Universidad de Huelva

Tema 7

Complejidad temporal

- En los temas anteriores demostramos que exiten problemas que pueden resolverse mediante Máquinas de Turing (o modelos de computación equivalentes) y otros que no pueden resolverse. Se definen como problemas ***decidibles*** y ***no-decidibles***.
- A partir de aquí vamos a estudiar la complejidad de los problemas decidibles.

- Podemos medir el tiempo que tarda una Máquina de Turing en resolver un problema como el número de pasos que ejecuta la máquina.
- Como el problema es decidible, sabemos que la máquina para y, por tanto, que el número de pasos es finito.
- Se denomina ***algoritmo*** al proceso que ejecuta una Máquina de Turing.
- El número de pasos que un algoritmo utiliza para analizar una entrada en particular puede depender de varios factores. Por ejemplo, si la entrada representa un grafo, la ejecución del algoritmo dependerá del número de nodos, del número de arcos, del nivel de profundidad, de si existen ciclos, …

- Vamos a centrar el estudio de la complejidad temporal en la dependencia del tiempo respecto al tamaño de la cadena de entrada, (que denotaremos n), despreciando otros factores.
- El ***análisis del peor caso*** considera el tiempo máximo que tarda el algoritmo para cadenas de entrada de un cierto tamaño.
- El ***análisis del caso promedio*** considera el valor medio del tiempo al ejecutar el algoritmo sobre todas las posibles entradas de un cierto tamaño.
- Vamos a centrarnos en un análisis del peor caso.

- Definición:

  Sea M una Máquina de Turing determinista que alcanza el estado de parada ante cualquier valor de entrada. Se define la ***complejidad temporal de M*** como una función f: $\mathbb{N} \rightarrow \mathbb{N}$, donde $f(n)$ es el máximo número de pasos que utiliza M para ejecutar una entrada de longitud n.

- Definición:

  Sean f y g funciones f,g: $\mathbb{N} \to \mathbb{R}^+$. Se dice que ***f* es de orden *g***, $f(n) = O(g(n))$ si existe los números enteros positivos c y n_0 tales que, para cada $n > n_0$

  $$f(n) \leq c \cdot g(n)$$

  Se dice también que $f(n)$ **está asintóticamente acotado por** $g(n)$ o que $g(n)$ **es una cota superior asintótica de** $f(n)$.

- Ejemplo: $f_1(n) = 5n^3+2n^2+22n+6$.

  $f_1(n) = O(n^3)$ ya que para $n>10$, $f_1(n) < 6n^3$

- Además, también se cumple que $f_1(n) = O(n^4)$, $f_1(n) = O(n^5)$, …

- Se denominan **órdenes polinómicos** a cotas de tipo $O(n^c)$ siendo c una constante.
- Se denomina **orden logarítmico** a la cota $O(\log n)$. En este caso no es necesario expresar la base del logaritmo, ya que la relación entre logaritmos de base diferente es una constante que queda integrada en la cota.
- Se denomina **orden exponencial** a las cotas de tipo $O(2^m)$ con $m=n^c$.
- La expresión $2^{O(\log n)}$ denota un valor $2^{(c \log_2 n)}$ que equivale a decir un orden $O(n^c)$ para un valor c desconocido.

- La **notación** $O(\cdot)$ expresa que una función es asintóticamente no mayor que otra. Para indicar que una función es asintóticamente menor que otra se utiliza la **notación** $o(\cdot)$ que se define como:

  Sean f y g funciones $f,g: \mathbb{N} \to \mathbb{R}^+$. Se dice que $f(n) = o(g(n))$ si

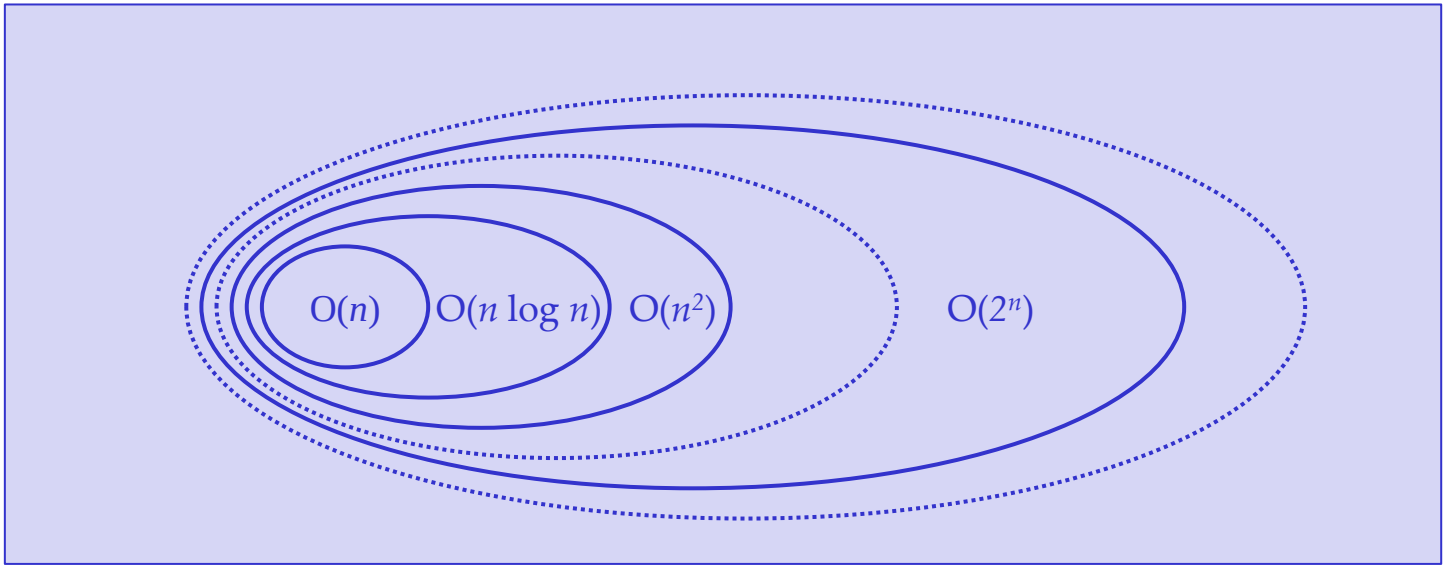
$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

