Teórica 1: Algoritmos - Complejidad computacional

Programa de la materia

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.

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.

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.

Complejidad computacional:

- Problemas tratables e intratables. Problemas de decisión. P y NP. Máquinas de Türing 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.

1. Algoritmos

Un ***algoritmo*** es un procedimiento para resolver un problema, descripto por una secuencia finita de pasos, que termina en un tiempo finito. Debe estar formulado en términos de pasos sencillos que sean:

- precisos: se debe indicar el orden de ejecución de cada paso,
- bien definidos: en toda ejecución del algoritmo se debe obtener el mismo resultado,
- finitos: el algoritmo tiene que tener un número determinado de pasos.

Los métodos enseñados en la primaria para multiplicar o dividir números son ejemplos de algoritmos, también el algoritmo de Euclides para calcular el máximo común divisor entre dos números.

Un algoritmo siempre debe brindar una respuesta. Puede ser que la respuesta sea que no hay respuesta. También debe estar garantizado que el algoritmo termina. La descripción debe ser clara y precisa, no dejar lugar a la utilización de la intuición o creatividad. Una receta de cocina es un algoritmo si no dice, por ejemplo, "salar a gusto".

Muchas veces, cuando tenemos que resolver un problema contamos con varios algoritmos para hacerlo. Obviamente, quisieramos elegir el *mejor* de ellos. Pero, ¿cuándo un algoritmo es mejor que otro? Algunas propiedades que nos gustaría que tenga el algoritmo pueden ser: que sea fácil de programar, que su ejecución sea rápida, que no requiera de mucho espacio de memoria. Entonces, ¿cómo medir la eficiencia de un algoritmo? Si sólo vamos a ejecutar el algoritmo sobre pocas instancias pequeñas, la decisión de cuál algoritmo elegir seguramente no sea muy importante y preferiremos el más fácil de programar. Sin embargo, si tenemos que resolver muchas instancias de medida considerable o si el problema es *difícil* debemos elegir el algoritmo que utilizaremos muy cuidadosamente. Abordaremos luego este tema en detalle.

Otra pregunta que intentaremos responder a lo largo del curso es cuándo consideramos que un ***problema*** está bien resuelto. Veremos que algunos problemas son *fáciles* de resolver mientras que otros son *difíciles*. Pero, ¿cómo se mide la dificultad de un problema? Esto lo discutiremos hacia el final del curso.

1.1. Pseudocódigo

Si queremos describir un algoritmo en lenguaje natural podemos generar confusión. Una forma de escribir un algoritmo para intentar evitar imprecisiones es mediante ***pseudocódigo***, semejantes a los usados para escribir los programas informáticos.

El pseudocódigo es una descripción informal de alto nivel de un algoritmo. En la materia no vamos a definir un pseudocódigo específico, vamos a usar el sentido común (frases cortas descriptivas que expliquen tareas específicas, sangría para mostrar bloques de instrucciones). Lo importante es transmitir el procedimiento de forma clara y precisa.

En el siguiente ejemplo mostramos el pseudocódigo de un algoritmo para calcular el entero máximo de un arreglo. Sin necesidad de habernos puesto de acuerdo en la sintaxis utilizada, creemos que todos entendemos de forma clara el algoritmo que describe.

Ejemplo 1. *Encontrar el máximo de un arreglo de enteros:*

```
maximo(A,n)
    entrada: arreglo de enteros A no vacío
    salida: el elemento máximo de A

    max ← A[0]
    para i = 1 hasta dim(A) − 1 hacer
        si A[i] > max entonces
            max ← A[i]
        fin si
    fin para
    retornar max
```

1.2. Análisis de algoritmos

Como mencionamos, cuando tenemos más de un algoritmo para resolver un problema y queremos elegir el *mejor* entre ellos, hay diferentes criterios que podemos analizar para medir la eficiencia de los mismos, según el contexto de aplicación. Los recursos de mayor interés suelen ser el tiempo de cómputo y el espacio de almacenamiento requerido, con el primero generalmente más crítico. En la materia, en general, analizaremos los algoritmos utilizando como medida de eficiencia su tiempo de ejecución.
Para realizar esta elección podemos basarnos en dos enfoques:

El análisis empírico (o a posteriori) consiste en 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.

