Ficha 29

Teoría de la Complejidad

1.] Introducción.

Los programadores en particular, y los investigadores de las ciencias de la computación en general, saben que existen problemas más difíciles de resolver que otros. En este contexto, indicar que un problema es *más difícil que otro* hace referencia a la *cantidad de recursos* (tiempo de ejecución o memoria) que son necesarios para alcanzar una solución, suponiendo que tal solución exista, y no a la naturaleza o estructura lógica de los algoritmos diseñados. Así por ejemplo, si la única solución que se conoce para un problema *p1* es de tiempo de ejecución $O(2^n)$ (exponencial) y para otro problema *p2* se sabe que existen soluciones cuyo tiempo está en $O(n^2)$ u $O(n^3)$, entonces diremos que *p1* es *inherentemente más difícil de resolver* que *p2*.

La *Teoría de la Complejidad Computacional* es la rama de las Ciencias de la Computación que estudia y clasifica a los problemas de acuerdo a su *complejidad inherente*. La medición de la complejidad inherente se hace con técnicas propias del *análisis de algoritmos*, que por lo tanto constituye una disciplina relacionada con la Teoría de la Complejidad, aunque sus objetivos son diferentes: el análisis de algoritmos busca determinar la cantidad de recursos que emplea un algoritmo, mientras que la Teoría de la Complejidad intenta caracterizar a todos los posibles algoritmos para resolver un problema dado (o más precisamente, determinar qué problemas pueden o no ser resueltos empleando una cierta cantidad limitada de recursos) [1].

Independientemente de su complejidad inherente y las clasificaciones que surjan a partir de ella, es conveniente hacer una primera distinción entre tipos de problemas. En general, de acuerdo al objetivo a cumplir, los problemas pueden clasificarse al menos en tres tipos:

- ***Problemas de decisión***: son aquellos que sólo admiten como resultado un valor de verdad (*true* o *false, sí* o *no*). Ejemplos de problemas de decisión son algunos como: ¿Es primo el número *x*? o ¿hay forma de recorrer un conjunto dado de *n* ciudades sin pasar dos veces por la misma ciudad?
- ***Problemas de búsqueda de resultados***: son aquellos que proponen encontrar una solución específica (si existe) para el planteo dado. Ejemplos de problemas de búsqueda son: hallar las raíces de una ecuación de segundo grado dados los coeficientes de la misma, dados n números, obtener el promedio de esos números, etc. En inglés se los suele designar como *function problems*.
- ***Problemas de optimización***: son aquellos en los que se pretende encontrar *la mejor solución posible* (y no necesariamente *cualquier solución*) para el problema analizado. Algunos ejemplos son: ¿cuál es la ruta más corta que permite visitar *n* ciudades sin pasar dos veces por la misma? o ¿cuál es la combinación de mayor valor monetario en libros que se puede comprar, sabiendo el precio de cada uno, pero sabiendo también que el peso acumulado de los libros comprados no debe superar el peso máximo admitido para una valija en un avión?

Como dijimos, la Teoría de la Complejidad intenta estudiar y clasificar a los problemas de acuerdo a su complejidad o dificultad inherente. En ese sentido, entonces, el impacto del tiempo de ejecución juega un papel preponderante ya que parece lógico clasificar como parte del mismo conjunto a todos los problemas que son *inherentemente simples* de resolver (o sea, sus soluciones ejecutan en un *tiempo aceptable*), y en otro conjunto a todos los que se suponen *inherentemente difíciles* (o sea, sus soluciones conocidas no ejecutan en tiempo razonable, incluso para un computador potente).

La idea anterior es intuitivamente el punto de partida de la Teoría de la Complejidad. Y ya puede verse el primer gran inconveniente: ¿*cuál es el límite entre los problemas inherentemente simples y los inherentemente difíciles (en términos de tiempo de ejecución)*? ¿Cuál medida o expresión del tiempo de ejecución se considera *aceptable* o *razonable*?

El hecho es que desde los primeros momentos del planteo de la teoría, se ha convenido en aceptar que el tiempo de ejecución de un algoritmo *es razonable* si puede expresarse como una *función polinómica de su tamaño de entrada*. Es decir, se asume que un *tiempo de ejecución polinómico* será razonablemente manejado por una computadora, y podrá esperarse que esa computadora llegue a un resultado en un tiempo prudente. Por lo tanto, el límite aceptado entre lo inherentemente simple y lo inherentemente difícil es el *tiempo de ejecución polinomial*.