- Aquí utilizaremos siempre el orden de las funciones como **notación** $O(\cdot)$.

- Definición:

  Sea t una función t: $\mathbb{N} \rightarrow \mathbb{R}^+$. Se define la **clase de complejidad temporal** **TIME**($t(n)$) como el conjunto de todos los problemas o lenguajes decidibles mediante una Máquina de Turing determinista de orden $O(t(n))$.

- Podemos representar estas clases mediante un diagrama de conjuntos.



- Ejemplo:

  Algoritmo para reconocer el lenguaje A= { $0^k 1^k \mid k \geq 0$ }

#	0	0	0	0	1	1	1	1	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~

1.- Avanzar a la derecha.
2.- Si encuentra 1, ***rechazar***. Si encuentra 0, seguir. En otro caso, ***aceptar***.
3.- Marcar el 0. (Sustituir el 0 por el símbolo *)
4.- Avanzar hacia la derecha buscando el primer 1. Si se encuentra ~~b~~, ***rechazar***.
5.- Marcar el símbolo 1. (Sustituir el 1 por el símbolo %).
6.- Avanzar a la izquierda hasta encontrar el símbolo *. Ir a paso 1.

Para cada símbolo 0, el algoritmo avanza a la derecha buscando un 1 y vuelve a la izquierda a la posición inicial dando n pasos. El número de 0s que debe cotejar es $n/2$, así que en total el algoritmo tardará un tiempo $O(n^2)$.

- Ejemplo:

  Algoritmo alternativo para reconocer el lenguaje A= { $0^k 1^k \mid k \geq 0$ }

#	0	0	0	0	1	1	1	1	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~

1.- Recorrer la cinta. Si se encuentra un 0 a la derecha de un 1, **rechazar**.
2.- Repetir mientras queden 0s y 1s:
3.- Recorrer la lista verificando si el número de 0s y 1s es par o impar.
Si es impar, ***rechazar***.
4.- Recorrer la lista marcando (quitando) la mitad de 0s y la mitad de 1s.
5.- Si no quedan 0s ni 1s, ***aceptar***. En otro caso, ***rechazar***.

Cada iteración de los pasos 3 y 4 requiere n pasos. Sin embargo, el número de iteraciones necesario es $\log_2 n$ ya que en cada iteración el número de 0s y 1s se reduce a la mitad. Por tanto, el algoritmo es $O(n \log n)$.

- Ejemplo:

  Algoritmo para reconocer el lenguaje A= { $0^k\,1^k \mid k \geq 0$ } sobre una máquina de dos cintas.

#	0	0	0	0	1	1	1	1	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~
#	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~	~~b~~

1.- Recorrer la primera cinta leyendo 0s y copiandolos en la segunda cinta.
2.- Al llegar a los 1s, recorrer la primera cinta leyendo 1s y eliminando los 0s de la segunda cinta.
3.- Si detrás de los 1s aparece algún 0, ***rechazar***.
Si al finalizar los 1s quedan 0s en la segunda cinta, ***rechazar***.
Si al recorrer los 1s se llega al final de la segunda cinta, ***rechazar***.
En otro caso, ***aceptar***.

En este caso sólo es necesario recorrer la primera cinta una vez, así que el algoritmo es $O(n)$.

- Conclusión. El modelo de computación utilizado influye en la complejidad temporal de los problemas. El mismo problema en una Máquina de Turing de dos cintas puede resolverse en $O(n)$, pero necesita un algoritmo de $O(n \log n)$ en una Máquina de Turing de una cinta.
- Según la **Tesis de Church-Turing**, todos los modelos de computación lo suficientemente complejos tienen la misma capacidad de resolución que las Máquinas de Turing de una cinta. Por ejemplo, las MT multicinta, las MT indeterministas, el cálculo lambda, las funciones recursivas, …
- Un problema decidible puede resolverse en cualquiera de estos modelos, pero lo complejidad temporal puede ser distinta en función del modelo utilizado.
- ¿Podemos relacionar la complejidad temporal de distintos modelos?

- Teorema:

  Todo problema de orden $O(t(n))$ en una Máquina de Turing multicinta puede ser resuelto por un algoritmo de orden $O(t^2(n))$ en una Máquina de Turing de una cinta.

- Demostración. En el tema 4, apartado 4, mostramos un algoritmo que simulaba el funcionamiento de una Máquina de Turing multicinta sobre una Máquina de Turing de una cinta. Para simular t pasos de la máquina multicinta se requerían t^2 pasos en la máquina de una cinta.

- Observación: Eso no quiere decir que sea imposible encontrar un algoritmo mejor en la Máquina de Turing de una cinta.

- Teorema:

  Todo problema de orden $O(t(n))$ en una Máquina de Turing indeterminista puede ser resuelto por un algoritmo de orden $2^{O(t(n))}$ en una Máquina de Turing de una cinta.