Las principales desventajas de este enfoque son:

- pérdida de tiempo y esfuerzo de programador: ya que debemos programar todos los algoritmos que deseamos evaluar.
- pérdida de tiempo de cómputo: ya que debemos ejecutar todos los programas sobre todas las instancias de prueba
- conjunto de instancias acotado: sólo vamos a poder ejecutar los algoritmos sobre un conjunto limitado de instancias relativamente pequeñas y las conclusiones alcanzadas pueden no ser verdaderas para instancias de mayor tamaño, que son las críticas.

Además este análisis dependerá de la habilidad del programador, del lenguaje de implementación, del compilador, del sistema operativo, entre otros factores que afectan al tiempo de cómputo requerido por un programa.

El análisis teórico (o a priori) consiste en 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 del lenguaje para hacerlo. Para esto necesitamos definir:

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

1.3. Modelo de cómputo: Máquina RAM

En el análisis de un algoritmo, nos queremos independizar de la plataforma y lenguaje particular que utilicemos al implementarlo. Entonces, lo primero que haremos, es definir un modelo de cómputo formal conveniente sobre el cual haremos el análisis. La ***máquina de acceso random (RAM por Random Access Machine)*** fue introducida por Cook y Reckhow en 1973 [1] para estudiar la complejidad algorítmica de programas escritos en computadoras basadas en registros. Modela, adecuadamente para este propósito, computadoras en las que la memoria es suficiente y donde los datos 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. Esta es la principal característica de una RAM y lo que le da su nombre (acceso random).

Las partes que componen una RAM son:

- ***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.
- ***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 ariméticos 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).

Para reflejar computadoras reales, el conjunto de instrucciones de una RAM es el esperable en cualquier computadora: operaciones aritméticas, instrucciones de movimiento de datos e instrucciones de control de flujo:

Instrucciones de movimientos de datos	
LOAD $= a$	Carga en el acumulador el valor a
LOAD i	Carga en el acumulador el contenido del registro i
STORE i	Guarda el contenido del acumulador en el registro i
READ i	Lee el dato de la cinta de entrada y lo guarda en el registro i
WRITE i	Escribe el contenido del registro i en la cinta de salida
Operaciones artiméticas	
ADD $= a$	Suma a al valor del acumulador. El resultado queda almacenado en el acumulador
ADD i	Suma el valor del registro i al valor del acumulador. El resultado queda almacenado en el acumulador
SUB $= a$	Resta a al valor del acumulador. El resultado queda almacenado en el acumulador
SUB i	Resta el valor del registro i al valor del acumulador. El resultado queda almacenado en el acumulador
MULT $= a$	Multiplica a por el valor del acumulador. El resultado queda almacenado en el acumulador
MULT i	Multiplica el valor del registro i por el del acumulador. El resultado queda en el acumulador
DIV $= a$	Divide el valor del acumulador por a. El resultado queda almacenado en el acumulador
DIV i	Divide el valor del acumulador por el del registro i. El resultado queda almacenado en el acumulador
Instrucciones de control de flujo	
JUMP *label*	Salta a la instrucción señalada con *label* (salto incondicional)
JGTZ *label*	Salta a la instrucción señalada con *label* si el valor del acumulador es positivo
JZERO *label*	Salta a la instrucción señalada con *label* si el valor del acumulador es cero
HALT	Termina el programa

Nota 1: Hay un tercer operando posible, $*i$, que indexa por el valor del registro i. Por ejemplo, **LOAD** $*i$ carga en el acumulador el contenido del registro indexado por el contenido del registro i.

Nota 2: En alguna bibliografía las instrucciones **MULT** y **DIV** no se encuentran.

En los siguientes ejemplos mostramos el programa RAM correspondiente a bloques de pseudocódigo.

Ejemplo 2. *Equivalencias pseudocódigo vs máquina RAM:*

Pseudocódigo	**Máquina RAM**		
$a \leftarrow b$		**LOAD**	*2*
		STORE	*1*
$a \leftarrow b - 3$		**LOAD**	*2*
		SUB	*=3*
		STORE	*1*
mientras $x > 0$ **hacer**	*guarda*	**LOAD**	*1*
$\quad x \leftarrow x - 2$		**JGTZ**	*mientras*
fin mientras		**JUMP**	*finmientras*
	mientras	**SUB**	*=2*
		STORE	*1*
		JUMP	*guarda*
	finmientras	....	
si $x \leq 0$ **entonces**	*guarda*	**LOAD**	*1*
$\quad y \leftarrow x + y$		**JGTZ**	*sino*
sino		**ADD**	*2*
$\quad y \leftarrow x$		**STORE**	*2*
fin si		**JUMP**	*finsi*
	sino	**STORE**	*2*
	finsi	...	

