



DEPARTAMENTO
DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

Algoritmos y Estructuras de Datos III

Primer cuatrimestre 2021 (*dictado a distancia*)

Algoritmos - Complejidad computacional

Programa

1. Algoritmos:

- ▶ Definición de algoritmo. Máquina RAM. Complejidad. Algoritmos de tiempo polinomial y no polinomial. Límite inferior.
- ▶ Técnicas de diseño de algoritmos: *divide and conquer*, *backtracking*, algoritmos golosos, programación dinámica.
- ▶ Algoritmos aproximados y algoritmos heurísticos.

Programa

2. Grafos:

- ▶ Definiciones básicas. Adyacencia, grado de un nodo, isomorfismos, caminos, conexión, etc.
- ▶ Grafos eulerianos y hamiltonianos.
- ▶ Grafos bipartitos.
- ▶ Árboles: caracterización, árboles orientados, árbol generador.
- ▶ Planaridad. Coloreo. Número cromático.
- ▶ Matching, conjunto independiente. Recubrimiento de aristas y vértices.

3. Algoritmos en grafos y aplicaciones:

- ▶ Representación de un grafo en la computadora: matrices de incidencia y adyacencia, listas.
- ▶ Algoritmos de búsqueda en grafos: BFS, DFS.
- ▶ Mínimo árbol generador, algoritmos de Prim y Kruskal.
- ▶ Algoritmos para encontrar el camino mínimo en un grafo: Dijkstra, Ford, Floyd, Dantzig.
- ▶ Algoritmos para determinar si un grafo es planar. Algoritmos para coloreo de grafos.
- ▶ Algoritmos para encontrar el flujo máximo en una red: Ford y Fulkerson.
- ▶ Matching: algoritmos para correspondencias máximas en grafos bipartitos. Otras aplicaciones.

4. Complejidad computacional:

- ▶ Problemas tratables e intratables. Problemas de decisión. P y NP. Máquinas de Turing determinísticas vs no determinísticas. Problemas NP-completos. Relación entre P y NP.
- ▶ Problemas de grafos NP-completos: coloreo de grafos, grafos hamiltonianos, recubrimiento mínimo de las aristas, corte máximo, etc.

Bibliografía

Según orden de utilización en la materia:

1. G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.
2. T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, McGraw-Hill, 2001.
3. F. Harary, *Graph theory*, Addison-Wesley, 1969.
4. J. Gross and J. Yellen, *Graph theory and its applications*, CRC Press, 1999.
5. R. Ahuja, T. Magnanti and J. Orlin, *Network Flows: Theory, Algorithms, and Applications*, Prentice-Hall, 1993.
6. M. Garey and D. Johnson, *Computers and intractability: a guide to the theory of NP- Completeness*, W. Freeman and Co., 1979.

Algoritmos

- ▶ ¿Qué es un algoritmo?
- ▶ ¿Qué es un buen algoritmo?
- ▶ Dados dos algoritmos para resolver un mismo problema, ¿cuál es mejor?
- ▶ ¿Cuándo un problema está bien resuelto?

Algoritmos

- ▶ Secuencia finita de pasos que termina en un tiempo finito.
- ▶ Deben estar formulados en términos de pasos sencillos que sean:
 - ▶ precisos: se indica el orden de ejecución de cada paso
 - ▶ bien definidos: en toda ejecución del algoritmo se debe obtener el mismo resultado bajo los mismo parámetros
 - ▶ finitos: el algoritmo tiene que tener un número determinado de pasos
- ▶ Los describiremos mediante *pseudocódigo*.

Pseudocódigo

Encontrar el máximo de un vector de enteros:

maximo(*A*)

entrada: un vector de enteros *A* no vacío

salida: el elemento máximo de *A*

$max \leftarrow A[0]$

para $i = 1$ **hasta** $dim(A) - 1$ **hacer**

si $A[i] > max$ **entonces**

$max \leftarrow A[i]$

fin si

fin para

retornar max

Análisis de algoritmos

En general, analizaremos los algoritmos utilizando como medida de eficiencia su tiempo de ejecución.