- Demostración. En el tema 4, apartado 5, mostramos un algoritmo que simulaba el funcionamiento de una Máquina de Turing indeterminista sobre una Máquina de Turing de tres cintas. Para simular t pasos de la máquina indeterminsita se requerían 2^t pasos en la máquina determinista de tres cintas y $(2^t)^2$ pasos en una máquina equivalente de una cinta, pero $(2^t)^2$ es 2^{2t} que es de orden $2^{O(t)}$.

- Observación: Eso no quiere decir que sea imposible encontrar una algoritmo mejor en la Máquina de Turing de una cinta.

- Se define la **clase P** como el conjunto de problemas que son decidibles en tiempo polinomial sobre una Máquina de Turing determinista de una cinta.

$$P = \bigcup_k TIME(n^k)$$

- La clase P es invariante para todos los modelos de computación que puedan simularse en tiempo polinomial sobre una Máquina de Turing determinista de una cinta. (Por ejemplo, las Máquinas de Turing multicinta).
- Los problemas de complejidad exponencial terminan siendo intratables para valores altos de n. Por ejemplo para n=1000, 2^{1000} es mucho mayor que el número de átomos que se calcula que existen en el universo.
- NOTA: El número de átomos del universo se estima entre $4x10^{78}$ y $6x10^{79}$, es decir, entre 2^{261} y 2^{265}.

- A la hora de analizar los problemas es importante considerar no solo el modelo de computación a utilizar sino también la codificación utilizada para representar las entradas.
- Por ejemplo, un número natural k puede ocupar un tamaño k en notación unaria (1111…1) o un tamaño $\log_2 k$ en notación binaria (100101).
- Para representar grafos con n nodos y e arcos (el número máximo de arcos es $O(n^2)$ y para codificar los nodos se necesitan $\log_2 n$ bits) podemos utilizar una lista de arcos (con un tamaño máximo de $O(n^2 \log_2 n)$ o una matriz de adyacencia (con un tamaño fijo de $O(n^2)$). En cualquier caso, el tamaño de la entrada depende polinómicamente del número de nodos.

- Teorema

  Sea PATH el lenguaje formado por las cadenas <G,s,t> donde G es un grafo dirigido, s y t son nodos del gafo y existe un camino que conecta s con t. El lenguaje PATH pertenece a la clase P.

- Demostración

  Hay que encontrar un algoritmo que resuelva el problema en tiempo polinomial.

  Si lo intentamos por fuerza bruta, habría que generar todos los caminos que comiencen en s y estudiar si alguno llega a t. Como mucho la longitud del camino será n (número de nodos) y el número de caminos distintos es n^n, es decir, de orden exponencial. Por tanto hay que buscar otro algoritmo que no se base en fuerza bruta.

- Demostración (PATH $\in$ P)

  El siguiente algoritmo se basa en construir la lista de nodos accesibles desde s. Para ello se crea una lista L1 de nodos accesibles y cuyas conexiones no se han analizado y una lista L2 de nodos accesibles cuyas conexiones ya se han analizado.

  1.- Añadir el nodo *s* a L1.
  2.- Repetir mientras la lista L1 no esté vacía:
  3.- Escoger el primer nodo, *a*, de la lista L1.
  4.- Estudiar sus conexiones (*a*,*b*) añadiendo a L1 los nodos *b* que no estén en L1 ni L2.
  5.- Borrar *a* de L1 y añadirlo a L2.
  6.- Si el nodo *t* pertenece a L2, ***aceptar***. Si no, ***rechazar***.

  En el peor caso, el bucle se ejecutará *n* veces (si en cada ocasión eliminamos un nodo y añadimos otro). El punto 4 depende de la notación utilizada, pero en cualquier caso requiere un tiempo polinomial. Por tanto el algoritmo es de orden polinomial.

- Teorema

  Sea RELPRIME el lenguaje formado por las cadenas $<x,y>$ donde x e y son números naturales primos entre sí, es decir, su máximo común denominador es 1. El lenguaje RELPRIME pertenece a la clase P.

- Demostración

  Si los números se describen en notación binaria, la longitud de la entrada es $n=\log_2 x+\log_2 y$.

  Si lo intentamos por fuerza bruta, habría que estudiar todos los números entre 1 y $\min(x,y)$ buscando un número que sea divisor de x y de y. El número de iteraciones sería $\min(x,y)$, que es de orden $O(2^n)$.

- Demostración (RELPRIME$\in$ P)

  Se puede calcular el máximo común divisor entre dos números por medio del algoritmo de Euclides.

  1.- Repetir mientras $y \neq 0$:
  2.- $x \leftarrow x \bmod y$.
  3.- Intercambiar x e y.
  4.- Si $x = 1$, ***aceptar***. Si no, ***rechazar***.

  En cada ejecución del paso 2 el valor de x se devide al menos a la mitad. En el peor caso el valor de x se irá dividiendo por dos hasta llegar a la unidad. El número de pasos necesario para hacer eso es $\log_2 x$, es decir, el algoritmo es de orden lineal $O(n)$.

- Teorema

  Todos los lenguajes libres de contexto pertenecen a P.

- Demostración

  Todo lenguaje libre de contexto puede describirse mediante una gramática en forma normal de Chomsky.

  Si lo intentamos por fuerza bruta, dada una cadena de entrada w de longitud n habría que estudiar todos las posibles derivaciones de longitud n que puedan contruirse a partir del símbolo inicial y verificar si alguna de ellas es la cadena w. El número de derivaciones de tamaño n para una gramática en forma normal de Chomsky con r reglas es de orden exponencial $O(r^n)$.