Ejemplo 3. *Programa para calcular* k^k

Pseudocódigo

```
kalak(k)
    entrada: k ∈ ℤ
    salida: k^k si k > 0,
            0 caso contrario

    si k ≤ 0 entonces
        x ← 0
    sino
        x ← k
        y ← k − 1
        mientras y > 0 hacer
            x ← x · k
            y ← y − 1
        fin mientras
    fin si
    retornar x
```

Máquina RAM

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

2. Complejidad computacional

Dicho de una forma informal, la ***complejidad*** de un algoritmo es una función que representa el tiempo de ejecución en función del tamaño de la entrada (que todavía no definimos qué es).

2.1. Operaciones elementales: modelo uniforme vs modelo logarítmico

Una operación es elemental si su tiempo de ejecución puede ser acotado por una constante dependiente sólo de la implementación particular utilizada (la máquina, el lenguaje de programación). Esta constante no depende de la medida de los parámetros de la instancia considerada.

En una Máquina RAM, asumimos que toda instrucción es una operación elemental con un **tiempo de ejecución** asociado y definimos el tiempo de ejecución de un programa A como:

$t_A(I)$ = suma de los tiempos de ejecución de las instrucciones realizadas por el programa A con la *instancia* I.

Como queremos tener una estimación de cómo crece el tiempo de ejecución de un algoritmo cuando el tamaño de las instancias de entrada crece, lo importante en el análisis es la cantidad de operaciones elementales ejecutadas, y no el tiempo exacto requerido por cada una de ellas.

Por ejemplo, supongamos que al analizar un programa A determinamos que para resolver una instancia I de un tamaño determinado, necesitamos realizar s sumas, m multiplicaciones y a asignaciones. También supongamos que una suma nunca requiere más de t_s microsegundos, una multiplicación nunca más que t_m microsegundos y una asignación (LOAD / STORE) nunca más que t_a microsegundos, donde t_s, t_m y t_a son constantes.Entonces, suma, multiplicación y asignación son operaciones elementales y el tiempo total t requerido por nuestro programa está acotado por:

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

Entonces $t_A(I)$ está acotado por una constante multiplicada por el número de operaciones elementales realizadas al ejecutar el programa sobre I. Como el tiempo exacto requerido por cada operación elemental no es importante, simplificamos asumiendo que toda operación elemental puede ser ejecutada en una unidad de tiempo.

Pasando a lo práctico... El modelo de máquina RAM brinda un marco adecuado para estudiar la complejidad algorítmica, pero no resulta conveniente para describir algoritmos. Para esto seguiremos utilizando pseudocódigo. En la especificación de un algoritmo mediante pseudocódigo (o en la implementación), una instrucción puede requerir la ejecución de varias operaciones elementales y algunas operaciones matemáticas podrían ser muy complejas para ser consideradas elementales. ¿Qué sucede por ejemplo con x^y? Además, en una máquina RAM asumimos que la suma y la multiplicación son operaciones elementales. Pero esto no es del todo cierto si pasamos a las computadoras reales, ya que el tiempo necesario para ejecutarlas incrementa con la medida de los operandos. Sin embargo, en la práctica, a veces es posible hacer esta asunción, porque los operandos envueltos en las instancias que esperamos encontrar son de una medida razonable. Si analizamos el siguiente pseudocódigo para sumar los elementos de un arreglo de enteros:

```
suma(A)
    entrada: arreglo de enteros A
    salida: suma de los elemento de A

    resu ← 0
    para i = 0 hasta dim(A) − 1 hacer
        resu ← 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. En teoría, sin embargo, el algoritmo debería poder ser aplicado a todas las instancias posibles y ninguna máquina real puede ejecutar estas sumas en tiempo constante si el valor de los elementos o la dimensión del arreglo es elegida lo suficientemente grande. Entonces, las suposiciones válidas al analizar un algoritmo dependen del dominio de aplicación esperado.

En cambio, en el siguiente pseudocódigo para calcular $n!$ la situación es muy distinta.

```
factorial(n)
    entrada: n ∈ ℕ
    salida: n!

    resu ← 1
    para i = 2 hasta n hacer
        resu ← 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.

