3.1. Algoritmos

Formalmente ¿qué es un algoritmo? no es una pregunta fácil de responder, de hecho, no daremos una definición formal, apelaremos a la intuición que el alumno tenga hasta el momento. Intuitivamente podemos decir que un algoritmos es un *método* o *conjunto de instrucciones* que sirven para resolver cierto problema. La duda que puede surgir es ¿qué clase de problemas pueden ser resueltos por un algoritmo? Claramente no existe un método para solucionar el problema del hambre en el mundo, o el problema de cómo una persona puede llegar a fin de mes sin deudas. Diremos entonces que los problemas que nos interesan son los *problemas computacionales*, más adelante definiremos formalmente qué se entiende por problema computacional. Por ahora nos limitaremos a decir que un problema computacional se plantea en forma de INPUT y OUTPUT: dada una representación de datos de entrada válida INPUT queremos obtener una representación de datos de salida OUTPUT que depende del INPUT y que cumple ciertas condiciones. Con estos conceptos podemos decir que un algoritmo es un método para convertir un INPUT válido en un OUTPUT. ¿Qué significa que un INPUT sea válido? eso dependerá completamente del problema en cuestión. A este método le exigiremos ciertas propiedades:

- Precisión: cada instrucción debe ser planteada en forma precisa y no ambigua.
- Determinismo: cada instrucción tiene un único comportamiento que depende solamente del input.
- Finitud: el método está compuesto por un conjunto finito de instrucciones.

Diremos que las instrucciones del algoritmo se *ejecutan*[1] y que el algoritmo se detiene si no hay más instrucciones que ejecutar o si ya se produjo un OUTPUT. Un concepto importantísimo es cuándo consideraremos que un algoritmo es correcto. En nuestro caso diremos que un algoritmo es correcto si para todo INPUT válido luego de comenzar la ejecución del algoritmo, este se detiene y produce un OUTPUT correcto para el INPUT. Una implicación importante es que de aquí se deduce cuándo un algoritmo es incorrecto: si existe al menos un INPUT válido para el cuál, o el algoritmo no se detiene, o calcula un OUPUT incorrecto, entonces diremos que el algoritmo es incorrecto.

El alumno debe estar acostumbrado a los ejemplos tipo "receta de cocina" muy comentados en cursos introductorios de computación. Nos evitaremos este tipo de ejemplos e introduciremos nuestra notación de algoritmo con un problema más cercano a lo que nos interesará en este curso. Para representar un algoritmo usaremos seudo–código, muy parecido a lo que sería un código en un lenguaje como C o Java pero que no se preocupa de problemas de sintaxis y en algunos casos nos permitirá trabajar de manera simple con conjuntos, operaciones matemáticas, etc.

Ejemplo: Queremos un algoritmo para, dada una secuencia $S = (s_1, s_2, \ldots, s_n)$ de n números enteros de input, genere como output el valor máximo de esa secuencia. El siguiente es un algoritmo "iterativo" que resuelve el problema planteado:

INPUT: Una secuencia $S = (s_1, s_2, \ldots, s_n)$ y un natural $n \geq 1$ que representa el largo de la secuencia.
OUTPUT: $m = \text{máx}\{s_1, s_2, \ldots, s_n\}$, el máximo de los números de la secuencia.

MAX(S, n)
1 $m := s_1$
2 $k := 2$
3 **while** $k \leq n$ **do**
4 **if** $s_k > m$ **then**
5 $m := s_k$
6 $k := k + 1$
7 **return** m

[1] Cuando hablemos de la ejecución de las instrucciones de un algoritmo el sujeto siempre será el algoritmo mismo, diremos que él ejecuta las instrucciones.

El siguiente es un algoritmo "recursivo" que resuelve el problema planteado:

INPUT: Una secuencia $S = (s_1, s_2, \ldots, s_n)$ un natural $n \geq 1$ que representa el largo de la secuencia.
OUTPUT: $m = \text{máx}\{s_1, s_2, \ldots, s_n\}$, el máximo de los números de la secuencia.

```
REC-MAX(S, n)
1  if n = 1 then
2      return s_1
3  else
4      k := REC-MAX(S, n − 1)
5      if s_n ≥ k then
6          return s_n
7      else
8          return k
```

En el ejemplo se han introducido algunas instrucciones que debieran resultar familiares, **while**, **if–else**, **return**, que tienen el sentido habitual de un lenguaje de programación de propósito general, hemos usado el operador ":=" para la asignación, y a veces usaremos también la instrucción **for**. También se ha introducido la forma de hacer recursión. Los algoritmos recursivos en general tienen relación directa con definiciones inductivas de objetos (como vimos en la sección 1.1). Supondremos que el alumno tiene cierta familiaridad también con algoritmos recursivos. En el algoritmo recursivo anterior se está aprovechando la definición inductiva del máximo de una secuencia de elementos:

1. $\text{máx}\{s_1, s_2, \ldots, s_n\} = \text{máx}\{\text{máx}\{s_1, s_2, \ldots, s_{n-1}\}, s_n\}$
2. $\text{máx}\{s_1\} = s_1$.

En general los algoritmos recursivos se obtendrán casi en forma directa de definiciones inductivas de objetos o propiedades.

¿Cómo podemos asegurar que los anteriores algoritmos son correctos, o sea, que siempre se detienen y entregan el resultado esperado? La forma en cómo se diseñaron nos da una intuición de su *corrección*, pero necesitamos métodos más formales para establecer la corrección de un algoritmo.

3.1.1. Corrección de Algoritmos

La herramienta principal que usaremos para demostrar la corrección de los algoritmos será la inducción en sus distintas formulaciones. Dividiremos nuestro estudio en dos partes, la corrección de algoritmos iterativos y la corrección de algoritmos recursivos.

Corrección de Algoritmos Iterativos

Supongamos que queremos demostrar que un algoritmo no recursivo (o sea, uno en que no hay llamadas recursivas a sí mismo) es correcto. La única dificultad proviene de la posibilidad de que el programa contenga *loops*, o sea que realice iteraciones, por lo que nos centramos en este caso[2]. Generalmente, dividimos la demostración de que un programa iterativo es correcto en dos tareas independientes, que llamamos **corrección parcial** y **terminación** que se definen de la siguiente forma. Para demostrar la corrección parcial de un algoritmo debemos establecer que, si el algoritmo se detiene, entonces entrega un resultado correcto. Para la

[2]No nos preocuparán algoritmos no recursivos que no tengan iteraciones ya que en ellos se puede hacer un trazado completo de su ejecución viendo todos los casos y la corrección resulta trivial

terminación debemos demostrar que el algoritmo efectivamente se detiene. Introduciremos estos conceptos con un ejemplo.

El siguiente es un algoritmo que recibe como input un par de números naturales x e y, analizaremos luego cuál es el output del algoritmo.

INPUT: Dos números $x, y \in \mathbb{N}$.
OUTPUT: $z = ?$.

M(x, y)
1 $z := 0$
2 $w := y$
3 **while** $w \neq 0$ **do**
4 $\quad z := z + x$
5 $\quad w := w - 1$
6 **return** z

En este algoritmo existe un único *loop*. Para demostrar la corrección parcial de un algoritmo, lo que generalmente se hace es buscar un **invariante** del *loop*, una expresión lógica que sea verdadera antes de iniciar el *loop* y al final de cada una de las iteraciones del *loop*. Esta expresión siempre tendrá relación con las variables del algoritmo y generalmente es una expresión que "une a las variables", las hace depender unas de otras. Si encontramos una expresión y logramos demostrar que esta es efectivamente un invariante, entonces esta expresión será verdadera también al finalizar el *loop*, luego de su última iteración, lo que nos permitirá decir que, si el *loop* se detiene entonces la propiedad se cumple. Dado un *loop* pueden existir muchas propiedades que siempre sean verdaderas antes de iniciar y al final de cada iteración, debemos elegir la que más información nos entregue con respecto a lo que el algoritmo pretende realizar. Por ejemplo, en el anterior algoritmo está claro que la expresión:

$$0 \leq w \leq y$$

es un invariante del loop, el problema principal es que no nos dice nada por ejemplo de los valores de las variables x y z que también están participando del algoritmo.

¿Cómo buscamos entonces un invariante que nos ayude a demostrar la corrección parcial? En general debemos centrarnos en buscar propiedades que tengan que ver con lo que queremos que el algoritmo haga. ¿Qué hace el anterior algoritmo? No es difícil notar que el algoritmo anterior, en base a sumas y decrementos, calcula la multiplicación entre x e y, por lo que al finalizar el algoritmo se debiera cumplir que $z = x \cdot y$. Para estar más seguros podemos hacer una tabla con los valores que toman cada variable después de cada iteración. La siguiente es la tabla mencionada si se supone que los valores de x e y son inicialmente a y b, la columna 0 corresponde a los valores antes de que comience el *loop* y la columna i a los valores al finalizar la i–ésima iteración.

variables	0	1	2	3	4	...	b
x	a	a	a	a	a	$\cdots$	a
y	b	b	b	b	b	$\cdots$	b
z	0	a	$2a$	$3a$	$4a$	$\cdots$	$b \cdot a$
w	b	$b-1$	$b-2$	$b-3$	$b-4$	$\cdots$	0

El *loop* alcanza a ejecutar b iteraciones luego de lo cuál el valor de w se hace 0 y se termina con $z = b \cdot a$ que era lo que esperábamos. Esta tabla nos permite inferir también que la siguiente expresión debiera cierta luego de cada iteración del *loop*:

$$z = (y - w)x$$

Si demostráramos que la expresión anterior es un invariante entonces estaríamos seguros que después de la última iteración del *loop* se cumple que $z = (y-w)x$, y dado que el *loop* se detiene sólo si $w = 0$ obtendríamos que, si el algoritmo se detiene entonces $z = y \cdot x$. Para demostrar que $z = (y-w)x$ es un invariante usaremos el principio de inducción simple, en el siguiente lema.