- Demostración (CFL $\in$ P) Algoritmo de Cocke-Younger-Kasami

  Para verificar que una cadena de entrada $w=w_1 \ldots w_n$ pertenece al lenguaje vamos a construir una tabla $n \times n$ en la que en la celda (i,j) se almacenarán los símbolos no terminales que permiten derivar la subcadena $w_i..w_j$.

1.- Si $w = \lambda$ y $S \rightarrow \lambda$ es una regla de la gramática, ***aceptar***.
2.- Para i =1 hasta n.
3.- Para cada símbolo no terminal A
4.- Si existe la regla $A \rightarrow w_i$, añadir A a la celda tabla(i,i)
5.- Para l=2 hasta n.
6.- Para i = 1 hasta n-l+1
7.- Sea $j = i + l - 1$
8.- Para $k = i$ hasta j-1
9.- Para cada regla $A \rightarrow BC$
10.- Si tabla(i,k) contiene B y tabla(k+1,j) contiene C, añadir A a tabla(i,j)
11.- Si S pertenece a tabla(1,n), ***aceptar***. Si no, ***rechazar***.

- Demostración (CFL $\in$ P) Algoritmo de Cocke-Younger-Kasami

  Sea v el número de símbolos no terminales y r el número de reglas de la gramática en formal normal de Chomsky.

  La instrucción 4, dentro del bucle 2 se repite $v \cdot n$ veces. La instrucción 10, dentro del bucle 5 se repite $r \cdot n^3$ veces. Por tanto, el algoritmo es de orden $O(n^3)$.

- Sea t una función t: $\mathbb{N} \rightarrow \mathbb{R}^+$. Se define la **clase de complejidad temporal NTIME**($t(n)$) como el conjunto de todos los problemas o lenguajes decidibles mediante una Máquina de Turing indeterminista de orden $O(t(n))$.

- Se define la **clase NP** como el conjunto de problemas que son decidibles en tiempo polinomial sobre una Máquina de Turing indeterminista.

$$NP = \bigcup_k NTIME(n^k)$$

- Un **verificador** de un lenguaje A es un algoritmo V tal que

$$A = \{ w \mid V \text{ acepta } \langle w,c \rangle \text{ para alguna cadena } c \}$$

es decir, w pertenece a A si existe una cadena c tal que el algoritmo V acepta $\langle w,c \rangle$.

$$w \in A \iff \exists\, c \mid V(w,c) \text{ se acepta}$$

- Se dice que V es un **verificador en tiempo polinomial** si su complejidad temporal depende polinomialmente de la longitud de w.
- Se dice que un lenguaje A es **verificable polinomialmente** si existe un verificador en tiempo polinomial para ese lenguaje.

- Teorema:

  Un lenguaje A es NP si y solo si es verificable polinomialmente.

- Demostración: ($\Longleftarrow$)

  Si A es verificable polinomialmente, entonces existe un verificador V en tiempo polinomial $O(n^k)$. Podemos construir una Máquina de Turing indeterminista en tiempo polinomial de la siguiente forma:

  Dada w de longitud n:

  1.- Generar de forma indeterminista una cadena c de longitud máxima n^k.

  2.- Ejecutar V sobre $<w,c>$

  3.- Si V acepta, ***aceptar***. Si no, ***rechazar***.

- Teorema:

  Un lenguaje A es NP si y solo si es verificable polinomialmente.

- Demostración: ($\Longrightarrow$)

  Si A es NP entonces existe una Máquina de Turing no determinista (MT) que decide en tiempo polinomial $O(n^k)$. Podemos construir un verificador V en tiempo polinomial $O(n^k)$ de la siguiente forma:

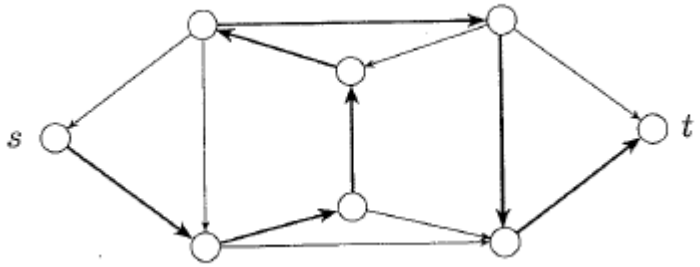
  Dadas $<w,c>$:

  1.- Ejecutar MT sobre w, tratando cada símbolo de c como la elección de cada paso indeterminista de MT.

  2.- Si el camino seleccionado por c acepta w, ***aceptar***. Si no, ***rechazar***.

- Teorema

  Dado un grafo dirigido *G*, se define un **camino Hamiltoniano** como un camino que pasa por todos los nodos del grafo una sola vez. Sea HAMPATH el lenguaje formado por las cadenas <*G*,*s*,*t*> donde *G* es un grafo dirigido y *s* y *t* son nodos del grafo. El lenguaje HAMPATH pertenece a la clase NP.



- Demostración (HAMPATH $\in$ NP)

  Se puede demostrar generando una MT indeterminista con complejidad polinomial o un verificador polinomial.