Modelo uniforme: Se asume que cada operación básica tiene un tiempo de ejecución constante. Por lo tanto este enfoque consiste en determinar el número total de operaciones básicas ejecutadas por el algoritmo. Es

apropiado cuando los operandos de las operaciones entran en una palabra. Este método es sencillo, pero puede generar serias anomalías para valores arbitrariamente grandes de operandos.

Modelo logarítmico: El tiempo de ejecución de cada operación es una función (que dependerá del algotimo utilizado para resolverla) del tamaño (cantidad de bits) de los operandos (una suma proporcional al tamaño del mayor operando, una multiplicación *ingenua* proporcional al producto de los tamaños). Es apropiado cuando los operandos de las operaciones intermedias pueden crecer arbitrariamente. Por ejemplo, en el cáculo del factorial de un número los operandos de los cálculos que realiza el algoritmo cambian de orden de magnitud con respecto al valor de la entrada.

2.2. Tamaño de la instancia

El *tamaño* de una instancia formalmente corresponde al número de bits necesarios para representar la instancia en una computadora, usando algún esquema de codificación preciso y razonable.

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

Por lo tanto el tamaño de una instancia depende del *alfabeto* y de la *base* utilizada en la representación. Por ejemplo, para almacenar $n \in \mathbb{N}$, se necesitan $L(n) = \lfloor \log_2(n) \rfloor + 1$ dígitos binarios. Mientras que 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!).

Sin embargo, para hacer el análisis más claro, muchas veces seremos menos formales que esto, y dependiendo del problema que estemos analizando, usaremos como tamaño un entero que, de alguna forma, mida el número de componentes de una instancia. Por ejemplo, si la instancia es un conjunto de enteros, el tamaño de la entrada será la cantidad de enteros, ignorando que cada uno de ellos requiera más de un bit para ser almacenado en la computadora.

Esta simplificación alcanza para reflejar de forma adecuada el espacio de instancias que esperamos encontrar en la práctica donde podemos considerar que los valores de las componentes de las instancias se mantienen dentro de valores razonables, mientras que la cantidad de estas componentes crece de forma más significativa e influye en la performance de los algoritmos. En general, para problemas sobre arreglos, matrices o grafos, utilizaremos este enfoque.

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. En estos casos, mediremos el tamaño de la instancia como la cantidad de bits necesarios para almacenar los parámetros. Por ejemplo, para cálculo del factorial es más apropiado utilizar como tamaño de la entrada la cantidad de bits necesarios para representar la instancia de entrada en notación binaria.

Otra vez, el criterio más adecuado dependerá del contexto de uso del algoritmo.

2.3. Análisis promedio y peor caso

Ya discutimos sobre el modelo de cómputo y el tamaño de la entrada, y, como dijimos de forma informal, queremos calcular una función que *represente* el tiempo de ejecución en función del tamaño de la entrada. Pero el tiempo requerido por un algoritmo puede variar considerablemente entre dos instancias del mismo tamaño. Por ejemplo, en el siguiente algoritmo de búsqueda,

```
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 ← 0
    mientras i < dim(A) y elem ≠ A[i] hacer
        i ← i + 1
    fin mientras
    si i < dim(A) entonces
        retornar Verdadero
    sino
        retornar Falso
```

¿qué sucede si el elemento buscado se encuentre en la primera posición del arreglo? ¿Y si en cambio *elmem* no está en A? En cuanto a tiempo de ejecución, el primer escenario (que $elem = A[0]$) es lo mejor que nos puede pasar. Mientras que el segundo es lo peor.

Esto abre dos criterios posibles para el análisis (en realidad tres si agregamos mejor caso):

Complejidad en el peor caso: Para cada tamaño de instancia, consideramos aquellas para las cuales el algoritmo requiere la mayor cantidad de tiempo. Definiremos la complejidad de un algoritmo A para las instancias de tamaño n, $T_A(n)$, como:

$$T_A(n) = \text{máx}_{I:|I|=n}\, t_A(I).$$

Complejidad en el caso promedio: Por otro lado, si un algoritmo será utilizado sobre un conjunto de instancias determinado, tal vez es importante saber el tiempo de ejecución promedio sobre el subconjunto de instancias de medida n. Generalmente el análisis del tiempo de ejecución promedio es más dificíl que cuando se considera el peor caso. Por otro lado, para estudiar el comportamiento promedio se debe conocer la distribución de las instancias que se resolverán y en muchas aplicaciones esto no es posible.