Lema 3.1.1: La expresión $z = (y-w)x$ es un invariante para el loop del algoritmo $\mathrm{M}(x, y)$, o sea, si $x, y \in \mathbb{N}$ y se ejecuta el algoritmo $\mathrm{M}(x, y)$ entonces luego de cada iteración se cumple que $z = (y - w)x$.

Demostración: Las únicas variables que cambian sus valores a medida que el algoritmo se ejecutan son z y w, luego hay que centrarse en la evolución de los valores de estas variables. Sean z_i y w_i los valores que almacenan las variables z y w luego de que se termina la i–ésima iteración del algoritmo, z_0 y w_0 son los valores iniciales. Demostraremos por inducción que para todo i se cumple que $z_i = (y - w_i)x$.

B.I. Para $i = 0$, se tiene que $z_i = z_0 = 0$ y que $w_i = w_0 = y$. Ahora, $z_0 = 0 = (y - y)x = (y - w_0)x$ por lo que se cumple que $z_i = (y - w_i)x$ para $i = 0$.

H.I. Supongamos que efectivamente se cumple que $z_i = (y - w_i)x$.

T.I. Al final de la iteración $i+1$ el valor de z se ha incrementado en x y el valor de w se ha decrementado en 1 por lo que se cumple la relación $z_{i+1} = z_i + x$ y $w_{i+1} = w_i - 1$. Ahora

$$z_{i+1} = z_i + x \overset{HI}{=} (y - w_i)x + x = (y - w_i + 1)x = (y - (w_i - 1))x = (y - w_{i+1})x$$

por lo que la propiedad también se cumple para $i + 1$.

Por inducción entonces se ha demostrado que después de cada iteración se cumple que $z = (y - w)x$ y por lo tanto si el algoritmo termina, el resultado será $z = y \cdot x$ □

Hemos demostrado la corrección parcial del algoritmo, ahora debemos demostrar que el algoritmo efectivamente termina, generalmente esta parte es más simple de demostrar, y casi siempre basta con encontrar una expresión entera que decrezca o se incremente con cada iteración y argumentar que llegado a cierto valor en la sucesión el algoritmo se detendrá El siguiente lema demuestra la terminación del anterior algoritmo.

Lema 3.1.2: Si $x, y \in \mathbb{N}$ y se ejecuta el algoritmo $M(x, y)$ entonces este se detiene.

Demostración: Note que el valor de w es inicialmente $y \in \mathbb{N}$ y después de cada iteración se decrementa en 1. Los valores de w forman entonces una sucesión siempre decreciente de valores consecutivos que se inicia en un número natural, por lo que en algún momento w tomará el valor 0 (ya que 0 es el menor de los números naturales) y por lo tanto el *loop* terminará y también el algoritmo. □

Finalmente con los dos lemas anteriores se demuestra directamente el siguiente teorema.

Teorema 3.1.3: El algoritmo $M(x, y)$ retorna el valor $x \cdot y$ (cuando $x, y \in \mathbb{N}$).

Demostración: Directa de los dos lemas anteriores. □

Podemos usar esta misma estrategia para demostrar que el algoritmo iterativo $\textsc{Max}(S, n)$ es también correcto, o sea, que después de ejecutarlo con una secuencia S de valores y un natural $n \in \mathbb{N}$ correspondiente a la cantidad de valores de la secuencia, el algoritmo retorna el máximo de la secuencia S. En el siguiente ejemplo se discuten algunos aspectos de esta demostración.

Ejemplo: Demostraremos que si el algoritmo $\textsc{Max}(S, n)$ del principio de esta sección se ejecuta con una secuencia S de n valores $s_1, s_2, \ldots, s_n$ entonces retorna el máximo de la secuencia. Nos evitaremos la formalidad de establecer teoremas y lemas para la demostración.

Lo primero es notar que el algoritmo termina, su justificación se deja como ejercicio. Ahora debemos justificar que si el algoritmo termina calcula el resultado correcto. Debemos encontrar un invariante del *loop* del algoritmo, una expresión que siempre se mantenga verdadera. Dado que las únicas variables que se modifican durante la ejecución son m y k debemos encontrar una invariante que incluya a estas dos variables y a los valores de la secuencia S. Proponemos el siguiente invariante: en todo momento se cumple que

$$m \in S \text{ y } m \geq s_1, s_2, \ldots, s_{k-1} \tag{3.1}$$

o sea, en todo momento m es mayor o igual a los $k-1$ elementos iniciales de S. Nótese que si demostramos que (3.1) es efectivamente un invariante entonces, dado que al terminar el algoritmo el valor de k es $n+1$, sabríamos que al terminar el algoritmo m es efectivamente el máximo. La primera parte del invariante ($m \in S$) es simple de justificar ya que basta mirar el código para notar que cada asignación posible a m es con uno de los valores s_i. La que es un poco más complicada es la segunda parte que la demostraremos por inducción. Sean m_i y k_i los valores de las variables m y k al finalizar la i–ésima iteración del *loop*, por inducción demostraremos que para todo i se cumple que

$$m_i \geq s_1, s_2, \ldots, s_{k_i-1}$$

B.I. Para $i = 0$ (o sea antes de comenzar la primera iteración del *loop*) se cumple que $m_i = m_0 = s_1$ y que $k_i = k_0 = 2$, por lo que $m_i = s_1 \geq s_1 = s_{2-1} = s_{k_i-1}$.

H.I. Supongamos que al final de la iteración i efectivamente se cumple que $m_i \geq s_1, s_2, \ldots, s_{k_i-1}$.

T.I. Al final de la iteración $i+1$ el valor de k se incrementa en 1 por lo que $k_{i+1} = k_i + 1$. Ahora el valor de m se modifica dependiendo del resultado del **if** de las líneas 4–5, dependiendo de su comparación con el valor s_k. Algo muy importante de notar es que al momento de la comparación en el **if**, los valores de m y k que se comparan son los correspondientes a la iteración anterior (i en nuestro caso). Tendremos entonces dos casos dependiendo del resultado del **if**: (1) $m_{i+1} = m_i$ si $m_i \geq s_{k_i}$, o (2) $m_{i+1} = s_{k_i}$ si $s_{k_i} > m_i$. Dividiremos entonces la demostración:

(1) Si $m_i \leq s_{k_i}$, entonces al final de la iteración $i+$ sabemos que $m_{i+1} = m_i$ y $k_{i+1} = k_i + 1$. Por HI sabemos que $m_i \geq s_1, s_2, \ldots, s_{k_i-1}$ y como además sabemos que $m_i \leq s_{k_i}$ podemos concluir que $m_i \geq s_1, s_2, \ldots, s_{k_i-1}, s_{k_i}$. Ahora, dado que en este caso $m_{i+1} = m_i$ obtenemos que

$$m_{i+1} = m_i \geq s_1, s_2, \ldots, s_{k_i-1}, s_{k_i}.$$

Cómo también sabemos que $k_{i+1} = k_i + 1$, se cumple que $s_{k_i} = s_{k_{i+1}-1}$, así finalmente obtenemos que

$$m_{i+1} \geq s_1, s_2, \ldots, s_{k_{i+1}-1}$$

que es lo que queríamos demostrar.

(2) Si $s_{k_i} > m_i$, entonces al final de la iteración $i+1$ sabemos que $m_{i+1} = s_{k_i}$ y $k_{i+1} = k_i + 1$. Por HI sabemos que $m_i \geq s_1, s_2, \ldots, s_{k_i-1}$ y como además sabemos que $s_{k_i} > m_i$ podemos concluir que obtenemos que $s_{k_i} \geq s_1, s_2, \ldots, s_{k_i-1}$. Ahora, dado que en este caso $m_{i+1} = s_{k_i}$ y es claro que $s_{k_i} \geq s_{k_i}$ obtenemos que

$$m_{i+1} = s_{k_i} \geq s_1, s_2, \ldots, s_{k_i-1}, s_{k_i}.$$

Cómo también sabemos que $k_{i+1} = k_i + 1$, se cumple que $s_{k_i} = s_{k_{i+1}-1}$, así finalmente obtenemos que

$$m_{i+1} \geq s_1, s_2, \ldots, s_{k_{i+1}-1}$$

que es lo que queríamos demostrar.

Finalmente la propiedad también se cumple para $i+1$.

Por inducción hemos demostrado que (3.1) es efectivamente un invariante para el algoritmo $\textsc{Max}(S, n)$ y que por lo tanto siempre se cumple que $m \in S$ y $m \geq s_1, s_2, \ldots, s_{k-1}$. Ahora si el algoritmo termina el valor de k es $n+1$ por lo que se cumpliría que

$$m \in S \text{ y } m \geq s_1, s_2, \ldots, s_n$$

y por lo tanto m es el máximo. Cómo antes demostramos que el algoritmo terminaba, podemos decir que $\textsc{Max}(S, n)$ correctamente calcula el máximo de la secuencia S.