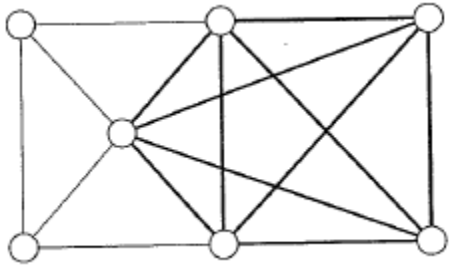
  Verificador:

> 1.- Dada $w=<G,s,t>$, sea c una cadena formada por nodos del grafo $c=p_1,\ldots,p_n$ siendo n el número de nodos del grafo.
> 2.- Si algún nodo aparece duplicado en c, ***rechazar***.
> 3.- Si $p_1 \neq s$ o $p_n \neq t$, ***rechazar***.
> 4.- Para cada p_i, si G no contiene el arco (p_i, p_{i+1}), ***rechazar***.
> 5.- Si todos los pasos anteriores se han verificado, ***aceptar***.

  Todos los pasos del verificador pueden ejecutarse en tiempo polinomial. Por tanto, el verificador es polinomial.

- Teorema

  Dado un grafo no dirigido *G*, se define un **clique** como un subgrafo en el que todos los nodos están conectados entre sí. Un **k-clique** es un clique formado por *k* nodos. Sea CLIQUE el lenguaje formado por las cadenas <*G*,*k*> donde *G* es un grafo no dirigido que contiene un clique de *k* nodos. El lenguaje CLIQUE pertenece a la clase NP.



- Demostración (CLIQUE$\in$ NP)

  Se puede demostrar generando una MT indeterminista con complejidad polinomial o un verificador polinomial.

  Verificador:

1.- Dada $w=<G,k>$, sea c una cadena formada por k nodos del grafo $c=p_1,\ldots,p_k$.
2.- Si algún nodo aparece duplicado en c, ***rechazar***.
4.- Para cada p_i y p_j, si G no contiene el arco (p_i, p_j), ***rechazar***.
5.- Si todos los pasos anteriores se han verificado, ***aceptar***.

- La clase P contiene los problemas que pueden ser resueltos en tiempo polinomial, es decir, que se puede encontrar la solución en tiempo polinomial.
- La clase NP contiene los problemas que pueden ser verificados en tiempo polinomial, es decir, que podemos comprobar una solución en tiempo polinomial (aunque no sepamos cuanto puede costar encontrarla).
- Aunque parece intuitivo que P y NP deben ser conjuntos diferentes, hasta ahora ha sido imposible demostrar que P ≠ NP. Eso requiere demostrar que para algún problema es imposible sustituir la búsqueda por fuerza bruta por algún otro algoritmo de búsqueda más rápido.
- Que no hayamos encontrado un algoritmo más rápido no quiere decir que no exista. Para demostrar que P ≠ NP hay que demostrar que ese algoritmo no existe.

- Para demostrar que P ≠ NP bastaría con demostrar que un determinado problema NP no tiene solución en tiempo polinomial.
- Stephen Cook y Leonid Levin demostraron que existen una serie de problemas pertenecientes a NP que, si tuvieran solución en tiempo polinomial, permitirían demostrar que todos los problemas NP tienen solución en tiempo polinomial. Estos problemas se denominan **NP-completos**.
- Los problemas NP-completos son importantes porque:
  1. Si algún problema NP no tiene solución en tiempo polinomial, entonces ningún problema NP-completo puede tener solución en tiempo polinomial.
  2. Si algún problema NP-completo tiene solución en tiempo polinomial, entonces todos los problemas NP tienen solución en tiempo polinomial.
  3. Si asumimos que P ≠ NP (aunque aún no se haya demostrado), entonces demostrar que un problema es NP-completo equivale a demostrar que no tiene solución en tiempo polinomial.

- El problema de satisfactibilidad

  Una **variable booleana** es una variable que puede tomar los valores ***verdadero*** o ***falso***.

  Una **fórmula booleana** es una expresión fomada por variables booleanas y operadores lógicos AND, OR y NOT.

$$(\bar{x} \wedge y) \vee (x \wedge \bar{z})$$

  Una fórmula booleana es **satisfactible** si existen valores de las variables booleanas que hacen que la expresión sea cierta. (Por ejemplo: x=0; y=1; z=0)

  Se denomina **problema de satisfactibilidad** a la demostración de que una fórmula es satisfactible.

- Teorema de Cook-Levin

  Sea *SAT* el lenguaje formado por las fórmulas booleanas satisfactibles. El teorema de Cook-Levin establece que

$$SAT \in P \Leftrightarrow P = NP$$

  Podemos interpretar el teorema como que SAT es NP-completo.

- Definiciones

  Se dice que una función $f:\Sigma^* \rightarrow \Sigma^*$ es **computable en tiempo polinomial** si existe una Máquina de Turing, M, que calcula la función en tiempo polinomial.

  Se dice que **un lenguaje A es reducible polinomialmente a un lenguaje B**, denotado como $A \leq_p B$, si existe una función f computable en tiempo polinomial tal que

$$w \in A \Leftrightarrow f(w) \in B$$

- Teorema

  Si $A \leq_p B$, y $B \in P$ entonces $A \in P$

- Demostración

  Si $B \in P$ entonces existe una Máquina de Turing, M, de orden polinomial que resuelve B. Si es así podemos definir una nueva Máquina de Turing que resuelva A en tiempo polinomial:

  1. Dada w, calcular $f(w)$.
  2. Aplicar M a $f(w)$. Si se acepta, ***aceptar***; si se rechaza, ***rechazar***.

- Definición

  Un lenguaje B es NP-completo si $B \in NP$ y todo $A \in NP$ es reducible en tiempo polinomial a B.

- Teorema

  Si B es NP-completo, $C \in NP$ y $B \leq_p C$ entonces C es NP-completo.