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 referiendo vamos a obviar el subíndice A, utilizando $T(n)$ en lugar de $T_A(n)$.

2.4. Notación asintótica: $\mathcal{O}$

Esta notación es llamada ***asintótica*** porque expresa el comportamiento de la función en el límite, esto es, para valores suficientemente grandes de los parámetros. Aunque las conclusiones basadas en la notación asintótica pueden fallar cuando los parámetros toman valores *pequeños*, un algoritmo mejor asintóticamente, casi siempre tiene performance muy superior ya en instancias de medida moderada. Esto hace importante su estudio.

Formalmente, dadas dos funciones $T, g : \mathbb{N} \to \mathbb{R}$, decimos que:

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

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

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

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

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

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

- $T(n)$ es $\Theta(g(n))$ si

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

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

Obs: $T(n)$ es $\mathcal{O}(g(n))$ si y sólo si $g(n)$ es $\Omega(T(n))$.

Veamos un ejemplo:

Ejemplo 4. $2n^2 + 10n$ *es* $\mathcal{O}(n^2)$*, porque tomando* $n_0 = 0$ *y* $c = 12$*, tenemos que*

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

También podríamos decir que $2n^2 + 10n$ es $\mathcal{O}(n^3)$, cosa que es cierta. Sin embargo, nos interesa calcular de la mejor manera posible el orden de una función, es decir de la forma más ajustada posible.

Entonces, 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 de orden de n^2, es decir, $\mathcal{O}(n^2)$. El uso de microsegundos es totalmente irrelevante, ya que sólo necesitamos cambiar la constante para acotar el tiempo por años o nanosegundos.

Ejemplo 5. 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.*

Ejemplo 6. *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* $\Omega(\log_b(n))$*.*

Las definiciones de $\mathcal{O}$, Ω y Θ piden que existan las constantes $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$, pero no son relevantes sus valores específicos (que además no son únicos), lo relevante es que existan. Sabemos, por definición, que $T(n)$ es $\mathcal{O}(g(n))$ si existen $c \in \mathbb{R}_+$ y $n_0 \in \mathbb{N}$ tal que $\frac{T(n)}{g(n)} \leq c$. Esto sugiere relacionar la definición de $\mathcal{O}$ con el cálculo de límites.

Si $\lim_{n\to\infty} \frac{T(n)}{g(n)} = a$ 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 $\lim_{n\to\infty} \frac{T(n)}{g(n)} \in [0, \infty)$.
- $T(n)$ es $\Omega(g(n))$ si y sólo si $\lim_{n\to\infty} \frac{T(n)}{g(n)} \in (0, \infty]$.
- $T(n)$ es $\Theta(g(n))$ si y sólo si $\lim_{n\to\infty} \frac{T(n)}{g(n)} \in (0, \infty)$.

Esta equivalencia muchas veces puede ser muy útil para simplificar las demostraciones, como en los siguientes ejemplos. El primer caso, demuestra que cualquier función exponencial es *peor* que cualquier función polinomial.

Ejemplo 7. *Si $k, d \in \mathbb{N}$ entonces k^n no es $\mathcal{O}(n^d)$, con $k \geq 2$.*

Por la propiedad anterior, sabemos que k^n es $\mathcal{O}(n^d)$ si y sólo si $\lim_{n\to\infty} \frac{k^n}{n^d} \in [0, \infty)$.

Aplicando l'Hôpital, podemos ver que $\lim_{n\to\infty} \frac{k^n}{n^d} = \infty$. Por lo tanto, k^n no es $\mathcal{O}(n^d)$

Y ahora, mostraremos que la función logarítmica es *mejor* que la función lineal (ya vimos que no importa la base), es decir $ln(n) \prec O(n)$.

Ejemplo 8. *$ln(n)$ es $\mathcal{O}(n)$ y n no es $\mathcal{O}(ln(n))$.*

Nuevamente, operando y aplicando l'Hôpial, podemos ver que $\lim_{n\to\infty} \frac{\ln(n)}{n} = 0$, que es lo mismo que $\lim_{n\to\infty} \frac{n}{\ln(n)} = \infty$.

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).