Corrección de Algoritmos Recursivos

Para los algoritmos recursivos generalmente no se hace la división entre corrección parcial y terminación, simplemente se demuestra la propiedad deseada por inducción. La inducción es en general en función del tamaño del input. La hipótesis de inducción siempre dirá algo como "si el algoritmo se ejecuta con un input de tamaño i entonces es correcto[3]", y a partir de esa hipótesis intentamos demostrar que "si el algoritmo se ejecuta con un input de tamaño $i+1$ también será correcto". Obviamente el argumento también se puede plantear por inducción por curso de valores: como hipótesis tomaremos que "si el algoritmo se ejecuta con un input de tamaño menor que i entonces es correcto" y a partir de ella intentamos demostrar que "si el algoritmo se ejecuta con un input de tamaño i también será correcto". En cualquier caso siempre se tendrá que tomar en cuenta el caso base por separado. Mostraremos el método con un ejemplo.

Ejemplo: Demostraremos que el algoritmo Rec-Max(S, n) está correcto, o sea, efectivamente retorna el máximo de una secuencia $S = s_1, s_2, \ldots, s_n$ de largo n. La demostración la haremos directamente por inducción en el tamaño del input, en este caso la inducción se hará en el largo i de la secuencia. Demostraremos que para todo valor i se cumple que el resultado retornado por Rec-Max(S, i) es el máximo de los valores $\geq s_1, s_2, \ldots, s_i$.

B.I. La base en este caso es $i = 1$. Si se ejecuta Rec-Max(S, i) con $i = 1$ el resultado será s_1 por lo que el resultado será el máximo de la secuencia compuesta sólo por s_1 y se cumple la propiedad.

H.I. Supongamos que si se ejecuta Rec-Max(S, i) el resultado será el máximo de los valores $s_1, s_2, \ldots, s_i$.

T.I. Queremos demostrar que si se ejecuta Rec-Max$(S, i+1)$ el resultado será el máximo de los valores $s_1, s_2, \ldots, s_n, s_{i+1}$. Si se ejecuta Rec-Max$(S, i+1)$ solo tenemos que preocuparnos de las líneas 4 en adelante (ya que el $i+1$ no es igual a 1). En la línea 4 se llama a Rec-Max(S, i) $(i = (i+1)-1)$ que por HI es correcto, o sea, luego de la ejecución de la línea 4, el valor de k será igual al máximo entre los valores $s_1, s_2, \ldots, s_i$. Ahora tenemos dos posibilidades, que se ejecute la línea 6 o la línea 8 dependiendo del resultado del **if** de la línea 5:

(1) Si se ejecuta la línea 6, es porque se cumple que s_{i+1} es mayor o igual a k, o sea s_{i+1} es mayor o igual al máximo de los valores $s_1, s_2, \ldots, s_i$, luego si esto ocurre, sabemos que s_{i+1} es el máximo de los valores $s_1, s_2, \ldots, s_{i+1}$ y por lo tanto Rec-Max$(S, i+1)$ entregaría un resultado correcto, ya que se retorna s_{i+1}.

(2) Si se ejecuta la línea 8, es porque se cumple que s_{i+1} es menor estricto que k, por lo que el máximo de la secuencia $s_1, s_2, \ldots, s_{i+1}$ sería simplemente k, luego Rec-Max$(S, i+1)$ entregaría un resultado correcto ya que retorna k.

Por inducción demostramos que para todo i se cumple que el resultado retornado por Rec-Max(S, i) es el máximo de los valores $\geq s_1, s_2, \ldots, s_i$ y por lo tanto Rec-Max(S, i) es correcto.

3.1.2. Notación Asintótica

En la sección anterior estudiamos la corrección de algoritmos que se refería a determinar si un algoritmo es o no correcto. Sin embargo, a pesar de que un algoritmo sea correcto puede ser poco útil en la práctica porque, al implementar el algoritmo en algún lenguaje de programación de propósito general como C o JAVA, el tiempo necesario para que el algoritmo termine puede ser demasiado alto. Necesitamos entonces estimaciones del tiempo que le tomará a un algoritmo ejecutar en función del tamaño de los datos de entrada, generalmente, a medida que el tamaño de los datos de entrada crece el algoritmo necesitará más tiempo para ejecutar. Estas estimaciones deben ser lo más independientes del lenguaje en el que lo implementemos, del compilador que usemos e idealmente independientes de las características del hardware que utilicemos para

[3] correcto en el sentido de que entrega el resultado que efectivamente debiera entregar para un input de tamaño i

la ejecución en la práctica. No nos interesará entonces el tiempo exacto de ejecución de un algoritmo dado, si no más bien el "orden de crecimiento" del tiempo necesario con respecto al tamaño del input. En lo que sigue introduciremos una notación que nos permitirá obtener estimaciones con estas características.

Trataremos con funciones con dominio natural $\mathbb{N}$ y recorrido real positivo $\mathbb{R}^+$. El dominio representará al tamaño del input de un algoritmo y el recorrido al tiempo necesario para ejecutar el algoritmo, por eso nos interesará $\mathbb{R}^+$ (no tiene sentido hablar de tiempo negativo).

Def: Sean $f : \mathbb{N} \to \mathbb{R}^+$ y $g : \mathbb{N} \to \mathbb{R}^+$ funciones con dominio natural y recorrido real positivo. Definimos $O(g(n))$ como el conjunto de todas las funciones f tales que existen $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$, que cumplen:

$$\forall n > n_0, \quad f(n) \leq cg(n).$$

Cuando $f(n) \in O(g(n))$ diremos que $f(n)$ es *a lo más* de orden $g(n)$, o simplemente que $f(n)$ *es* $O(g(n))$.

Definimos $\Omega(g(n))$ como el conjunto de todas las funciones f tales que existen $c \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$, que cumplen:

$$\forall n > n_0, \quad f(n) \geq cg(n).$$

Cuando $f(n) \in \Omega(g(n))$ diremos que $f(n)$ es *a lo menos* de orden $g(n)$, o simplemente que $f(n)$ *es* $\Omega(g(n))$.

Finalmente definimos $\Theta(g(n))$ como la intersección entre $O(g(n))$ y $\Omega(g(n))$, o sea, como el conjunto de todas las funciones f tales que $f(n) \in O(g(n))$ y $f(n) \in \Omega(g(n))$. Definiéndolo de manera similar a los casos anteriores diremos que $\Theta(g(n))$ es el conjunto de todas las funciones f tales que existen $c_1, c_2 \in \mathbb{R}^+$ y $n_0 \in \mathbb{N}$ que cumplen:

$$\forall n > n_0, \quad f(n) \leq c_1 g(n) \wedge f(n) \geq c_2 g(n).$$

Cuando $f(n) \in \Theta(g(n))$ diremos que $f(n)$ *es (exactamente)* de orden $g(n)$, o simplemente que $f(n)$ *es* $\Theta(g(n))$.

Estos conceptos se ven en la figura 3.1.

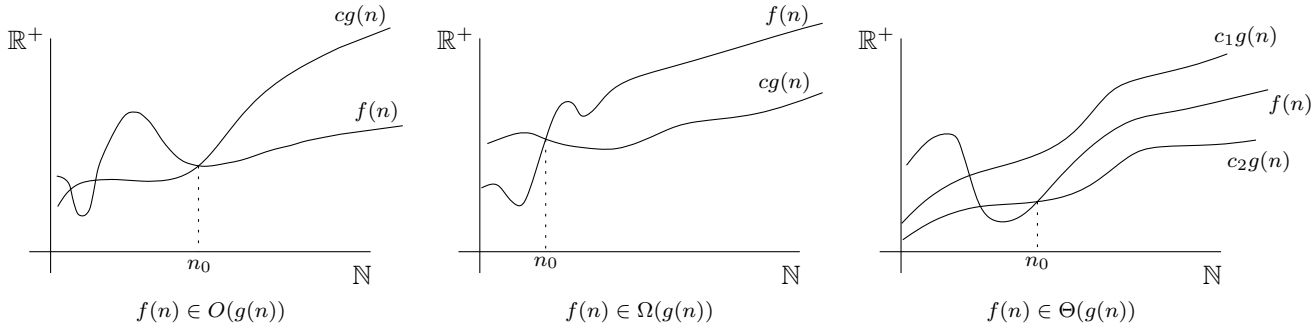

Figura 3.1: Órdenes de crecimiento de funciones.

Ejemplo: La función $f(n) = 60n^2$ es $\Theta(n^2)$. Para comprobarlo debemos demostrar que $f(n)$ es $O(n^2)$ y $\Omega(n^2)$ simultaneamente. En el primer caso debemos encontrar c y n_0 tal que $\forall n > n_0$ se cumpla que $f(n) \leq cn^2$, es claro que si elegimos $c = 60$ y $n_0 = 0$ tenemos que

$$\forall n > 0, \quad f(n) = 60n^2 \leq 60n^2$$

y por lo tanto se cumple que $f(n)$ es $O(n^2)$. En el segundo caso debemos encontrar c y n_0 tal que $\forall n > n_0$ se cumpla que $f(n) \geq cn^2$, eligiendo los mismos valores $c = 60$ y $n_0 = 0$ tenemos que

$$\forall n > 0, \quad f(n) = 60n^2 \geq 60n^2$$

y por lo tanto se cumple que $f(n)$ es $\Omega(n^2)$. Finalmente concluimos entonces que $f(n) = 60n^2$ es efectivamente $\Theta(n^2)$.