La respuesta puede parecer arbitraria, pero en general tiene sentido aunque sea desde un punto de vista teórico: en principio, un tiempo polinomial es mejor o más aceptable que un tiempo exponencial o factorial, y asumir como razonable a un tiempo polinomial parece adecuado considerando las alternativas. Sin embargo, debe entenderse que ese es un límite teórico, que en la práctica podría verse desbordado convirtiéndolo en un subterfugio: si resultase que el algoritmo diseñado para un problema dado tiene un tiempo de ejecución $O(n^{60})$, ese tiempo sería polinomial (y quizás muy bienvenido por la teoría) pero tan inaceptable en términos prácticos como el tiempo exponencial [2].

Cuando se habla se *aceptable en términos prácticos*, se quiere decir que se espera que la computadora pueda llegar a una solución en un tiempo que sea útil para el usuario que pidió la solución, y que ese tiempo aumente en forma prudente a medida que aumenta la cantidad de datos. Un tiempo polinomial (si el grado del polinomio es bajo) puede implicar una demora de varios minutos o incluso algunas horas, pero eso es mucho más razonable que una demora de miles de años o de toda la vida del universo (que es a lo que nos enfrenta el tiempo exponencial incluso para tamaños pequeños de la entrada).

Aceptado el límite del tiempo polinomial, digamos entonces que un problema para el que sólo se conocen algoritmos de tiempo exponencial, factorial o combinación entre factorial y polinomial, se conoce como un *problema intratable*. Los *problemas intratables* llevan la discusión al límite práctico de la computabilidad: un *problema intratable* sólo podrá ser resuelto en tiempo prudente para pequeños valores del tamaño de su entrada (digamos $n <= 30$ para poner una cota) Pequeños crecimientos en el valor de *n* llevarán progresivamente el tiempo de ejecución a valores tan grandes que la computadora simplemente será inútil.

2.] Reducción de Problemas.

Antes de continuar con las clasificaciones típicas de la Teoría de la Complejidad, conviene introducir la noción de *reducción entre problemas*, ya que como veremos, juega un rol muy importante en esa teoría [1].

Dados dos problemas *t1* y *t2*, una *reducción R de t1 a t2* (que denotaremos como *t2 = R(t1)*) es un *proceso* por el cual las entradas del problema *t1* se convierten en entradas del problema *t2*, de forma tal que si luego se resuelve el problema *t2*, el mismo algoritmo resuelve también *t1* [2].

En esencia, lo que se logra con una reducción es que un algoritmo ya planteado para resolver cierto problema, pueda aplicarse sin cambios internos para resolver otro, a condición de ajustar las entradas. Un ejemplo muy simple servirá para aclarar la cuestión: Sea *t2* el problema de calcular el *mayor* valor entre tres números *a*, *b*, y *c*. Es evidente que la siguiente función hace el trabajo:

```
def mayor(a, b, c):
    if  a > b  and  a > c:
        may = a
    elif  b > c:
        may = b
    else:
        may = c
    return may
```

Sea ahora el problema *t1* de calcular el *menor* entre tres números *a*, *b* y *c*. Está claro que podríamos escribir una segunda función *menor()* que haga lo mismo que la función *mayor()*, pero cambiando el operador > (mayor que) por el operador < (menor que). Sin embargo, también podríamos reusar la función *mayor()*: si queremos el menor entre tres números *a*, *b* y *c*, sólo debemos cambiar el signo de esos tres valores (*-a*, *-b* y *–c*), enviar los valores opuestos como parámetro a la función *mayor()*, y cambiar el signo nuevamente al resultado devuelto por ella. El siguiente método *test()* aplica lo visto:

```
def test():
    a = int(input('A: '))
    b = int(input('B: '))
    c = int(input('C: '))

    # calcular el mayor y mostrar...
    may = mayor(a, b, c)
    print('El mayor es: ', may, end='\n')

    # aplicar la reducción, calcular el menor y mostrar...
    # m = mayor(-a, -b, -c)
    # men = -m

    # o bien...
    men = -(mayor(-a, -b, -c))
    print('El menor es: ', men, end='\n')
```

El proceso de cambio de signo de los tres valores originales, es una *reducción del problema t1 al problema t2*: se ajustaron los datos de *t1* para que puedan ser procesados por el algoritmo que resuelve a *t2*, y en este caso, se reajustó también el resultado para adecuarlo nuevamente a lo esperado para *t1*.

Está claro que aplicando la idea de *reducción* en este caso hemos obtenido una pequeña ganancia: si bien el tiempo de ejecución es constante y el mismo para ambos casos, y si bien además el espacio de memoria empleado es también el mismo, el hecho es que al menos en términos de *complejidad de código* hemos ahorrado trabajo, ya que en lugar de dos funciones tenemos y reusamos una sola.

Note entonces que para que una *reducción entre problemas* tenga sentido práctico, se esperaría obtener una ganancia en cuanto a eficiencia. Por lo pronto el propio proceso para llevar a cabo la reducción debería, en general, ejecutar en un tiempo menor al que llevaría la solución del problema hacia el cual se hace la reducción (en el caso del ejemplo el tiempo es constante tanto para la reducción como para la solución y no parece que se obtenga ventaja alguna, pero como vimos, se gana en simplicidad de código fuente).