Si un algoritmo es $\mathcal{O}(n)$, se dice *lineal*. En el caso que sea $\mathcal{O}(n^2)$, se dice *cuadrático*. Similarmente, un algoritmo es *logarítmico*, *cúbico*, *polinomial* o *exponencial* si son $\mathcal{O}(\log n)$, $\mathcal{O}(n^3)$, $\mathcal{O}(n^k)$ o $\mathcal{O}(d^n)$ respectivamente, donde k y d son constantes apropiadas.

Comúnmente son ignoradas las constantes multiplicativas (las c) y se asume que son todas del mismo orden de magnitud. Por eso, se suele decir que un algoritmo lineal es más rápido que uno cuadrático, sin considerar que esto puede no ser cierto para algunos casos. En algunos casos puede ser necesario ser más cuidadoso en el análisis.

Por ejemplo, consideremos dos algoritmos para resolver el mismo problema. Uno de ellos necesita n^2 días y el otro n^3 segundos para resolver una instancia de tamaño n. 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 aunque el algoritmo cuadrático es asintóticamente mejor, su constante multiplicativa 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 sea adecuado.

Ejemplo 9. *Las complejidades de algunos algoritmos conocidos son:*

- *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.

2.5. Algoritmos eficientes vs no eficientes

Para comparar algunas posibles complejidades, en la tabla 1 mostramos los tiempos insumidos por algoritmos con esas complejidades cuando son ejecutados sobre instancias de distintos tamaños por la misma máquina (suponiendo 0.001 miliseg por operación).

	10	20	30	40	50	60
$\log(n)$	0.000001 seg	0.000001 seg	0.000001 seg	0.000002 seg	0.000002 seg	0.000002 seg
n	0.00001 seg	0.00002 seg	0.00003 seg	0.00004 seg	0.00005 seg	0.00006 seg
$n \log(n)$	0.00001 seg	0.000026 seg	0.000044 seg	0.000064 seg	0.000085 seg	0.0001 seg
n^2	0.0001 seg	0.0004 seg	0.0009 seg	0.0016 seg	0.0025 seg	0.0036 seg
n^3	0.001 seg	0.008 seg	0.027 seg	0.064 seg	0.125 seg	0.216 seg
n^5	0.1 seg	3.2 seg	24.3 seg	1.7 min	5.2 min	13.0 min
2^n	0.001 seg	1.0 seg	17.9 min	12.7 días	35.6 años	366 siglos
3^n	0.59 seg	58 min	6.5 años	3855 siglos	2 E+8 siglos	1.3 E+13 siglos
$n!$	3.63 seg	771 siglos	8.4 E+16 siglos	2.5 E+32 siglos	9.6 E+48 siglos	2.6 E+66 siglos

Cuadro 1: Complejidad vs tamaño de la entrada

Los datos de la tabla 1 corresponden a una máquina muy vieja (datos del libro de Garey y Johnson de 1979). ¿Qué pasa si tenemos una máquina 1000 veces más rápida? ¿Un millón de veces más rápida? ¿Cuál sería el tamaño del problema que podemos resolver en una hora comparado con el problema que podemos resolver ahora? En la tabla 2 comparamos el tamaño de las instancias que podríamos resolver en 1 hora con una máquina 1000 veces más rápida para distintas complejidades algorítmicas.

	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

Cuadro 2: Comparación de tamaño de instancias resueltas en 1 hora

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

Esto sugiere el siguiente criterio:

POLINOMIAL = "bueno"

EXPONENCIAL = "malo"

2.6. 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 resolverlos, 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.

3. Resumiendo

- Vamos a describir nuestro 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.
- 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.

4. Más ejemplos

Ejemplo 10. *Cálculo del promedio de un arreglo:*

```
promedio(A)
    entrada: arreglo de reales A de tamaño ≥ 1
    salida: promedio de los elementos de A

    suma ← 0                                        O(1)
    para i = 0 hasta dim(A) − 1 hacer               dim(A) veces
        suma ← suma + A[i]                          O(1)
    fin para
    retornar suma/dim(A)                            O(1)
```

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

Ejemplo 11. *Algoritmo de búsqueda:*

```
esta?(A,elem)
    entrada: arreglo de enteros A no vacío
    salida: Verdadero si el entero elem se encuentra en A, Falso en caso contrario

    i ← 0                                                    O(1)
    mientras i < dim(A) y elem ≠ A[i] hacer                  a lo sumo dim(A) veces
        i ← i + 1                                            O(1)
    fin mientras
    si i < dim(A) entonces                                   O(1)
        retornar Verdadero                                   O(1)
    sino
        retornar Falso                                       O(1)
```

Este algoritmo es lineal en función del tamaño de la entrada.