Ejemplo: La función $f(n) = 60n^2 + 5n + 1$ es $\Theta(n^2)$ es $\Theta(n^2)$. Para comprobarlo debemos demostrar que $f(n)$ es $O(n^2)$ y $\Omega(n^2)$ simultaneamente. En el primer caso debemos encontrar c y n_0 tal que $\forall n > n_0$ se cumpla que $f(n) \leq cn^2$. Para encontrar estos valores usaremos lo siguiente:

$$\forall n \geq 1, \quad f(n) = 60n^2 + 5n + 1 \leq 60n^2 + 5n^2 + n^2 = 66n^2,$$

esto porque $n^2 \geq n \geq 1$ para todo $n \geq 1$, entonces si tomamos $c = 66$ y $n_0 = 1$ obtenemos que $f(n)$ cumple con las condiciones para ser $O(n^2)$. Ahora si usamos que

$$\forall n \geq 0, \quad f(n) = 60n^2 + 5n + 1 \geq 60n^2 + 0 + 0 = 60n^2,$$

ya que $1 \geq 0$ y $5n > 0$ para todo $n \geq 0$, entonces si tomamos $c = 60$ y $n_0 = 0$ obtenemos que $f(n)$ cumple con las condiciones para ser $\Omega(n^2)$. Finalmente se concluye que $f(n)$ es $\Theta(n^2)$.

La moraleja que se puede sacar del primer ejemplo es que las constantes no influyen en la notación Θ. La moraleja en el segundo ejemplo es que para funciones polinomiales sólo el término con mayor exponente influye en la notación Θ. Estos conceptos se resumen en el siguiente teorema (cuya demostración se deja como ejercicio).

Teorema 3.1.4: Si $f(n) = a_k n^k + a_{k-1} n^{k-1} + \cdots + a_2 n^2 + a_1 n + a_0$, con $a_i \in \mathbb{R}$ y tal que $a_k > 0$, entonces se cumple que $f(n)$ es $\Theta(n^k)$.

Demostración: Ejercicio. □

Ejemplo: La función $f(n) = \log_2(n)$ es $\Theta(\log_3(n))$. Supongamos que $\log_2(n) = x$ y que $\log_3(n) = y$ de esto obtenemos que $2^x = 3^y$ por la definición de logaritmo. Si en esta última ecuación tomamos el $\log_2$ a ambos lados obtenemos:

$$x = \log_2(3^y) = y \log_2(3),$$

de donde reemplazando los valores de x e y obtenemos

$$\log_2(n) = \log_2(3) \log_3(n)$$

De donde obtenemos que

$$\forall n > 1, \quad \log_2(n) \leq \log_2(3) \log_3(n)$$
$$\forall n > 1, \quad \log_2(n) \geq \log_2(3) \log_3(n)$$

Si usamos entonces $c = \log_2(3)$ y $n_0 = 1$ resulta que $f(n) = \log_2(n)$ cumple las condiciones para ser $O(\log_3(n))$ y $\Omega(\log_3(n))$ simultáneamente y por lo tanto es $\Theta(\log_3(n))$.

El siguiente teorema generaliza el ejemplo anterior, su demostración se deja como ejercicio.

Teorema 3.1.5: Si $f(n) = \log_a(n)$ con $a > 1$, entonces para todo $b > 1$ se cumple que $f(n)$ es $\Theta(\log_b(n))$.

Demostración: Ejercicio. □

Este teorema nos permite independizarnos de la base del logaritmo cuando en el contexto de la notación asintótica. En adelante para una función logarítmica simplemente usaremos $\Theta(\log n)$ sin especificar la base. Si necesitáramos la base para algún cálculo generalmente supondremos que es 2.

En el siguiente ejemplo....

Ejemplo: La función $f(n) = \sqrt{n}$ es $O(n)$ pero no es $\Omega(n)$ (y por lo tanto no es $\Theta(n)$). La parte fácil es mostrar que $\sqrt{n}$ es $O(n)$, de hecho dado que

$$\forall n \geq 0, \quad \sqrt{n} \leq n$$

concluimos que tomando $c = 1$ y $n_0 = 0$ la función $f(n) = \sqrt{n}$ cumple las condiciones para ser $O(n)$. Queremos demostrar ahora que $f(n) = \sqrt{n}$ NO es $\Omega(n)$, lo haremos por contradicción. Supongamos que efectivamente $f(n) = \sqrt{n}$ es $\Omega(n)$, o sea, existe una constante $c > 0$ y un $n_0 \in \mathbb{N}$ tal que

$$\forall n > n_0, \quad \sqrt{n} \geq cn.$$

De lo anterior concluimos que

$$\forall n > n_0, \quad n \geq c^2 n^2 \Rightarrow 1 \geq c^2 n,$$

de donde concluimos que $c^2 < 1$ (ya que $n > n_0 \geq 0$).... [seguir]

Las distintas funciones que pertenecen a distintos órdenes de crecimiento (notación Θ, O y Ω), tienen nombres característicos, los más importantes se mencionan en la siguiente tabla, m y k son constantes positivas, $m \geq 0$ y $k \geq 2$.

Notación	Nombre
$\Theta(1)$	Constante
$\Theta(\log n)$	Logarítmico
$\Theta(n)$	Lineal
$\Theta(n \log n)$	$n \log n$
$\Theta(n^2)$	Cuadrático
$\Theta(n^3)$	Cúbico
$\Theta(n^m)$	Polinomial
$\Theta(k^n)$	Exponencial
$\Theta(n!)$	Factorial

3.1.3. Complejidad de Algoritmos Iterativos

Nos preguntaremos por el tiempo que tarda cierto algoritmo en ejecutar, este tiempo dependerá del tamaño del input y en general lo modelaremos como una función $T(n)$ con n el tamaño del input. No nos interesa el valor exacto de T para cada n si no más bien una *notasión asintótica* para ella. Para *estimar* el tiempo lo que haremos es contar las instrucciones ejecutadas por el algoritmo, en algunos casos puntuales estaremos interesados en contar una instrucción en particular y obtener para ese número una notación asintótica.

Ejemplo: Consideremos el siguiente trozo de código:

```
1  for i := 1 to n
2      for j := 1 to i
3          x := x + 1
```

Nos interesa encontrar una notación asintótica para el número de veces que se ejecuta la línea 3 en función de n, digamos que esta cantidad es $T(n)$. Si fijamos el valor de i, las veces que se ejecuta línea 3 está ligada sólo al ciclo que comienza en la línea 2, y es tal que, si $i = 1$ se ejecuta una vez, si $i = 2$ se ejecuta dos veces,... si $i = k$ se ejecuta k veces. Entonces, dado que el valore de i va incrementándose desde 1 hasta n, obtenemos la expresión

$$T(n) = 1 + 2 + 3 \cdots + (n-1) + n = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

de donde concluimos que la cantidad de veces que se ejecuta la línea 3 es $\Theta(n^2)$.

Ejemplo: Consideremos el siguiente trozo de código:

```
1  j := n
2  while j ≥ 1 do
3      for i := 1 to j
4          x := x + 1
5      j := ⌊j/2⌋
```

Nos interesa encontrar una notación asintótica para el número de veces que se ejecuta la línea 4 en función de n, digamos que esta cantidad es $T(n)$. En este caso no resulta tan simple el cálculo. Primero notemos que para un j fijo, las veces que se ejecuta la línea 4 está solamente ligada al ciclo **for** que comienza en la línea 3. Ahora cada vez se termina una iteración del ciclo **while** de la línea 2 el valor de j se disminuye en la mitad. Supongamos ahora que el ciclo **while** se ejecuta k veces, obtenemos entonces la siguiente expresión para la cantidad de veces que se ejecuta la línea 4:

$$T(n) = n + \frac{n}{2} + \frac{n}{4} + \cdots + \frac{n}{2^{k-1}} = \sum_{i=0}^{k-1} \frac{n}{2^i}.$$

Necesitamos una expresión para esta última sumatoria. Podemos reescribirla y usar la fórmula para la serie geométrica para obtener

$$\sum_{i=0}^{k-1} \frac{n}{2^i} = n \sum_{i=0}^{k-1} \left(\frac{1}{2}\right)^i = n\, \frac{1 - \left(\frac{1}{2}\right)^k}{1 - \frac{1}{2}} = 2n \left(1 - \left(\frac{1}{2}\right)^k\right).$$

Ahora, dado que $1 - (\frac{1}{2})^k \leq 1$ obtenemos que

$$\forall n > 1, \quad T(n) = \sum_{i=0}^{k-1} \frac{n}{2^i} \leq 2n,$$

y por lo tanto la cantidad de veces que se ejecuta la línea 4 es $O(n)$. Dado que inicialmente para el valor $j = n$ la línea 4 se ejecuta n veces, concluimos que $T(n) \geq n$ y por lo tanto es $\Omega(n)$. Sumado este último resultado al resultado anterior obtenemos que la cantidad de veces que se ejecuta la línea 4 es $\Theta(n)$.

En los dos ejemplos anteriores hemos visto analizado el número de veces que se ejecuta cierta instrucción con respecto a un parámetro. Cuando nos enfrentamos a un algoritmo y queremos estimar el tiempo que demora entregando una notación asintótica (como función del tamaño del input) para él, siempre debemos encontrar una o más instrucciones que sean representativas del tiempo que tardará el algoritmo y contar cuántas veces se repite. Por ejemplo, consideremos el siguiente algoritmo que busca un elemento dentro de una lista

INPUT: Una secuencia de enteros $S = s_1, s_2, \ldots, s_n$, un natural $n > 0$ correspondiente al largo de la secuencia y un entero k.
OUTPUT: El índice en el que k aparece en la secuencia S o 0 si k no aparece en S.