Análisis empírico: implementar los algoritmos disponibles en una máquina determinada utilizando un lenguaje determinado, luego ejecutarlos sobre un conjunto de instancias representativas y comparar sus tiempos de ejecución. Desventajas:

- ▶ pérdida de tiempo y esfuerzo de programador
- ▶ pérdida de tiempo de cómputo
- ▶ conjunto de instancias acotado

Análisis teórico: determinar matemáticamente la cantidad de tiempo que llevará su ejecución como una función de la medida de la instancia considerada, independizándonos de la máquina sobre la cuál es implementado el algoritmo y el lenguaje para hacerlo. Para esto necesitamos definir:

- ▶ un modelo de cómputo
- ▶ un lenguaje sobre este modelo
- ▶ tamaño de la instancia
- ▶ instancias relevantes

Modelo de cómputo: Máquina RAM

Motivación: Modelar computadoras en las que la memoria es suficiente y donde los enteros involucrados en los cálculos entran en una palabra. Esta memoria está dividida en celdas, que llamaremos registros, que se pueden acceder (leer o escribir) de forma directa.

Las partes que componen una RAM son:

- ▶ Unidad de entrada
- ▶ Unidad de salida
- ▶ Memoria
- ▶ Acumudor
- ▶ Programa

Modelo de cómputo: Máquina RAM

► *Unidad de entrada:*

- representa la instancia de entrada
- sucesión de celdas
- cada una con un dato de tamaño arbitrario
- se lee secuencialmente
- cuando una celda es leída, se avanza a la siguiente celda
- no puede volver a ser leída una celda leída previamente.

► *Unidad de salida:*

- sucesión de celdas
- se escribe secuencialmente
- cuando una celda es escrita, se avanza a la siguiente celda
- no puede volver a ser escrita una celda escrita previamente.

Modelo de cómputo: Máquina RAM

- ▶ *Memoria:*

- ▶ sucesión de celdas numeradas (registros)
- ▶ cada una puede almacenar un dato de tamaño arbitrario
- ▶ hay tantos registros como se necesiten
- ▶ se puede acceder de forma directa a cualquier celda.

- ▶ *Acumulador:* registro especial donde se realizan los cálculos aritméticos y lógicos.

- ▶ *Programa:*

- ▶ sucesión finita de instrucciones
- ▶ el proceso comienza ejecutando la primera instrucción
- ▶ las instrucciones son ejecutadas secuencialmente (respetando las instrucciones de control de flujo).

Máquina RAM - Instrucciones

Instrucciones de movimientos de datos:

- ▶ LOAD operando - *carga un valor en el acumulador*
- ▶ STORE operando - *carga el acumulador en un registro*
- ▶ READ operando - *lee un nuevo dato de entrada → operando*
- ▶ WRITE operando - *escribe el operando a la salida*

Operaciones aritméticas:

- ▶ ADD operando - *suma el operando al acumulador*
- ▶ SUB operando - *resta el operando al acumulador*
- ▶ MULT operando - *multiplica el operando por el acumulador*
- ▶ DIV operando - *divide el acumulador por el operando*

Instrucciones de control de flujo:

- ▶ JUMP label - *salto incondicional*
- ▶ JGTZ label - *salta si el acumulador es positivo*
- ▶ JZERO label - *salta si el acumulador es cero*
- ▶ HALT - *termina el programa*

Máquina RAM - Operandos

- ▶ $\text{LOAD } = a$ - *carga en el acumulador el valor a*
- ▶ $\text{LOAD } i$ - *carga en el acumulador el contenido del registro i*
- ▶ $\text{LOAD } *i$ - *carga en el acumulador el contenido del registro indexado por el valor del registro i*

Máquina RAM - Ejemplos

$a \leftarrow b$		LOAD	2
		STORE	1
$a \leftarrow b - 3$		LOAD	2
		SUB	=3
		STORE	1
mientras $x > 0$ hacer $x \leftarrow x - 2$ fin mientras	guarda	LOAD	1
		JGTZ	mientras
		JUMP	finmientras
	mientras	SUB	=2
		STORE	1
		JUMP	guarda
	finmientras	

Máquina RAM - Ejemplos

si $x \leq 0$ **entonces**

$y \leftarrow x + y$

sino

$y \leftarrow x$

fin si

guarda	LOAD	1
	JGTZ	sino
	ADD	2
	STORE	2
	JUMP	finsi
sino	STORE	2
finsi	...	