Ejemplo 12. *Invertir los dígitos de un entero:*

```
invertir_entero(n)
    entrada: entero n ≥ 0
    salida: número formado por los dígitos de n invertidos

    resu ← 0                                                 O(1)
    aux ← n                                                  O(1)
    mientras aux ≠ 0 hacer                                   log10(n) veces
        resu ← 10 * resu + aux mod 10                        O(1)
        aux ← aux div 10                                     O(1)
    fin mientras
    retornar resu                                            O(1)
```

El tamaño de la entrada es $t = \log_2 n$. El algoritmo es orden $3 * O(1) + \log_{10} n * 2 * O(1)$, que es $\mathcal{O}(\log_{10} n) = O(\log_2 n) = O(t)$. El algoritmo es lineal en el tamaño de la entrada.

Ejemplo 13. *Sean m y n dos enteros positivos. El máximo común divisor (mcd) de m y n es el entero más grande que divide a ambos. El algoritmo obvio para calcular el (mcd) se desprende de la definición:*

```
mcd(m,n)
    entrada: m, n ∈ ℕ+
    salida: mcd de m y n

    i ←min(m,n)+1                                   O(???)
    repetir                                         a lo sumo mín(m,n) veces
        i ← i − 1                                   O(1)
    hasta m mod i = 0 y n mod i = 0                 O(1)
    retornar resu                                   O(1)
```

Para poder calcular la complejidad del algoritmo anterior, necesitamos conocer la complejidad de la función min.

```
min(m,n)
    entrada: m, n ∈ ℕ+
    salida: mcd de m y n

    si m < n entonces                               O(1)
        retornar m                                  O(1)
    sino
        retornar n                                  O(1)
```

El tamaño de la entrada es $t = O(log_2(\text{máx}\{m,n\}))$. *Como la función min es* $\mathcal{O}(1)$, *el orden de mcd es* $2 * O(1) + \text{mín}\{m,n\} * 2 * O(1)$, *que es* $\mathcal{O}(\text{mín}\{m,n\})$. *Si m y n son del mismo orden, el algoritmo es* $\mathcal{O}(2^t)$, *exponencial en el tamaño de la entrada.*

Ejemplo 14. *Algoritmo de ordenamiento (selection sort):*

```
selection(A)
    entrada: A arreglo de reales
    salida: A ordenado

    para i = 0 hasta dim(A) − 2 hacer               dim(A) − 1 veces
        minj ← i                                    O(1)
        minval ← A[i]                               O(1)
        para j = i + 1 hasta dim(A) − 1 hacer       a lo sumo dim(A) veces
            si A[j] < minval entonces               O(1)
                minj ← j                            O(1)
                minval ← A[j]                       O(1)
            fin si
        fin para
        A[minj] ← A[i]                              O(1)
        A[i] ← minval                               O(1)
    retornar A                                      O(dim(A))
```

Si $dim(A) = n$, *el algoritmo es orden* $n * (4 * O(1) + n * 3 * O(1) + 1)$, *que es* $\mathcal{O}(n^2)$, *es decir, cuadrático en función*

del tamaño de la entrada.

Ejemplo 15. *Primalidad: Dado un número $n \in \mathbb{N}$, ¿n es primo?*

```
esPrimo(n)
    entrada: n ∈ ℕ
    salida: Verdadero si n es primo, Falso en caso contrario

    i ← 2                                          O(1)
    mientras i ≤ √n hacer                          √n veces - O(1)
        si n mod i = 0 hacer                       O(1)
            retornar Falso                         O(1)
        fin si
    fin mientras
    retornar Verdadero                             O(1)
```

El tamaño de la entrada es $t = \log_2(n)$. Bajo el modelo de costo uniforme suponemos que todas las operaciones son $\mathcal{O}(1)$. Por lo tanto la cantidad de operaciones totales es $\mathcal{O}(\sqrt{n}) = O(\sqrt{(2^t)}) = O(\sqrt{2}^t)$. Notar que se conocen algoritmos polinomiales para saber si un entero es primo.