BÚSQUEDA(S, n, k)

```
1  for i := 1 to n
2      if s_i = k then
3          return i
4  return 0
```

En este algoritmo por ejemplo, las líneas 3 y 4 no son representativas del tiempo que tardará este algoritmo ya que cada una de ellas o no se ejecuta o se ejecuta sólo una vez. Resulta entonces que una instrucción representativa es la que se realiza en la línea 2 más específicamente la comparación que se realiza en 2. Otro punto importante es cómo medimos el tamaño del input. En este y otros casos en donde el input sea una lista de elementos, será natural tomar como tamaño del input la cantidad de elementos en la lista. Queremos entonces encontrar una notación Θ para las veces que se ejecuta la comparación de la línea 2 en función del parámetro n, llamaremos a esta cantidad $T(n)$. Un problema que surge es que esta cantidad no sólo dependerá de n si no también de cuáles sean los datos de entrada. Para sortear este problema dividiremos entonces el estudio de un algoritmo en particular en estimar el tiempo que tarda en el *peor caso* y en el *mejor caso* por separado para un tamaño de input específico. Por peor caso se refiere a la situación en que los datos de entrada hacen que el algoritmo tarde lo más posible, y por mejor caso a la situación en que los datos hacen que se demore lo menos posible. En nuestro ejemplo, el mejor caso ocurre cuando $s_1 = k$ en cuyo caso la línea 2 se ejecuta una única vez y por lo tanto $T(n)$ es $\Theta(1)$. En cuanto al peor caso, este ocurre cuando k no aparece en S y por lo tanto la línea 2 se ejecuta tantas veces como elementos tenga S, o sea $T(n)$ es $\Theta(n)$. Diremos finalmente que el algoritmo BÚSQUEDA(S, n, k) es de *complejidad* $\Theta(n)$ en el peor caso y $\Theta(1)$ en el mejor caso.

Tomemos el siguiente ejemplo para analizar también el mejor y peor caso.

Ejemplo: Consideremos el siguiente algoritmo para ordenar una secuencia de números enteros.

INPUT: Una secuencia de enteros $S = s_1, s_2, \ldots, s_n$, un natural $n > 0$ correspondiente al largo de la secuencia.
OUTPUT: Al final del algoritmo la secuencia S se encuentra ordenada, es decir $s_1 \leq s_2 \leq \cdots \leq s_n$.

INSERT-SORT(S, n)

```
1  for i := 2 to n
2      t := s_i
3      j := i − 1
4      while s_j > t ∧ j ≥ 1 do
5          s_{j+1} := s_j
6          j := j − 1
7      s_j := t
```

Este algoritmo para cada iteración i del ciclo principal, busca la posición que le corresponde a s_i en la secuencia ya ordenada $s_1, s_2, \ldots, s_i$ y lo "inserta" directamente en la posición que corresponde "moviendo" los valores que sean necesarios. La búsqueda del lugar mas el movimiento de los valores se hacen en el ciclo que comienza en la línea 4. El alumno como ejercicio puede demostrar la corrección de este algoritmo, o sea que efectivamente ordena la secuencia S.

En este caso las instrucciones representativas para calcular el tiempo $T(n)$ que demora el algoritmo en ordenar una secuencia de largo n, son las que se encuentran en el ciclo que comienza en 4. El mejor caso del algoritmo ocurre entonces cuando el ciclo en 4 ni siquiera comienza, o sea en el caso que cada vez que se llegue a él, se cumpla que $s_j \leq t$, o sea que $s_{i-1} \leq s_i$ para $2 \leq i \leq n$, es decir, el mejor caso ocurre cuando la secuencia inicialmente está ordenada, en este caso la comparación de la línea 4 se repite $n-1$ veces y por lo tanto $T(n)$ es $\Theta(n)$ en el mejor caso. El peor caso ocurre cuando el ciclo en 4 se detiene porque $j < 1$ en este caso el ciclo de la línea $i-1$ veces para cada valor de i entre 2 y n, por lo tanto una estimación para $T(n)$ estaría dada por la expresión

$$1 + 2 + \cdots + (n-1) = \frac{n(n-1)}{2}$$

que sabemos que es $\Theta(n^2)$. Finalmente diremos que el algoritmo INSERT-SORT(S, n) tiene complejidad $\Theta(n^2)$ en el peor caso y $\Theta(n)$ en el mejor caso.

En computación estamos interesados principalmente en el peor caso de ejecución de un algoritmo, ya que esto nos dará una estimación de qué tan mal puede comportarse el algoritmo en la práctica. A veces puede resultar difícil encontrar una notación Θ (O y Ω a la vez) para un algoritmo, por lo que nos conformaremos con una buena aproximación O (tanto para el mejor como peor caso) que también nos dará una cota superior al tiempo total usado por el algoritmo.

Un punto importante en el que no hemos puesto suficiente énfasis es en cómo medimos el tamaño del input para un algoritmo. Mencionamos que cuando consideremos el input como una lista de elementos usaremos como tamaño del input la cantidad de elementos de la lista, esto principalmente porque las operaciones que se harán serán comparaciones o intercambios de valores entre los elementos de la secuencia. En el siguiente ejemplo el algoritmo recibe un único elemento y decide si es o no un número primo.

INPUT: Un número natural $n > 1$.
OUTPUT: **SI** si n es un número primo, **NO** en otro caso.

EsPrimo(n)

```
1  for i := 2 to n − 1
2      if  n mod i = 0 then
3              return NO
4  return SI
```

En este caso podemos estimar el tiempo en función de las veces que se ejecute la línea 2. Es claro que el mejor caso es que el valor n sea par, en cuyo caso será inmediatamente divisible por 2 y el algoritmo toma tiempo $\Theta(1)$. El peor caso para el algoritmo es cuando n sea efectivamente un número primo. Si tomáramos el tamaño del input como el valor numérico de n entonces podríamos decir que en el peor caso este algoritmo tarda $\Theta(n)$ que es exactamente la cantidad de veces que se ejecuta el ciclo de la línea 1. Sin embargo, cuando el input es un número n, el valor numérico no es una buena estimación del tamaño del input. Una mucho mejor estimación del tamaño del input, y la que usaremos en el caso de algoritmos numéricos, es la cantidad de dígitos que son necesarios utilizar para representar a n. Si d es la cantidad de dígitos necesaria para representar a n, d es aproximadamente $\log_{10}(n)$, y por lo tanto n es aproximadamente 10^d. Esto nos lleva a concluir que si el tamaño del input es d (la cantidad de dígitos de n) el algoritmo en el peor caso tarda $\Theta(10^d)$, o sea exponencial. Resulta entonces que el comportamiento del algoritmo es lineal con respecto al valor numérico del input pero exponencial con respecto a la cantidad de dígitos del input. No lo hemos mencionado aun, pero un tiempo exponencial es muy malo en la práctica, tocaremos este tema en la siguiente sección.

El que el tiempo se mida en cuanto al valor numérico o a la cantidad de dígitos hace diferencias muy importantes en la práctica. Por ejemplo, una mejora que se puede hacer al algoritmo EsPrimo es cambiar el ciclo de la línea 1 para que sólo revise los valores hasta un umbral de $\sqrt{n}$ (¿por qué funciona?), ¿cómo afecta esto a la complejidad asintótica del algoritmo?. Si medimos el tamaño del input como el valor de n vemos que el tiempo se disminuye a $\Theta(\sqrt{n})$ que es bastante mejor que $\Theta(n)$. Sin embargo si tomamos a d, la cantidad de dígitos de n, como el tamaño del input, por lo que $\sqrt{n} = n^{\frac{1}{2}}$ es aproximadamente $10^{\frac{d}{2}}$. Ahora es posible demostrar que $10^{\frac{d}{2}} > 3^d$ por lo que el tiempo de ejecución del algoritmo sería $\Omega(3^d)$ o sea sigue siendo exponencial y, como veremos en la siguiente sección, en la práctica no es mucho lo que ganamos con disminuir el espacio de búsqueda hasta $\sqrt{n}$.

En el siguiente ejemplo también se trata el tema de cómo medimos el tamaño del input para un algoritmo.

Ejemplo: El siguiente algoritmo calcula la representación en base 3 de un número en base decimal. Usa una secuencia de valores s_i donde almacena los dígitos en base 3

INPUT: Un número natural $n > 1$.
OUTPUT: muestra una lista con los dígitos de la representación en base 3 de n.

BASE3(n)

1 $s_1 := 0$
2 $j := 1$
3 $m := n$
4 **while** $m > 0$ **do**
5 $\quad s_j := n \bmod 3$
6 $\quad m := \lfloor \frac{m}{3} \rfloor$
7 $\quad j := j + 1$
8 **print** $s_{j-1}, s_{j-2}, \ldots, s_1$

No discutiremos en detalle el por qué este algoritmo es correcto, dejaremos esta discusión como ejercicio y nos centraremos en la complejidad del algoritmo. El valor de m se va dividiendo por 3 en cada iteración del ciclo en la línea 4, por lo que en la iteración j, el valor de m es aproximadamente $n/3^j$. El ciclo termina cuando el valor de m es 0, o sea cuando j es tal que $n/3^j < 1$, o sea $j > \log_3 n$, por lo que el ciclo se ejecuta al menos $\log_3 n$ veces. Si tomamos el tamaño del input como el valor numérico de n obtenemos que el algoritmo tarda $O(\log n)$, o sea tiempo logarítmico. Si por otra parte tomamos como tamaño del input la cantidad d de dígitos decimales de n, el algoritmo tarda $O(d)$, o sea tiempo lineal.

3.1.4. Relaciones de Recurrencia y Complejidad de Algoritmos Recursivos

[...falta completar...]