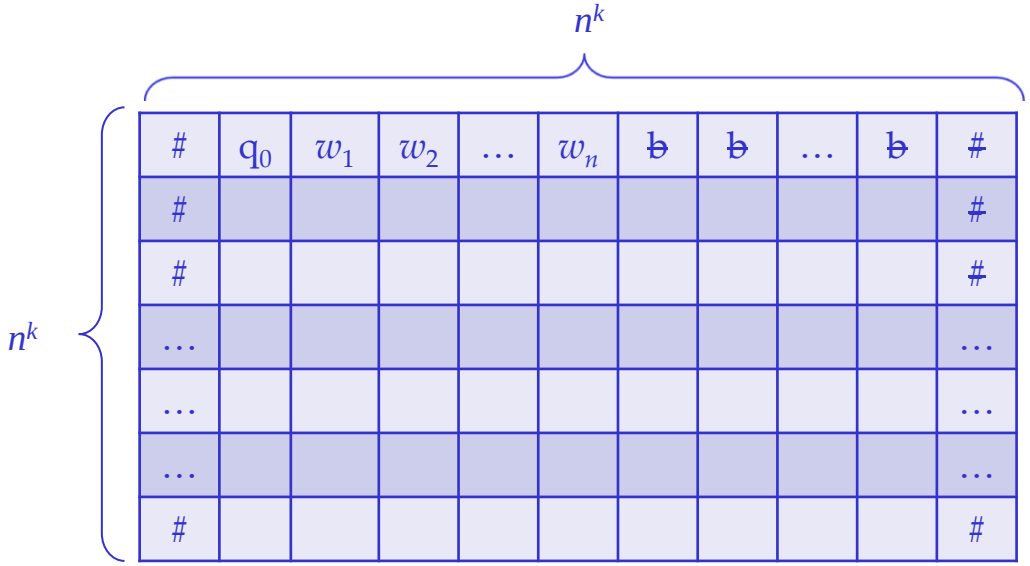
- Demostración

  Si B es NP-completo entonces todo $A \in NP$ es reducible en tiempo polinomial a B. Si B es reducible en tiempo polinomial a C, entonces A es reducible en tiempo polinomial a C.

- Teorema de Cook-Levin (*SAT* es NP-completo)

  Dado un lenguaje $A \in NP$, existe una Máquina de Turing no determinista, N, que resuelve el lenguaje en tiempo polinomial n^k. Es decir, para una entrada w de longitud n el número máximo de iteraciones de N es n^k.

  Vamos a definir un tablero basado en las configuraciones del proceso de ejecución de N:



- Teorema de Cook-Levin (*SAT* es NP-completo)

  Cada fila del tablero contiene una configuración de la máquina N.

  La primera fila contiene la configuración incial y las siguientes filas corresponden a una transición de la máquina.

  Un tablero se acepta si contiene una configuración que alcance el estado de aceptación.

  Un tablero aceptado corresponde a una ejecución de la máquina N que conduce a un estado de aceptación.

  Calcular si la máquina N acepta una entrada w equivale a calcular si existe un tablero aceptable.

  Vamos a representar el problema de encontrar un tablero aceptable como un problema de satisfacción de una fórmula lógica.

- Teorema de Cook-Levin (*SAT* es NP-completo)

Sea Q el conjunto de estados de N y Σ el alfabeto del lenguaje. El conjunto de símbolos que puede aparecer en cada celda del tablero es $C = Q \cup \Sigma \cup \{\#, \not{b}\}$. El tamaño de C no depende de la longitud de la entrada (n).

Se define la variable booleana x_{ijc} como una variable que indica que en la celda (i,j) se encuentra el símbolo c.

La fórmula lógica asociada al tablero completo está formada por cuatro partes

$$\varphi = \varphi_{cell} \wedge \varphi_{start} \wedge \varphi_{move} \wedge \varphi_{accept}.$$

- Teorema de Cook-Levin (*SAT* es NP-completo)

La fórmula φ_{cell} corresponde a la exigencia de que para cada celda (i,j) una y solo una de las variables booleanas x_{ijc} puede ser cierta.

$$\phi_{cell} = \bigwedge_{i,j}((\bigvee_{c} x_{ijc}) \wedge (\bigwedge_{s \neq t}(\overline{x_{ijs}} \vee \overline{x_{ijt}})))$$

Teniendo en cuenta que el número de celdas es $(n^k)^2$, este término es de orden $O(n^{2k})$.

- Teorema de Cook-Levin (*SAT* es NP-completo)

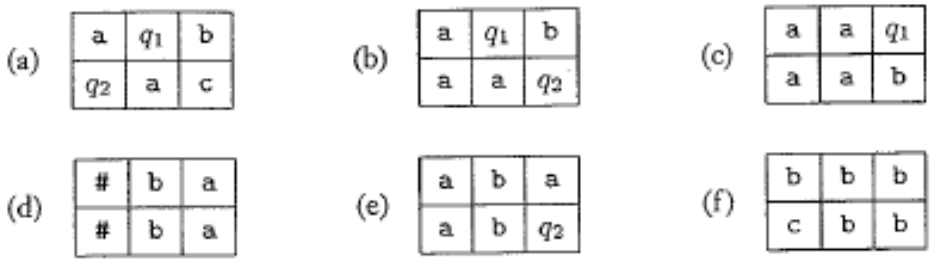
La fórmula φ_{start} corresponde a la descripción de la primera fila:

$$\phi_{start} = x_{1,1,\#} \wedge x_{1,2,q_0} \wedge x_{1,3,w_1} \wedge \ldots \wedge x_{1,n+2,w_n} \wedge x_{1,n+3,b} \wedge \ldots \wedge x_{1,n^k,\#}$$

Este término es de orden $O(n^k)$.