Ejemplo 16. *Cálculo iterativo del n-ésimo número de Fibonacci, $Fib(n)$:*

```
Fibonacci(n)
    entrada: entero n ≥ 1
    salida: n-ésimo número de Fibonacci

    i ← 1                                          O(1)
    j ← 0                                          O(1)
    para k = 0 hasta n − 1 hacer                   n veces
        j ← i + j                                  O(??)
        i ← j − i                                  O(??)
        k ← k + 1                                  O(1)
    fin para
    retornar j                                     O(log2(Fib(n))) = O(??)
```

Como hacían en Algo1, pueden demostrar que el invariante del ciclo asevera que j_k es $Fib(k)$ y i_k es $Fib(k-1)$, donde j_k e i_k son los valores de las variables j e i en la iteración k-ésima. Moivre encontró una fórmula cerrada para calcular los número de Fibonacci: $Fib(k) = \frac{(1+\sqrt{5})^k - (1-\sqrt{5})^k}{2^k\sqrt{5}}$. Por lo tanto, en la iteración k-ésima, los operandos de $j \leftarrow i + j$ y $i \leftarrow j - i$ son $\mathcal{O}(2^k)$, requiriendo $\mathcal{O}(log_2(2^k)) = O(k)$ bits para almacenarlos.

Esto muestra que los operandos de las operaciones intermedias crecen exponencialmente, por lo que es necesario utilizar el modelo logarítmico para realizar un análisis realista. Entonces, el tiempo requerido por las operaciones es proporcional al tamaño de los operandos (cantidad de bits que ocupan), que en este caso es $\mathcal{O}(k)$ en la iteración k-ésima.

Como $k \leq n$, podemos suponer que el valor de los operandos son $\mathcal{O}(2^n)$ y por lo tanto su tamaño $\mathcal{O}(n)$. Esto implica que las operaciones $j \leftarrow i + j$ y $i \leftarrow j - i$ son $\mathcal{O}(n)$, resultando el algoritmo $\mathcal{O}(n^2)$. Como el tamaño de la entrada

es $t = log_2(n)$, el algoritmo es $\mathcal{O}(4^t)$, exponencial en el tamaño de la entrada.

Ejemplo 17. *Cálculo iterativo de $n!$:*

```
factorial(n)
    entrada: n ∈ Z_{≥0}
    salida: n!

    resu ← 1                                   O(1)
    para k = 2 hasta n hacer                   n − 1 veces
        resu ← resu * k                        O(??)
    fin para
    retornar resu                              O(log2(n!)) = O(n * log2(n))
```

Al ingresar a la k-ésima iteración del ciclo, el valor de la variable resu es $(k-1)!$. Esto muestra que los operandos de las operaciones intermedias crecen factorialmente, por lo que es necesario utilizar el modelo logarítmico para realizar un análisis realista, donde el tiempo requerido por las operaciones es proporcional al tamaño de los operandos (cantidad de bits que ocupan).

En la iteración k-ésima, el operando resu de $resu \leftarrow resu * k$ requiere $log_2((k-1)!)$ bits para almacenarlos. Podemos acotar esto, $log_2((k-1)!) \leq log_2(k!) \leq log_2(k^k) = k * log_2(k)$. El otro operando es k, que requiere $log_2(k)$ bits.

Entonces, la operación $resu \leftarrow resu * k$ de la iteración k-ésima es $\mathcal{O}(k * log_2(k) log_2(k))$ suponiendo un algoritmo de multiplicación ingenuo, y como $k \leq n$ esto es $\mathcal{O}(n * log_2^2(n))$. Como esta operación está dentro de un ciclo que se ejecuta $n-1$ veces, la complejidad del algoritmo es $\mathcal{O}(n^2 log_2^2(n))$.

Como el tamaño de la entrada es $t = log_2(n)$, el algoritmo es $\mathcal{O}(t^2 4^t)$, exponencial en el tamaño de la entrada.

5. Bibliografía recomendada

- Capítulos 1, 2 y 3 de G. Brassard and P. Bratley, *Fundamental of Algorithmics*, Prentice-Hall, 1996.
- Secciones I.1 y I.2 de T. Cormen, C. Leiserson, R. Rivest, C. Stein, *Introduction to Algorithms*, The MIT Press, McGraw-Hill, 2001.

Referencias

[1] S. A. Cook and R. A. Reckhow. Time bounded random access machines. 7(4):354–375, Aug. 1973.