3.2. Problemas Computacionales

En la sección anterior tratamos con algoritmos para resolver problemas computacionales y estudiamos la corrección y complejidad de esos algoritmos. Sin embargo se puede estudiar un problema computacional sin ligarse directamente con un algoritmo en particular que lo resuelva, si no más bien ligandose con *todos los posibles algoritmos que lo resuelven.*

Por ejemplo en la sección anterior vinos el algoritmo EsPrimo que para un natural $n > 2$ verificaba si este era o no divisible por cada uno de los números menores que él y respondía acerca de la *primalidad* (si es o no primo) de n. Vimos también una modificación a este algoritmo que verificaba sólo hasta un umbral de $\sqrt{n}$ para decidir acerca de la primalidad de n. En ambos casos concluíamos que el algoritmo demoraba tiempo exponencial con respecto a la cantidad de dígitos de n (que era la mejora medida de tamaño del input en este caso). La pregunta que nos hacemos es ¿qué tan malo puede ser un algoritmo exponencial en la práctica? En la siguiente tabla se muestra una estimación del tiempo que le tardaría ejecutar un algoritmo de distintas complejidades en un computador de propósito general. Estamos suponiendo que el computador puede ejecutar 10^9 instrucciones por segundo (algo como 1GHz de velocidad).

	$n = 3$	$n = 20$	$n = 100$	$n = 1000$	$n = 10^5$
$\log n$	$2 \cdot 10^{-9}$ seg	$5 \cdot 10^{-9}$ seg	$7 \cdot 10^{-9}$ seg	10^{-8} seg	$2 \cdot 10^{-8}$ seg
n	$3 \cdot 10^{-9}$ seg	$2 \cdot 10^{-8}$ seg	10^{-7} seg	10^{-6} seg	10^{-4} seg
n^3	$3 \cdot 10^{-8}$ seg	$8 \cdot 10^{-6}$ seg	0,001 seg	1 seg	10 días
2^n	$8 \cdot 10^{-9}$ seg	0,001 seg	$4 \cdot 10^{13}$ años	$3 \cdot 10^{284}$ años	$3 \cdot 10^{30086}$años
10^n	10^{-9} seg	100 años	$3 \cdot 10^{89}$ años	$3 \cdot 10^{989}$ años	...

Vemos que los tiempos exponenciales, incluso para instancias pequeñas (entre 20 y 100) se hacen inmensamente grandes, pero que sin embargo los tiempo polinomiales incluso para instancias grandes se mantienen dentro de lo que se podría estar dispuesto a esperar. Alguien podría pensar que esperar 10 días para un algoritmo cúbico (n^3) es demasiado, pero este tiempo no es tan grande si pensamos que al contar con un computador que fuese 10 veces más rápido el tiempo se disminuiría a sólo un día. Sin embargo para un algoritmo exponencial, no importa que contemos con un computador 10, 100 o incluso 10000 veces más rápido, el tiempo seguirá siendo demasiado alto para instancias pequeñas. Esto nos da una gran diferencia entre algoritmo polinomiales y exponencial

Volviendo al problema de determinar si un número es o no primo, conocemos un algoritmo exponencial que lo resuelve ¿puede hacerse más rápido que eso, o exponencial es lo mejor que podemos lograr? Esta y otras preguntas motivan las siguientes secciones.

En adelante estaremos más interesados en *problemas de decisión*, o sea, en que las respuestas al problema son SI o NO. Formalizaremos esta noción en la siguiente sección.

3.2.1. Problemas de Decisión y la Clase *P*

Partiremos definiendo lo que es un *problema de decisión.* Informalmente un problema de decisión es uno en el cuál las respuestas posibles son SI o NO. Por ejemplo, si definimos el problema PRIMO que se trata de dado un natural n responder SI cuando n sea primo y NO cuando no lo sea, o el problema EULERIANO que se trata de dado un grafo conexo G responde SI cuando el grafo contenga un ciclo Euleriano y NO cuando no lo contenga, ambos son problemas de decisión.

Def: Formalmente un problema de decisión se puede definir en base a pertenencia de elementos a conjuntos. Un problema de decisión π está formado por un conjunto I_π de instancias, y un subconjunto L_π de él, $L_\pi \subseteq I_\pi$. El problema π se define entonces cómo:

$$\text{dado un elemento } w \in I_\pi \text{ determinar si } w \in L_\pi.$$

Al conjunto I_π se le llama conjunto de inputs o de posibles instancias de π, al conjunto L_π se le llama conjunto de instancias positivas de π.

Diremos que un algoritmo A resuelve un problema de decisión π si para cada input $w \in I_\pi$, el algoritmo A responde SI cuando $w \in L_\pi$ y responde NO cuando $w \in I_\pi - L_\pi$ (para este último caso a veces diremos simplemente $w \notin L_\pi$). En la figura 3.2 se muestra un diagrama de un problema de decisión y de un algoritmo que lo resuelve.

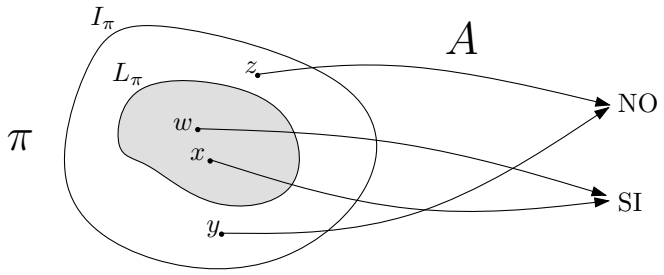

Figura 3.2: El algoritmo A resuelve el problema de decisión π, w y x son instancias positivas del problema (pertenecen a L_π), y y z no lo son (perteneces a $I_\pi - L_\pi$).

Ejemplo: En el caso del problema de decisión PRIMO los conjuntos asociados son $I_{\text{PRIMO}} = \mathbb{N} - \{0, 1\}$ y $L_{\text{PRIMO}} = \{p \in \mathbb{N} \mid p \text{ es primo}\}$. El algoritmo EsPrimo resuelve el problema PRIMO ya que para cada $n \in L_{\text{PRIMO}}$ responde SI y para cada $n \in I_{\text{PRIMO}} - L_{\text{PRIMO}}$ responde NO.

En el caso del problema de decisión EULERIANO, y suponiendo que $\mathcal{G}$ es el conjunto de todos los grafos, los conjuntos asociados son $I_{\text{EULERIANO}} = \{G \in \mathcal{G} \mid G \text{ es conexo}\}$ y $L_{\text{EULERIANO}} = \{G \in \mathcal{G} \mid G \text{ tiene un ciclo Euleriano}\}$. El siguiente es un algoritmo para resolver el problema EULERIANO.

INPUT: Un grafo G conexo (asociado a G están los conjuntos $V(G)$ y $E(G)$ de vértices y aristas de G).
OUTPUT: **SI** si G tiene un ciclo Euleriano, **NO** si no lo tiene.

```
EsEuleriano(G)
1  for each v ∈ V(G)
2      δ := 0
3      for each u ∈ V(G)
4          if vu ∈ E(G) then
5              δ := δ + 1
6      if δ mod 2 ≠ 0 then
7          return NO
8  return SI
```

El ciclo que comienza en la línea 3 cuenta la cantidad de vecinos que tiene un vértice particular v almacenando el resultado en δ, esto lo hace para cada vértice del grafo (ciclo de la línea 1). Si encuentra algún vértice con una cantidad impar de vecinos entonces responde NO (línea 6). Si todos los vértices tienen una cantidad par de vecinos entonces responde SI. El algoritmo es correcto por el teorema 2.2.5. La complejidad de este algoritmo es $O(n^2)$ si suponemos que $n = |V(G)|$, osea n es la cantidad de vértices del grafo G.

Ejemplo: Existen muchos problemas de decisión, algunos de interés teórico, otros de interés más práctico, la siguiente lista presenta otros problemas de decisión.

- CAM-EULER: Dado un grafo conexo G determinar si existe un camino Euleriano en él, o sea, un camino no cerrado que contiene a todas las aristas (y todos los vértices) de G.

- HAMILTONIANO: Dado un grafo conexo G determinar si existe un ciclo Hamiltoniano en él, o sea, un ciclo que contiene a todos los vértices de G y que no repite vértices. En este caso tenemos:
- CAM-HAMILTON: Dado un grafo conexo G determinar si existe un camino Hamiltoniano en él, o sea, un camino no cerrado que contiene a todos los vértices de G y que no repite vértices.
- k-CLIQUE: Dado un grafo G y un natural k determinar si G tiene un clique de tamaño k (clique con k vértices).
- BIPARTITO: Dado un grafo G determinar si G es un grafo bipartito.
- COMPILA-C: Dado un string s de caracteres ASCII determinar si s es un programa correctamente escrito según la sintaxis de C.
- OCURRENCIA: Dado un par de strings s_1 y s_2 de caracteres ASCII determinar si s_2 ocurre como sub–string de s_1.
- ORDEN: Dada una secuencia de enteros $S = (s_1, s_2, \ldots, s_n)$ decidir si la secuencia se encuentra ordenada.
- MIN: Dado un número entero m y una secuencia de enteros $S = (s_1, s_2, \ldots, s_n)$, determinar si m es el mínimo de la secuencia S.
- MCD : Dado un trío de naturales l, m y n determinar si n es el máximo común divisor de m y l.
- COMPUESTO: Dado un número natural n determinar si n es un número compuesto (o sea si n no es un número primo).

Como ejercicio, para cada uno de estos problemas se puede encontrar sus conjuntos I_π y L_π asociados.

En el ejemplo anterior, el problema de decisión MCD proviene del problema de búsqueda "dados dos naturales l y m, encontrar el máximo común divisor entre ellos", algo similar ocurre con los problemas ORDEN y MIN. La mayoría de los problemas de búsqueda (o incluso de optimización) se pueden transformar de manera similar en problemas de decisión.