Programa para calcular k^k - Pseudocódigo

kalak(k)

entrada: $k \in \mathbb{Z}$

salida: k^k si $k > 0$, 0 caso contrario

si $k \leq 0$ **entonces**

$x \leftarrow 0$

sino

$x \leftarrow k$

$y \leftarrow k - 1$

mientras $y > 0$ **hacer**

$x \leftarrow x \cdot k$

$y \leftarrow y - 1$

fin mientras

fin si

retornar x

Programa para calcular k^k - máquina RAM

	READ	1	<i>carga en R1 la primera celda de la unidad de entrada</i>
	LOAD	1	<i>carga en el acumulador el valor de R1</i>
	JGTZ	sino	<i>si el valor del acumulador es > 0 salta a sino</i>
	LOAD	= 0	<i>carga 0 en el acumulador</i>
	STORE	2	<i>escribe el valor del acumulador en R2</i>
	JUMP	finsi	<i>salta a finsi</i>
sino	STORE	2	<i>escribe el valor del acumulador en R2</i>
	SUB	= 1	<i>resta 1 al valor del acumulador</i>
	STORE	3	<i>escribe el valor del acumulador en R3</i>
guarda	LOAD	3	<i>carga en el acumulador el valor de R3</i>
	JGTZ	mientras	<i>si el valor del acumulador es ≥ 0 salta a mientras</i>
	JUMP	finmientras	<i>salta a finmientras</i>
mientras	LOAD	2	<i>carga en el acumulador el valor de R2</i>
	MULT	1	<i>multiplica el valor del acumulador por el valor de R1</i>
	STORE	2	<i>escribe el valor del acumulador en R2</i>
	LOAD	3	<i>carga en el acumulador el valor de R3</i>
	SUB	= 1	<i>resta 1 al valor del acumulador</i>
	STORE	3	<i>escribe el valor del acumulador en R3</i>
	JUMP	guarda	<i>salta a guarda</i>
finmientras finsi	WRITE	2	<i>escribe el valor de R2 en la unidad de salida</i>
	HALT		<i>para la ejecución</i>

Complejidad computacional

Definición informal: La *complejidad* de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada del algoritmo.

Definición formal?

Una operación es *elemental* si su tiempo de ejecución puede ser acotado por una constante que

- ▶ depende sólo de la implementación particular utilizada (la máquina, el lenguaje de programación)
- ▶ no depende de la medida de los parámetros de la instancia considerada.

Complejidad en la Máquina RAM

- ▶ Cada instrucción tiene un *tiempo de ejecución* asociado.
- ▶ Tiempo de ejecución de un algoritmo A con instancia I , $t_A(I)$ = suma de los tiempos de ejecución de las instrucciones realizadas por el algoritmo con la instancia I .
- ▶ Ineteresa *cómo* crece el tiempo de ejecución de un algoritmo cuando el tamaño de las instancias de entrada crece. No interesa el tiempo exacto requerido por cada una de ellas.
- ▶ s sumas, m multiplicaciones y a asignaciones, que consumen t_s , t_m y t_a microsegundos respectivamente. Entonces:

$$t_A(I) \leq st_s + mt_m + at_a \leq \max(t_s, t_m, t_a) * (s + m + a)$$

- ▶ Simplificamos asumiendo que toda operación elemental puede ser ejecutada en una unidad de tiempo.

Operaciones básicas en la práctica

suma(A)

entrada: arreglo de enteros A

salida: suma de los elemento de A

$resu \leftarrow 0$

para $i = 0$ **hasta** $\dim(A) - 1$ **hacer**

$resu \leftarrow resu + A[i]$

fin para

retornar $resu$

El valor de $resu$ se mantiene en una medida *razonable* para todas las instancias que esperamos encontrarnos en la práctica.

Operaciones básicas en la práctica

factorial(*n*)

entrada: $n \in \mathbb{N}$

salida: $n!$

resu $\leftarrow 1$

para $i = 2$ **hasta** n **hacer**

resu $\leftarrow resu * i$

fin para

retornar *resu*

Ya para valores relativamente chicos de n , no es para nada realista considerar que la operación $resu \leftarrow resu * i$ puede ser realizada en una unidad de tiempo.

Operaciones básicas en la práctica: modelo uniforme vs modelo logarítmico

- ▶ *Modelo uniforme*: Cada operación básica tiene un tiempo de ejecución constante.
 - ▶ Consiste en determinar el número total de operaciones básicas ejecutadas por el algoritmo.
 - ▶ Es un método sencillo, pero puede generar serias anomalías para valores arbitrariamente grandes de operandos.
 - ▶ Apropiado cuando los operandos entran en una palabra.