- Teorema de Cook-Levin (*SAT* es NP-completo)

La fórmula φ_{move} corresponde a la descripción de los valores aceptables de trozos de 3x2 celdas. Esto permite describir las transiciones de la máquina de Turing. Por ejemplo, los siguientes grupos podrían ser aceptables:



Este término es de orden $O(n^{2k})$, aunque la constante pueda ser muy grande (como mucho c^6).

- Teorema de Cook-Levin (*SAT* es NP-completo)

La fórmula φ_{accept} indica que para que un tablero se acepte, al menos una celda debe corresponder al estado de aceptación:

$$\phi_{accept} = \bigvee_{i,j} x_{i,j,q_{accept}}$$

Este término es de orden $O(n^{2k})$.

- Teorema de Cook-Levin (*SAT* es NP-completo)

Por tanto, cualquier problema NP puede transformarse en buscar una solución al problema SAT sobre una fórmula lógica φ. Como la fórmula lógica es de orden $O(n^{2k})$, esto quiere decir que cualquier problema NP puede reducirse en tiempo polinomial a un problema SAT. Por tanto, SAT es NP-completo.

- Teorema: *3SAT* es NP-completo

Dentro de una fórmula lógica, se denomina **literal** a una variable booleana o a la negación de una variable booleana.

Se denomina **clausula** a una lista de literales conectados mediante operadores OR. Por ejemplo, $(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4)$

Se dice que una fórmula lógica está en **forma normal conjuntiva** si está formada por una lista de claúsulas conectadas mediante el operador AND. Por ejemplo, $(x_1 \vee \bar{x}_2 \vee \bar{x}_3 \vee x_4) \wedge (x_3 \vee \bar{x}_5) \wedge (x_2 \vee \bar{x}_3 \vee \bar{x}_6)$

Toda fórmula lógica φ puede escribirse en forma normal conjuntiva.

- Teorema: *3SAT* es NP-completo

Se dice que una fórmula lógica es una **fórmula-cnf** si está escrita en forma normal conjuntiva.

Se dice que una fórmula lógica es una **fórmula-3cnf** si todas sus claúsulas tienen tres literales.

Se define *3SAT* como el lenguaje formado por las fórmulas-3cnf satisfactibles.

- Teorema: *3SAT* es NP-completo

Se puede demostrar que 3SAT es NP-completo modificando levemente la demostración de que SAT es NP-completo. Para eso es necesario escribir las fórmulas φ_{cell}, φ_{start}, φ_{move} y φ_{accept} como fórmulas-3cnf.

La fórmula φ_{start} ya la hemos descrito como fórmula-3cnf.

La fórmula φ_{cell} contiene una claúsula ($\vee x_{ijc}$) que tiene más de tres literales. Este tipo de claúsulas puede transformarse en una conjunción de claúsulas de tres literales aplicando la propiedad

$$(a \vee b \vee c \vee d) \equiv (a \vee b \vee z) \wedge (\bar{z} \vee c \vee d)$$

es decir, añadiendo nuevas variables booleanas "z" para trocear las claúsulas de más de tres literales.

- Teorema: *3SAT* es NP-completo

La fórmula φ_{accept} está formada por una única claúsula que puede expresarse como 3cnf aplicando la propiedad anterior.

La fórmula φ_{move} es la que requiere más modificaciones, ya que no la hemos descrito en forma conjuntiva. Sin embargo podemos transformar su expresión a forma normal conjuntiva y aplicar la propiedad anterior para asegurar que se expresa como 3cnf.

Por tanto, todo problema NP puede transformarse en una problema 3SAT luego 3SAT es un problema NP-completo.

- Teorema: *CLIQUE* es NP-completo

Podemos demostrarlo si encontramos una reducción del problema 3SAT a CLIQUE en tiempo polinomial.

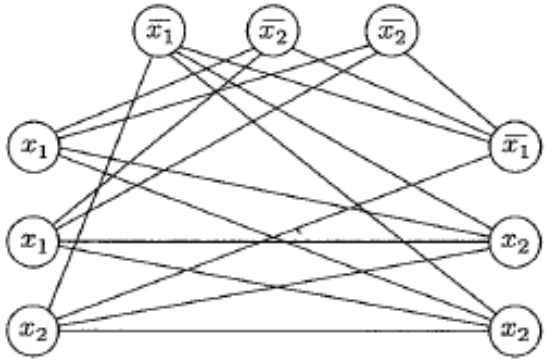
Dada una fórmula-3cnf compuesta de k claúsulas $(a \vee b \vee c)$, se puede construir un grafo G donde cada literal de cada claúsula genera un nodo y donde cada nodo está enlazado con todos los nodos del grafo excepto los de su cláusula y los contradictorios (por ejemplo, a y ā).

- Teorema: *CLIQUE* es NP-completo

Por ejemplo, la fórmula-3cnf

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$$

puede representarse por el grafo



- Teorema: *CLIQUE* es NP-completo

  Dada una fórmula con k claúsulas, el tamaño del grafo generado tendrá 3·k nodos y O(k2) arcos. Por tanto se puede construir el grafo en un tiempo polinómico.

  La fórmula es satisfactible si y solo si el grafo tiene un k-clique. Ese k-clique corresponde a la solución de la fórmula. Es decir, si la variable booleana x aparece en el k-clique entonces la solución contiene el valor x=TRUE. Si el k-clique contiene la negación de x, entonces la solución contiene el valor x=FALSE.