Si notamos en la definición del problema COMPUESTO, este es exactamente el problema *inverso* de PRIMO, esto motiva nuestra siguiente definición.

Def: A cada problema de decisión π con sus conjuntos I_π y L_π, se le puede asociar el problema de decisión π^c que se define de la siguiente manera:

$$\text{dado un elemento } w \in I_\pi \text{ determinar si } w \in I_\pi - L_\pi.$$

O sea, para el problema π^c las instancias posibles son exactamente las mismas que para π, pero las instancias positivas de π^c son exactamente las instancias contrarias a las de π, esto quiere decir que $I_{\pi^c} = I_\pi$ y $L_{\pi^c} = I_\pi - L_\pi$.

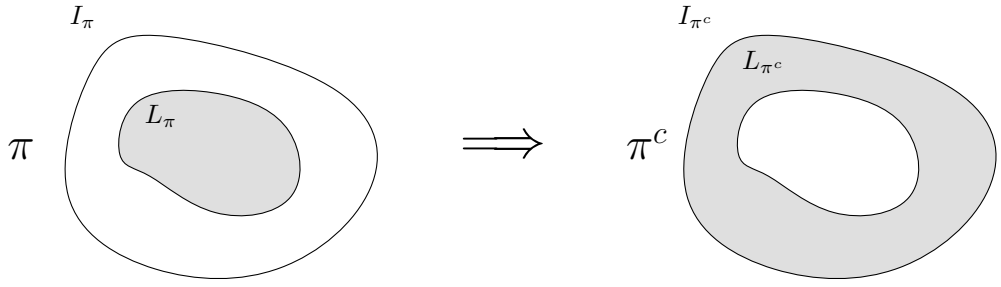

Figura 3.3: El problema π y su problema complemento π^c.

Es claro que si un algoritmo A resuelve el problema π, entonces se puede crear una algoritmo A' para resolver π^c: A' es exactamente igual que, A excepto que cuando A responde SI, A' responde NO, y cuando A responde NO, A' responde SI.

Ejemplo: Consideremos el problema de decisión CICLO-IMP que dado un grafo G decide si G contiene un ciclo con un número impar de vértices y aristas. El teorema 2.2.4 nos dice que se cumple la igualdad CICLO-IMP^C = BIPARTITO. Por la definición de los problemas PRIMO y COMPUESTO sabemos que también se cumple la igualdad PRIMO^C = COMPUESTO.

Estamos interesados en clasificar los problemas según su dificultad *en la práctica*, en cuanto a qué tan rápido puede ser un algoritmo que lo resuelva. Ya dimos evidencia de que un algoritmo exponencial es impracticable, pero que un algoritmo polinomial parecía ser mucho más realizable. Para formalizar estas nociones intuitivas usaremos las siguientes definiciones.

Def: Definimos el conjunto P de todos los problemas de decisión π para los cuales existe un algoritmo de complejidad a lo más polinomial en el peor caso que resuelve π

$$P = \{\pi \text{ problema de decisión} \mid \text{existe un algoritmo de peor caso polinomial para resolver } \pi\}$$

O sea, $\pi \in P$ si existe un natural k y un algoritmo A que resuelve π, tal que A es de complejidad $O(n^k)$ en el peor caso si n es el tamaño de la instancia para A. Diremos (informalmente) que un problema π es *tratable* o *soluble eficientemente en la práctica* si $\pi \in P$.

La primera observación que se puede hacer es que si $\pi \in P$ entonces claramente $\pi^c \in P$ (¿por qué?).

El algoritmo EsEuleriano resuelve el problema EULERIANO en tiempo $O(n^2)$ en el peor caso, por lo que EULERIANO $\in P$. No es difícil argumentar (ejercicio) que los siguientes problemas también están en P: CAM-EULER, OCURRENCIA, ORDEN, MIN, MCD. Tal vez no podemos dar una argumentación formal acerca de que el problema COMPILA-C pertenece a P, sin embargo la noción intuitiva de P nos ayuda en este caso, de hecho, todos los días hay gente compilando programas en C y en ningún caso ocurre que un programa demore años en compilar, dado que es un problema que se resuelve continuamente en la práctica este debe estar en P, de hecho efectivamente ocurre que COMPILA-C $\in P$.

Un caso a parte lo tiene el problema PRIMO. Hemos mostrado un algoritmo (EsPrimo) que resuelve el problema pero que tarda tiempo exponencial $O(3^d)$ (el algoritmo mejorado) con d la cantidad de dígitos del número de input. Este hecho no nos permite concluir nada acerca de si PRIMO pertenece o no a P, de hecho no nos entrega información alguna. Hasta el año 2002 no se sabía si el problema PRIMO estaba o no en P y la pregunta acerca de si existía un algoritmo eficiente para determinar si un número era primo había estado abierta durante siglos. En 2002, Agrawal, Kayal y Saxena (todos científicos Indios) encontraron un algoritmo de peor caso polinomial para resolver el problema PRIMO y por lo tanto ahora sabemos que PRIMO $\in P$. Una conclusión directa es que COMPUESTO $\in P$ también.

¿Existen problemas que se sepa que no están en P? La respuesta es SI, existen de hecho problemas para los cuales ni siquiera existe un algoritmo (de ninguna complejidad!) que los resuelva, pero esta discusión está fuera del alcance de nuestro estudio.

Nos interesan otro tipo de problemas que no se saben si están o no en P pero cuya respuesta traería implicaciones importantísimas en el área de computación.

Ejemplo: Coloración de Mapas. Una coloración de un mapa en un plano, es una asignación de colores para cada país en el mapa, tal que a dos países vecinos se les asigna colores diferentes. Países vecinos son los que comparten alguna frontera. Para simplificar supondremos que los colores son números naturales. En la figura 3.4 se muestra un mapa y una posible coloración para él. La coloración tiene 5 colores, una pregunta válida es ¿podremos colorearlo con menos de 5 colores? La respuesta es SI, de hecho sólo 4 son necesarios.

En 1977 Appel y Haken demostraron que cualquier mapa puede ser coloreado con 4 colores, sólo 4 colores son suficientes. Este hecho había sido una conjetura durante más de 100 años, se propuso inicialmente en

Figura 3.4:

el año 1853. La demostración de Appel y Haken se basó en el uso intensivo de poder computacional (ver `http://mathworld.wolfram.com/Four-ColorTheorem.html` para más información).

Nos podemos plantear el problema de decisión 4-MAP-COLOR que dado un mapa en el plano decide si este puede o no pintarse con 4 colores. El resultado de Appel y Haken nos permite hacer un algoritmo trivial para resolver este problema, uno que responde siempre SI para toda instancia.

Dado que la solución al problema anterior resulta trivial, nos podemos plantear los siguientes problemas no triviales:

- 2-MAP-COLOR: Dado un mapa en el plano decidir si este puede pintarse con sólo 2 colores.
- 3-MAP-COLOR: Dado un mapa en el plano decidir si este puede pintarse con sólo 3 colores.

Lo primero es notar que cualquier instancia positiva de 2-MAP-COLOR es también una instancia positiva de 3-MAP-COLOR, pero no a la inversa. Lo otro que se puede decir es que para resolver el problema 2-MAP-COLOR existe un algoritmo que toma tiempo polinomial en la cantidad de países del mapa y que por lo tanto 2-MAP-COLOR $\in P$....

¿Qué pasa con el problema 3-MAP-COLOR? En lo que sigue diseñaremos un algoritmo que lo resuelve. Dado un mapa con n países, numeraremos sus países de 1 a n, y representaremos el mapa por una matriz de adyacencia M tal que $M_{ij} = 1$ si los países i y j son vecinos, 0 en otro caso. Una asignación de colores para M la modelaremos como una secuencia $C = (c_1, c_2, \ldots, c_n)$ donde c_i es el color asignado al país i y cada $c_i \in \{0, 1, 2\}$. Note que a cada secuencia C distinta le corresponde un único número en base 3 entre 0 y $3^n - 1$, entonces podemos usar un procedimiento similar al algoritmo Base3 de la sección anterior para generar sistemáticamente las coloraciones posibles. El siguiente algoritmo 3-Coloracion resuelve el problema 3-MAP-COLOR, usa un procedimiento auxiliar Verifica-Coloracion.

INPUT: Una matriz M representante de un mapa en el plano y la cantidad de países n.
OUTPUT: **SI** si el mapa representado por M puede ser coloreado con 3 colores, **NO** si se necesitan más de 3 colores para colorear M.

3-Coloracion(M, n)

```
1  for i := 0 to 3^n − 1
2      C = BASE3(i)
3      if VERIFICA-COLORACION(M, n, C) then
4          return SI
5  return NO
```

```
VERIFICA-COLORACION(M, n, C)
1  for i := 1 to n − 1
2      for j := i + 1 to n
3          if M_ij = 1 ∧ c_i = c_j then
4              return false
5  return true
```

El algoritmo 3-COLORACION hace en el peor caso 3^n llamadas al procedimiento VERIFICA-COLORACION y dado que este último procedimiento toma tiempo $\Theta(n^2)$ con n la cantidad de países, 3-COLORACION toma tiempo $\Theta(3^{n^2})$ en el peor caso. Este algoritmo claramente no es polinomial, esto no indica que 3-MAP-COLOR no pertenezca a P, sólo nos dice que probando todas las posibilidades no obtenemos resultados realizables en la práctica. ¿Hay alguna manera polinomial de resolver el problema? Hasta el día de hoy no se ha encontrado un algoritmo polinomial para resolverlo, a grandes razgos lo mejor que se puede hacer es probar todas las posibilidades...