Operaciones básicas en la práctica: modelo uniforme vs modelo logarítmico

- ▶ *Modelo logarítmico*: El tiempo de ejecución de cada operación es función del tamaño (cantidad de bits) de los operandos.
 - ▶ Apropiado cuando los operandos crecen arbitrariamente.
 - ▶ Una suma proporcional al tamaño del mayor operando.
 - ▶ Una multiplicación *ingenua* proporcional al producto de los tamaños
 - ▶ Por ejemplo, en el cálculo del factorial los operandos de los cálculos que se realizan cambian de orden de magnitud con respecto al valor de la entrada.

Tamaño de una instancia

Dada una instancia I , se define $|I|$ como el número de símbolos de un alfabeto finito necesarios para codificar I .

- ▶ Depende del *alfabeto*.
- ▶ Para almacenar $n \in \mathbb{N}$, se necesitan $L(n) = \lfloor \log_2(n) \rfloor + 1$ dígitos binarios.
- ▶ Para almacenar una lista de m enteros, se necesitan $L(m) + mL(N)$ dígitos binarios, donde N es el valor máximo de la lista (notar que se puede mejorar!).

Tamaño de una instancia

- ▶ Rigurosidad depende del problema que se esté analizando.
- ▶ Para conjuntos, arreglos, matrices, grafos, usaremos el número de componentes. Esto modela de forma suficientemente precisa la realidad y facilita el análisis de los algoritmos.
- ▶ Cuando trabajamos con algoritmos que sólo reciben un número fijo de valores como parámetros, esta simplificación hace que carezca de sentido el análisis, y entonces debemos alejarnos de esta regla, midiendo el tamaño de la instancia la cantidad de bits necesarios para almacenar los parámetros.

Análisis promedio y peor caso

esta?(A, elem)

entrada: arreglo de enteros A no vacío,
entero $elem$

salida: *Verdadero* si $elem$ se encuentra en A ,
Falso en caso contrario

$i \leftarrow 0$

mientras $i < \dim(A)$ **y** $elem \neq A[i]$ **hacer**
 $i \leftarrow i + 1$

fin mientras

si $i < \dim(A)$ **entonces**
retornar *Verdadero*

sino
retornar *Falso*

Análisis promedio y peor caso

► *Complejidad en el peor caso:*

- Para cada tamaño, instancia que usa mayor cantidad de tiempo.
- Complejidad de un algoritmo A para las instancias de tamaño n , $T_A(n)$, como:

$$T_A(n) = \max_{I: |I|=n} t_A(I).$$

► *Complejidad en el caso promedio:*

- Tiempo de ejecución promedio sobre el conjunto de instancias determinado.
- Este análisis del tiempo de ejecución promedio es más difícil.
- Se debe conocer la distribución de las instancias que se resolverán y en muchas aplicaciones esto no es posible.

Análisis promedio y peor caso

- ▶ Si no se aclara lo contrario, en la materia siempre consideraremos el análisis del peor caso.
- ▶ Cuando no haya confusión a qué algoritmo nos estamos refiriendo, vamos a obviar el subíndice A , utilizando $T(n)$ en lugar de $T_A(n)$.

Notación \mathcal{O}

Dadas dos funciones $T, g : \mathbb{N} \rightarrow \mathbb{R}$, decimos que:

- ▶ $T(n) = \mathcal{O}(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$T(n) \leq c g(n) \text{ para todo } n \geq n_0.$$

T no crece más rápido que g , $T \preceq g$.

- ▶ $T(n) = \Omega(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que

$$T(n) \geq c g(n) \text{ para todo } n \geq n_0.$$

T crece al menos tan rápido como g , $T \succeq g$.

- ▶ $T(n) = \Theta(g(n))$ si

$$T = \mathcal{O}(g(n)) \text{ y } T = \Omega(g(n)).$$

T crece al mismo ritmo que g , $T \approx g$.

Notación \mathcal{O} - Ejemplos

- ▶ $T(n)$ es $\mathcal{O}(g(n))$ si y sólo si $g(n)$ es $\Omega(T(n))$.
- ▶ $2n^2 + 10n$ es $\mathcal{O}(n^2)$, porque tomando $n_0 = 0$ y $c = 12$, tenemos que

$$2n^2 + 10n \leq 12n^2.$$

- ▶ $2n^2 + 10n$ es $\mathcal{O}(n^3)$, pero nos interesa calcular el orden más ajustado posible.
- ▶ Si una implementación de un algoritmo requiere en el peor caso $2n^2 + 10n$ microsegundos para resolver una instancia de tamaño n , podemos simplificar diciendo que el algoritmo es $\mathcal{O}(n^2)$.
- ▶ El uso de microsegundos es totalmente irrelevante, ya que sólo necesitamos cambiar la constante para acotar el tiempo por años o nanosegundos.