- Teorema: *VERTEX-COVER* es NP-completo

Dado un grafo no dirigido, ***G***, se define un ***Vertex-Cover*** de G, ***VC***, como un subconjunto de nodos de G tales que cualquier arco de G tiene al menos un nodo en VC.

$$\langle a,b\rangle \in G \Rightarrow (a \in VC) \vee (b \in VC)$$

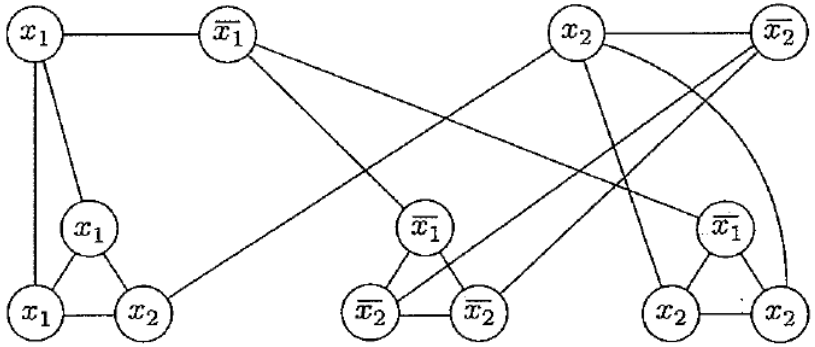
Se define el lenguaje VERTEX-COVER como el lenguaje formado por cadenas <*G*,*k*> donde *G* es un grafo no dirigido que contiene un Vertex-Cover de *k* nodos.

- Teorema: *VERTEX-COVER* es NP-completo

Para demostrar que el problema VERTEX-COVER es NP-Completo vamos a demostrar que cualquier problema 3SAT puede transformarse en tiempo polinomial en un problema VERTEX-COVER.

Dada un fórmula lógica escrita en forma 3-cnf se puede generar un grafo G donde para cada variable booleana a generamos dos nodos, a y $\bar{a}$ conectados entre sí, y para cada cláusula (a,b,c) generamos tres nodos, a, b y c conectados entre sí. El grafo se completa conectando los nodos de las variables con sus ocurrencias en las cláusulas.

- Teorema: *VERTEX-COVER* es NP-completo



Ejemplo para la fórmula

$$\phi = (x_1 \vee x_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_2) \wedge (\bar{x}_1 \vee x_2 \vee x_2)$$

- Teorema: *VERTEX-COVER* es NP-completo

Si la fórmula ϕ tiene m variables y l cláusulas el grafo G tiene $2m+3l$ nodos.

El problema 3-SAT(ϕ) puede reducirse al problema VertexCover(<G,k>) con $k=m+2l$.

Si el problema 3SAT tiene solución, cada cláusula contiene un literal TRUE. En ese caso se puede obtener un VERTEX-COVER del grafo asociado eligiendo los dos literales restantes de cada cláusula y el nodo asociado a la variable booleana.

Si el problema VertexCover(<G,k>) tiene solución entonces el VC debe tener un nodo por cada variable y dos nodos por cada cláusula, lo que permite obtener la solución de 3-SAT.

- Teorema: *HAMPATH* es NP-completo

- Teorema: *UHAMPATH* es NP-completo

- Teorema: *SUBSET-SUM* es NP-completo

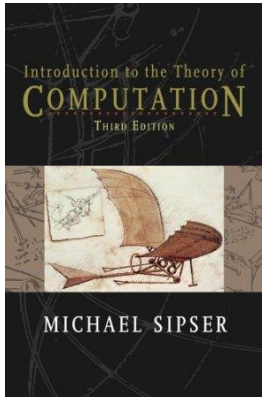
Sea S un conjunto de números enteros. Dado un número entero t, el problema SUBSET-SUM consiste en encontrar un subconjunto de S cuyos elementos sumen t.

Se puede demostrar que SUBSET-SUM es NP-completo reduciendo el problema 3SAT al problema SUBSET-SUM en tiempo polinomial.

- Teorema: *SUBSET-SUM* es NP-completo

Dada una fórmula ϕ con l variables y k cláusulas se puede formar un conjunto S con dos números (y, z) por cada variable y dos números (g, h) por cada cláusula como muestra la tabla.

	1	2	3	4	$\cdots$	l	c_1	c_2	$\cdots$	c_k
y_1	1	0	0	0	$\cdots$	0	1	0	$\cdots$	0
z_1	1	0	0	0	$\cdots$	0	0	0	$\cdots$	0
y_2		1	0	0	$\cdots$	0	0	1	$\cdots$	0
z_2		1	0	0	$\cdots$	0	1	0	$\cdots$	0
y_3			1	0	$\cdots$	0	1	1	$\cdots$	0
z_3			1	0	$\cdots$	0	0	0	$\cdots$	1
$\vdots$					$\ddots$	$\vdots$	$\vdots$		$\vdots$	$\vdots$
y_l						1	0	0	$\cdots$	0
z_l						1	0	0	$\cdots$	0
g_1							1	0	$\cdots$	0
h_1							1	0	$\cdots$	0
g_2								1	$\cdots$	0
h_2								1	$\cdots$	0
$\vdots$									$\ddots$	$\vdots$
g_k										1
h_k										1
t	1	1	1	1	$\cdots$	1	3	3	$\cdots$	3



- Ver *"Michael Sipser. Introduction to the Theory of Computation"* – Capítulo 7.