los tres algoritmos para **NUMERO DE FIBONACCI**

n	10	20	30	50	100	10.000	10^6	10^8
<i>Fib1</i>	8 mseg	1 seg	2 min	21 días	10^9 años			
<i>Fib2</i>	1/6 mseg	1/3 mseg	1/2 mseg	4/4 mseg	1,5 mseg	150 mseg	15 seg	25 min
<i>Fib3</i>	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

Se puede concluir a partir de los ejemplos anteriores que:

1. 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
2. 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
3. **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
4. se necesitan **herramientas matemáticas** que ayuden a manejar las fórmulas de los tiempos de ejecución