Notación \mathcal{O} - Ejemplos

- ▶ 3^n no es $\mathcal{O}(2^n)$.
 - ▶ Vamos a demostrarlo por el absurdo.
 - ▶ Supongamos que sí, es decir que 3^n es $\mathcal{O}(2^n)$.
 - ▶ Entonces, por definición, existirían $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tales que $3^n \leq c 2^n$ para todo $n \geq n_0$.
 - ▶ Por lo tanto, $(\frac{3}{2})^n \leq c$ para todo $n \geq n_0$.
 - ▶ Esto genera un absurdo porque c debe ser una constante y no es posible que una constante siempre sea mayor que $(\frac{3}{2})^n$ cuando n crece.

Notación \mathcal{O} - Ejemplos

- ▶ Si $a, b \in \mathbb{R}_+$, entonces $(\log_a(n))$ es $\Theta(\log_b(n))$.
Es decir, todas las funciones logarítmicas crecen de igual forma sin importar la base.
- ▶ Como $\log_a(n) = \frac{\log_b(n)}{\log_b(a)}$
- ▶ la constante $\frac{1}{\log_b(a)}$ sirve tanto para ver que $(\log_a(n))$ es $\mathcal{O}(\log_b(n))$
- ▶ como para $(\log_a(n))$ es $\Omega(\log_b(n))$.

Notación \mathcal{O} usando lím

Si $\lim_{n \rightarrow \infty} \frac{T(n)}{g(n)} = a$ con $0 \leq a < \infty$, significa que $|\frac{T(n)}{g(n)} - a| < \epsilon$ para algún $\epsilon > 0$. Entonces, $\frac{T(n)}{g(n)} < \epsilon + a$, donde $\epsilon + a$ es una constante, lo que es equivalente a decir que $T(n)$ es $\mathcal{O}(g(n))$.

De forma similar podemos analizar Ω y Θ , llegando a las siguientes propiedades.

- ▶ $T(n)$ es $\mathcal{O}(g(n))$ si y sólo si $\lim_{n \rightarrow \infty} \frac{T(n)}{g(n)} \in [0, \infty)$.
- ▶ $T(n)$ es $\Omega(g(n))$ si y sólo si $\lim_{n \rightarrow \infty} \frac{T(n)}{g(n)} \in (0, \infty]$.
- ▶ $T(n)$ es $\Theta(g(n))$ si y sólo si $\lim_{n \rightarrow \infty} \frac{T(n)}{g(n)} \in (0, \infty)$.

Notación \mathcal{O} usando lím - Ejemplos

- ▶ Cualquier función exponencial es *peor* que cualquier función polinomial: Si $k, d \in \mathbb{N}$ entonces k^n no es $\mathcal{O}(n^d)$.
 - ▶ Por la propiedad anterior, sabemos que k^n es $\mathcal{O}(n^d)$ si y sólo si $\lim_{n \rightarrow \infty} \frac{k^n}{n^d} \in [0, \infty)$.
 - ▶ Aplicando l'Hôpital, podemos ver que $\lim_{n \rightarrow \infty} \frac{k^n}{n^d} = \infty$.
 - ▶ Entonces, por lo tanto, k^n no es $\mathcal{O}(n^d)$

Notación \mathcal{O} usando lím - Ejemplos

- ▶ La función logarítmica es *mejor* que la función lineal (ya vimos que no importa la base), es decir $\ln(n) \prec \mathcal{O}(n)$:

$$\ln(n) \text{ es } \mathcal{O}(n) \text{ y } n \text{ no es } \mathcal{O}(\ln(n))$$

- ▶ Nuevamente, operando y aplicando l'Hôpital, podemos ver que $\lim_{n \rightarrow \infty} \frac{\ln(n)}{n} = 0$, que es lo mismo que $\lim_{n \rightarrow \infty} \frac{n}{\ln(n)} = \infty$.
- ▶ Lo que implica que $\ln(n)$ es $\mathcal{O}(n)$ ($\ln(n)$ no crece más rápido que n)
- ▶ y que n no es $\mathcal{O}(\ln(n))$ ($\ln(n)$ no acota superiormente el crecimiento de n).