Un contra-ejemplo es simple de exponer: suponga que se quiere obtener el mayor valor de una lista o de un arreglo desordenado. Eso puede hacerse con una función *mayor(lista)* que recorra la lista o arreglo con una sola pasada y se vaya quedando con el mayor valor de entre los que se van visitando, con lo cual se tiene un proceso de tiempo *O(n)*.

Si luego se quiere obtener el *menor* de la lista, podría pensarse en una reducción similar a la del problema anterior: recorrer la lista, cambiar el signo de todos los valores contenidos en ella, entregar la lista opuesta como parámetro a la función *mayor(lista)*, y cambiar el signo al resultado obtenido. Sin embargo, es fácil ver que en este caso no se obtiene ganancia alguna: Tanto la *reducción* como el *proceso* son de tiempo lineal. Si se cambian los signos en la misma lista original, luego deberemos volver a cambiarlos para seguir teniendo la lista original y eso duplica el tiempo de ejecución (aunque asintóticamente sigue siendo lineal). Y si se crea una segunda lista, entonces se duplica el espacio. Y aún cuando es cierto que se tiene una sola función en lugar de dos para calcular el óptimo, el hecho es que ahora debe sumarse la función que hace la reducción (el cambio de signos en los valores contenidos en la lista). Es claro entonces que la reducción propuesta para este problema no es aceptable desde el punto de vista práctico.

Por lo anterior, resulta particularmente apreciado que una *reducción* pueda entonces aplicarse en tiempo razonable de ejecución y obtener una ventaja en el tiempo de ejecución completo. En ese sentido, una *reducción polinómica R_p es una reducción que ejecuta ella misma en tiempo polinomial*. En los dos ejemplos simples que hemos analizado, las reducciones eran *O(1)* y *O(n)*, que en última instancia son ambas polinómicas. Y en ambos problemas, la solución ejecutaba también en tiempo polinómico [2].

El caso es que las *reducciones polinómicas* podrían ser muy valiosas en casos de enfrentar *problemas intratables*: si se tiene un problema *t1* que *se sabe que hasta ese momento es intratable*, y se encuentra una manera de *reducirlo polinómicamente* a otro problema *t2* para el cual se tiene una solución de tiempo razonable (por caso, polinómica ella también), entonces inmediatamente el problema *t1* dejaría de ser intratable, ya que la solución polinómica para *t2* implicaría también una solución polinómica/razonable para *t1*, considerando que a su vez la reducción *t2 = R_p(t1)* es polinómica y el proceso completo también lo será. Y como veremos, esta implicación no es el único gran resultado que podría obtenerse…

3.] Noción elemental de *grafo* y algunos problemas clásicos.

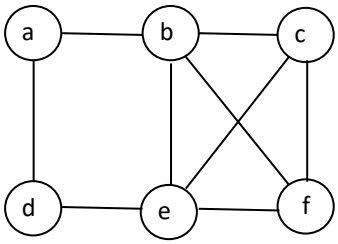
Para ilustrar la expectativa que existe en cuanto a lo dicho en la sección anterior, analizaremos conceptualmente algunos problemas del campo de los *grafos*. En general, un *grafo* G = {V, A} es una estructura de datos que consta de un conjunto *V* de elementos llamados *vértices* y de otro conjunto *A* de elementos llamados *arcos*, de tal forma que cada arco representa una relación o correspondencia entre dos vértices [3] [4] [1].

Cada vértice a su vez, representa objetos o elementos del enunciado o dominio del problema, y las relaciones entre ellos surgen de ese mismo dominio. Un ejemplo de un grafo puede verse en la *Figura 2* más abajo: en el grafo *G1* de la izquierda, cada vértice está identificado con letras desde la "a" hasta la "f" (y podrían representar ciudades, personas, asignaturas, aeropuertos, etc.) y cada arco está representado por líneas finas que unen dos vértices (dependiendo del problema, cada arco podría representar una ruta existente entre dos ciudades, una relación de amistad entre dos personas, un contacto aceptado en una red social entre dos usuarios, etc.) Formalmente, el arco que une dos vértices cualesquiera *x* e *y* se denota como *(x, y)*.

En general, un grafo puede implementarse de diversas formas en cualquier lenguaje de programación, pero una manera común (y muy clásica) consiste en usar una *matriz booleana m* (o también una matriz de valores *0* - *1*) para representar al conjunto de arcos [3] [1]: en esa matriz (llamada *matriz de adyacencias* del grafo) cada *fila* se usa para representar un *vértice de partida de un arco*, y cada columna para representar un *vértice de llegada*.