Hay muchos problemas de decisión de importancia práctica y teórica que están en la misma situación que 3-MAP-COLOR, estudiaremos estos problemas en la siguiente sección.

3.2.2. La Clase *NP* y Problemas *NP*-completos

En esta sección analizaremos una generalización del concepto de que un problema sea *tratable* o *soluble eficientemente en la práctica*. Antes dijimos que un problema π es *tratable* si ocurre que $\pi \in P$, o sea, si existe un algoritmo para resolver π que toma tiempo polinomial en el peor caso. Esta definición deja en *el limbo* a problemas como 3-MAP-COLOR para los cuales aun no se sabe si existe o no un algoritmo polinomial que lo resuelva. Para abarcar a este tipo de problemas se define la clase *NP* de problemas computacionales.

Existe una noción intuitiva simple de entender y muy útil para determinar cuándo un problema está en la clase de problemas *NP*. Suponga que hay dos personas A y V analizando el problema de decisión π, y supongamos que la persona A de alguna manera determina que una instancia posible $w \in I_\pi$ es una instancia positiva para π, o sea $w \in L_\pi$, entonces A siempre podrá encontrar "una forma de convencer rápidamente" a V de que w efectivamente es una instancia positiva para π. La frase "una forma de convencer rápidamente" es bastante imprecisa, con "una forma" nos referimos a algún tipo de información asociada a w que sirva para que convencer a V, y con "convencer rápidamente" nos referimos a que V puede verificar en tiempo a lo más polinomial (ayudado con la información proporcionada por A) que w es una instancia positiva. Por ejemplo, en el problema 3-MAP-COLOR si dado un mapa M, A determina que M puede ser coloreado con tres colores, basta con que A le muestre a V una asignación válida de colores para que V en tiempo polinomial se convenza de que efectivamente M era una instancia positiva, lo único que debería hacer V es verificar que la asignación mostrada por A no produce conflictos en los países de M. A la persona A se le llama *adivinador* y a la persona V se le llama *verificador*. Entonces, Un problema $\pi \in NP$ si para cada $w \in L_\pi$, el *adivinador* puede convencer en tiempo polinomial al *verificador* de que efectivamente $w \in L_\pi$.

La noción de "adivinar" y "verificar" se ve claramente en el algoritmo desarrollado para 3-MAP-COLOR de la sección anterior. En el procedimiento 3-COLORACION, en el ciclo de la línea 1, se está tratando de encontrar una secuencia de colores que formen una coloración válida para el mapa M de input. Si M efectivamente se

puede colorear con tres colores, estamos seguros de que el ciclo encontrará una secuencia de colores válida. Por su parte el procedimiento VERIFICA-COLORACION es el encargado de verificar que una asignación de colores encontrada es efectivamente una coloración válida. El ciclo principal en 3-COLORACION está "adivinando", el procedimiento VERIFICA-COLORACION por su parte está "verificando", verificación que se lleva a cabo en tiempo polinomial con respecto al tamaño del input (tiempo $\Theta(n^2)$ si n es la cantidad de países).

Ejemplo: Ya vimos que 3-MAP-COLOR $\in$ *NP* ya que para cada mapa que puede colorearse con tres colores, basta con que el *adivinador* le muestre la asignación de colores al *verificador* para que este se convenza en tiempo polinomial de que el mapa efectivamente se podía colorear con tres colores.

También ocurre que HAMILTONIANO $\in$ *NP*, ya que si un grafo G es efectivamente Hamiltoniano, basta con que el *adivinador* le muestre al *verificador* un ciclo en G que pasa por todos los vértices para que este último se convenza en tiempo polinomial de que G efectivamente era Hamiltoniano.

No es difícil darse cuenta que todos los problemas en la lista de problemas de decisión de la sección anterior pertenecen también a *NP*. Mención especial se podría hacer por ejemplo con el problema MIN, ¿de qué forma convence el *adivinador* al *verificador* de que dado un valor m y una secuencia de valores $S = (s_1, s_2, \ldots, s_n)$, m es el mínimo de la secuencia S? En este caso resulta que el *adivinador* no tiene que hacer nada, el *verificador* puede "convencerse sólo" ya que dado que MIN $\in P$, el *verificador* en tiempo polinomial puede comprobar que efectivamente m es el mínimo de S.

Este último ejemplo nos permite deducir lo siguiente: si un problema $\pi \in P$ entonces necesariamente $\pi \in NP$ ya que, dado que para π existe un algoritmo polinomial, el *verificador* puede "convencerse sólo" en tiempo polinomial de que una instancia w efectivamente está en L_π, no necesita ayuda del *adivinador*.

Formalizaremos la noción hasta ahora intuitiva de *NP* en la siguiente definición.

Def: Un problema de decisión π pertenece a la clase de problemas *NP* si a cada instancia positiva w de π, $w \in L_\pi$ se puede asociar un *certificado* de tamaño polinomial con respecto a w que llamaremos $c(w)$, tal que existe un algoritmo que usando $c(w)$ puede verificar en tiempo polinomial que w efectivamente pertenece a L_π.

Ejemplo: En el caso del problema 3-MAP-COLOR, dado un mapa M que se puede colorear con tres colores, el certificado $c(M)$ es la asignación de colores válida para M. El algoritmo VERIFICA-ASIGNACION usando $c(M)$ verifica en tiempo polinomial que M efectivamente se puede colorear con tres colores.

Para el problema HAMILTONIANO, dado un grafo Hamiltoniano G, el certificado $c(G)$ es una permutación de los vértices de G que formen un ciclo en G. En tiempo polinomial se puede chequear que $c(G)$ es efectivamente un ciclo en G que contiene a todos los vértices y por lo tanto que G es Hamiltoniano.

Para el problema BIPARTITO, dado un grafo bipartito G, el certificado $c(G)$ es el par de conjunto (V, U) que conforman la bipartición de G. Dados los conjuntos V y U se puede chequear en tiempo polinomial que G es bipartito, basta verificar que tanto V como U son conjuntos independientes en G.

Para el problema k-CLIQUE, dado un grafo G que posee un clique de tamaño k, el certificado $c(G)$ es un subconjunto K de vértices de G tal que $|K| = k$ y todos los vértices de K son vecinos mutuos. Dado K, se puede en tiempo polinomial verificar que su tamaño es k y que todos sus vértices son vecinos mutuos en G.

Para el problema EULERIANO, dado un grafo Euleriano G, el certificado sería simplemente el mismo G ya que en tiempo polinomial se puede verificar que G es Euleriano mirando los grados de cada uno de sus vértices.

Este último ejemplo nos indica cómo los problemas en P son siempre un caso particular de los problemas en *NP*. Podemos, a partir de la definición de la clase de problemas *NP*, definir la clase de problemas P, como un subconjunto de *NP*, de la siguiente forma: En P están todos los problemas π de *NP* tales que el certificado asociado a cada instancia positiva w de π es la misma instancia w, o sea, tales que $\forall w \in L_\pi \; c(w) = w$. De

esta última discusión obtenemos que

$$P \subseteq NP.$$

Sin embargo existen problemas en NP para los cuales no se sabe si hay una solución en tiempo polinomial como 3-MAP-COLOR por ejemplo, luego las siguientes son preguntas válidas

$$¿\ P = NP\ ? \qquad ¿\ P \neq NP\ ?$$

Si la primera de estas preguntas fuera cierta entonces sabríamos que todo problema en NP tendría una solución polinomial y entonces existiría un algoritmo polinomial para el problema 3-MAP-COLOR. Si en cambio la segunda de estas preguntas fuera cierta, entonces existiría **al menos un problema** en NP para el cual **no hay algoritmo polinomial** que lo resuelva. Veremos que si este último es el caso entonces existirían muchos problemas para los cuales no se podría encontrar un algoritmo polinomial que lo resuelva.

La pregunta acerca de si $P = NP$ o $P \neq NP$ es una de las preguntas abiertas más importantes (si no la más importante) en ciencia de la computación, y desde el momento en que se propuso en el año 1971, hasta la fecha no parece haber un avance significativo para poder responderla. Sin embargo, existe mucha más evidencia acerca de que $P \neq NP$ principalmente por la existencia de los llamados problemas NP-completos. Intuitivamente un problema NP-completo es un problema que es **más difícil que todos** los problemas en NP, en el sentido de que si se pudiera resolver en tiempo polinomial, entonces todos los problemas de NP se podrían resolver también en tiempo polinomial.

Def: Un problema de decisión π es NP-completo, si $\pi \in NP$ y, si se encontrara un algoritmo polinomial para resolver π, entonces se podría encontrar (a partir del algoritmo para π) un algoritmo polinomial para resolver cada uno de los problemas en NP.

Teorema 3.2.1: El problema 3-MAP-COLOR es NP-completo, el problema HAMILTONIANO es NP-completo, el problema k-CLIQUE es NP-completo, al igual que muchos otros problemas de decisión de interés práctico y teórico. Hasta la fecha siguen apareciendo nuevos problemas en las más diversas áreas que caen también dentro de esta categoría.

La implicación más importante de que un problema sea NP-completo es que si se llegara a encontrar un algoritmo polinomial para él, entonces se podría encontrar una solución polinomial para todos los problemas en NP. Por ejemplo, si alguien encontrara un algoritmo polinomial para resolver el problema 3-MAP-COLOR entonces inmediatamente se podría encontrar un algoritmo polinomial para HAMILTONIANO, k-CLIQUE, y todos los otros problemas NP-completos.

3.2.3. Reducción de Problemas y Demostraciones de *NP*-completitud**

[...falta completar...]

3.2.4. Otras Clases de Problemas Computacionales**

[...falta completar...]