Notación \mathcal{O}

- ▶ Si un algoritmo es $\mathcal{O}(n)$, se dice *lineal*.
- ▶ Si un algoritmo es $\mathcal{O}(n^2)$, se dice *cuadrático*.
- ▶ Si un algoritmo es $\mathcal{O}(n^3)$, se dice *cúbico*.
- ▶ Si un algoritmo es $\mathcal{O}(n^k)$, $k \in \mathbb{N}$, se dice *polinomial*.
- ▶ Si un algoritmo es $\mathcal{O}(\log n)$, se dice *logarítmico*.
- ▶ Si un algoritmo es $\mathcal{O}(d^n)$, $d \in \mathbb{R}_+$, se dice *exponencial*.

Notación \mathcal{O}

Dos algoritmos para resolver el mismo problema. Uno necesita n^2 días y el otro n^3 segundos.

- ▶ Desde un punto de vista teórico, el primero es *asintóticamente* mejor que el segundo
- ▶ Es decir, su performance es mejor sobre todas las instancias suficientemente grandes.
- ▶ Sin embargo, desde un punto de vista práctico, seguramente prefiramos el algoritmo cúbico.
- ▶ Esto sucede porque la constante multiplicativa del primero es muy grande para ser ignorada cuando se consideran instancias de tamaño razonable.
- ▶ Si bien la diferencia de magnitud de las constantes en este ejemplo fue muy grosera, la intención es remarcar que para instancias de tamaño relativamente pequeño puede que el análisis asintótico no se adecuado.

Complejidad de algoritmos conocidos

- ▶ Búsqueda secuencial: $\mathcal{O}(n)$.
- ▶ Búsqueda binaria: $\mathcal{O}(\log(n))$.
- ▶ Ordenar un arreglo (bubblesort): $\mathcal{O}(n^2)$.
- ▶ Ordenar un arreglo (quicksort): $\mathcal{O}(n^2)$ en el peor caso (!).
- ▶ Ordenar un arreglo (heapsort): $\mathcal{O}(n \log(n))$.

Es interesante notar que $\mathcal{O}(n \log(n))$ es la complejidad **óptima** para algoritmos de ordenamiento basados en comparaciones.

Ejemplo: Cálculo del promedio de un arreglo

promedio(A)

entrada: arreglo de reales A de tamaño ≥ 1

salida: promedio de los elementos de A

$suma \leftarrow 0$

$\mathcal{O}(1)$

para $i = 0$ **hasta** $\dim(A) - 1$ **hacer**

n veces

$suma \leftarrow suma + A[i]$

$\mathcal{O}(1)$

fin para

retornar $suma / \dim(A)$

$\mathcal{O}(1)$

- ▶ Si llamamos $n = \dim(A)$, la entrada es $\mathcal{O}(n)$.
- ▶ Utilizamos el modelo uniforme, porque asumimos que los valores de $suma$ entran en una palabra.
- ▶ El algoritmo es lineal en función del tamaño de la entrada.

Cálculo iterativo de $n!$

factorial(n)

$resu \leftarrow 1$

para $k = 2$ **hasta** n **hacer**

$resu \leftarrow resu * k$

fin para

retornar $resu$

$\mathcal{O}(1)$

$n - 1$ veces

$\mathcal{O}(n * \log_2^2(n))$

$\mathcal{O}(\log_2(n!)) = \mathcal{O}(n * \log_2(n))$

- ▶ El tamaño de la entrada es $t = \log_2(n)$.
- ▶ Modelo logarítmico, porque los operandos crecen factorialmente.
- ▶ En la iteración k -ésima, los operandos de $resu \leftarrow resu * k$ son:
 - ▶ $resu$ vale $(k - 1)!$, ocupa $\log_2((k - 1)!)$ que es menor que $k * \log_2(k)$.
 - ▶ el otro operando es k , que requiere $\log_2(k)$ bits.
- ▶ Es $\mathcal{O}(k \log_2(k) \log_2(k))$ (algoritmo de multiplicación ingenuo).
- ▶ Como $k \leq n$ esto es $\mathcal{O}(n * \log_2^2(n))$.
- ▶ Como se ejecuta $n - 1$ veces, la complejidad total es $\mathcal{O}(n^2 \log_2^2(n))$.
- ▶ El algoritmo es $\mathcal{O}(t^2 4^t)$, exponencial en el tamaño de la entrada.

Algoritmos eficientes vs no eficientes

Tiempos insumidos sobre por la misma máquina (en seg. suponiendo 0.001 miliseg por operación):

	$n = 10$	$n = 20$	$n = 30$	$n = 40$	$n = 50$
$\log(n)$	0.000001	0.000001	0.000001	0.000002	0.000002
n	0.00001	0.00002	0.00003	0.00004	0.00005
$n \log(n)$	0.00001	0.000026	0.000044	0.000064	0.000085
n^2	0.0001	0.0004	0.0009	0.0016	0.0025
n^3	0.001	0.008	0.027	0.064	0.125
n^5	0.1	3.2	24.3	1.7 min	5.2 min
2^n	0.001	1.0	17.9 min	12.7 días	35.6 años
3^n	0.59	58 min	6.5 años	3855 siglos	2 E+8 siglos
$n!$	3.63	771 siglos	8.4 E+16	2.5 E+32 siglos	9.6 E+48 siglos

- Datos del Garey y Johnson, 1979 (máquina muy vieja).
- ¿Qué pasa si tenemos una máquina 1000 veces más rápida?
¿Un millón de veces más rápida?
- ¿Cuál será el tamaño del problema que podemos resolver en una hora comparado con el que podemos resolver ahora?

Algoritmos eficientes vs no eficientes

Tamaño de las instancias que podríamos resolver en 1 hora con una máquina 1000 veces más rápida:

	actual	1000 veces mas rápida
$\log n$	N1	$N1^{1000}$
n	N2	1000 N2
n^2	N3	31.6 N3
n^3	N4	10 N4
n^5	N5	3.98 N5
2^n	N6	$N6 + 9.97$
3^n	N7	$N7 + 6.29$

- Obviamente, el tamaño de las instancias incrementa
- Pero para las complejidades exponenciales, n^2 y n^3 , el incremento es muy pequeño.
- Entonces, ¿cuándo un algoritmo es bueno o eficiente?

Algoritmos eficientes vs no eficientes

POLINOMIAL = “bueno”

EXPONENCIAL = “malo”

Conclusión: Los algoritmos polinomiales se consideran satisfactorios (cuanto menor sea el grado, mejor), y los algoritmos supra-polinomiales se consideran no satisfactorios.

Problemas bien resueltos

- ▶ Pero, ¿siempre será posible contar con algoritmos polinomiales para resolver un determinado problema?
- ▶ Existen problemas para los cuales no se conocen algoritmos polinomiales para resolverlo, aunque tampoco se ha probado que estos no existan.
- ▶ Sobre esto hablaremos hacia el final del curso.
- ▶ Por ahora nos conformamos sólo con la idea de que un problema está *bien resuelto* si existe un algoritmo de complejidad polinomial para resolverlo.

Resumen I

- ▶ Vamos a describir nuestros algoritmos mediante pseudocódigo.
- ▶ Usaremos el análisis teórico. En el labo lo validarán con el análisis empírico.
- ▶ En general, vamos a utilizar el modelo uniforme.
- ▶ Cuando los operandos de las operaciones intermedias crezcan *mucho*, para acercarnos a modelar la realidad, usaremos el modelo logarítmico.

Resumen II

- ▶ En el caso de arreglos, matrices, grafos, como tamaño de entrada generalmente usaremos la cantidad de componentes de la instancia.
- ▶ En el caso de algoritmos que reciben como parámetros sólo una cantidad fija de números, como tamaño de entrada usaremos la cantidad de bits necesarios para su representación.
- ▶ En general, calcularemos la complejidad de un algoritmo para el peor caso.
- ▶ Consideraremos que los algoritmos polinomiales son satisfactorios (cuanto menor sea el grado, mejor), mientras que los algoritmos supra-polinomiales son no satisfactorios.

Algunas dudas...

- ▶ Si los tamaños de instancia son pequeños, ¿es tan malo un algoritmo exponencial?
- ▶ ¿Cómo se comparan $\mathcal{O}(n^{85})$ con $\mathcal{O}(1,001^n)$?
- ▶ ¿Puede pasar que un algoritmo de peor caso exponencial sea eficiente en la práctica? ¿Puede pasar que en la práctica sea *el mejor*?
- ▶ ¿Qué pasa si no encuentro un algoritmo polinomial?