Los vértices se hacen coincidir con los números de fila y columna en forma progresiva: así, el vértice *a* sería el vértice *0* y se usarían la fila y la columna *0* para representar a sus arcos, y así sucesivamente. Un *true* en el casillero *m[i][j]* indicaría que el arco *(i, j)* que parte del vértice *i* y llega al vértice *j* efectivamente existe, mientras que un *false* implica lo contrario. En la siguiente figura, se muestra un grafo *G1* y su matriz de adyacencia (para simplificar, se asume que se usarán *0* y *1* en lugar de *true* y *false*, y que los valores *0* están implícitos en los casilleros vacíos):

Figura 1: Un grafo G1 representado por su matriz de adyacencias.



Grafo *G1* original

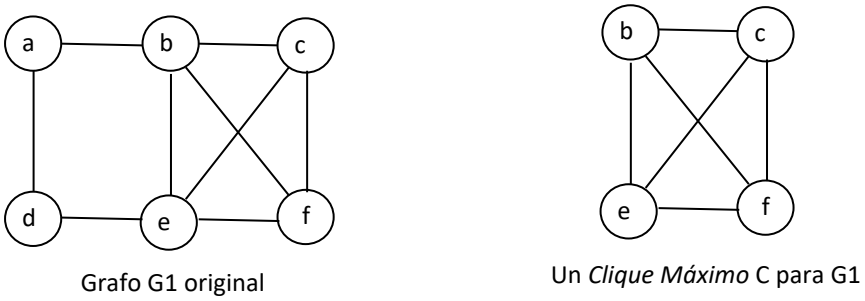
	a	b	c	d	e	f
a		1		1		
b	1		1		1	1
c		1			1	1
d	1				1	
e		1	1	1		1
f		1	1		1	

Representación del grafo *G1* mediante su *matriz de adyacencias*

Tomemos ahora un ejemplo de dos problemas muy conocidos del campo de los grafos: Uno de ellos se conoce como el *Problema del Clique Máximo* [2]: se tiene un grafo *G1* y se quiere saber cuál es el máximo subconjunto *C* de vértices de *G1* para el cual todos los vértices están conectados uno a uno con todos los demás[1] en *C*. Gráficamente, la figura de la izquierda representa el grafo *G1* original, y en la de la derecha mostramos un subgrafo *C* que constituye un *clique máximo* para *G1*:

[1] La palabra *clique* en inglés puede entenderse como *pandilla* o también como *cuadrilla* o *banda* o *tribu*, y el problema del *clique máximo* sería entonces el problema de encontrar la *máxima pandilla* de vértices que están todos unidos entre sí dentro del grafo original.

Figura 2: Un grafo y un clique máximo para ese grafo.



Grafo G1 original

Un *Clique Máximo* C para G1

El *clique* de la derecha es máximo: contiene cuatro vértices conectados uno con cada uno de los otros tres en forma directa, y es el máximo subconjunto de vértices de *G1* que cumple esa condición. Y bien: los *únicos algoritmos* que se conocen para calcular un *clique máximo* en un grafo de *n* vértices, son de tiempo exponencial [$O(2^n)$] y por lo tanto (por lo que hasta ahora se sabe) es un problema *intratable*.

Los algoritmos conocidos para resolver el problema del clique máximo son algoritmos de *fuerza bruta*, en los que se toman *todas las combinaciones posibles* entre los *n* vértices, luego una por una se determina cuántos de esos vértices están conectados uno a uno en cada combinación, y se va recordando la de mayor cantidad de vértices.

En términos muy generales, un esquema de pseudocódigo para resolver por *fuerza bruta* el problema de determinar un clique máximo en un grafo *g* de *n* vértices (y retornar la cantidad de vértices del mismo) podría ser el siguiente:

Figura 3: Pseudocódigo básico del algoritmo de fuerza bruta para el problema del Clique Máximo,

```
def max_clique(g):
   1.) Sea n la cantidad de vertices del grafo g.
   2.) Sea mc = 0 la mayor cantidad de vértices vista hasta aquí en un clique de g.
   3.) Para cada posible combinación c de los n vértices de g:
       3.1.) Si c es un clique válido para g:
             3.1.1.) Sea t la cantidad de vértices de c.
             3.1.2.) Si t > mc:
                     3.1.2.1.) mc = t
   4.) Retornar mc.
```

A modo de ejemplo, si el grafo *g* tiene tan solo *n = 3* vértices (sea por ejemplo *V* = {a, b, c} el conjunto de vértices de *g*) entonces el conjunto *S* de todas las combinaciones posibles de esos *n = 3* vértices tiene $2^n = 2^3 = 8$ elementos:

```
S = {{}, {a}, {b}, {c}, {a, b}, {a, c}, {b, c}, {a, b, c}}
```

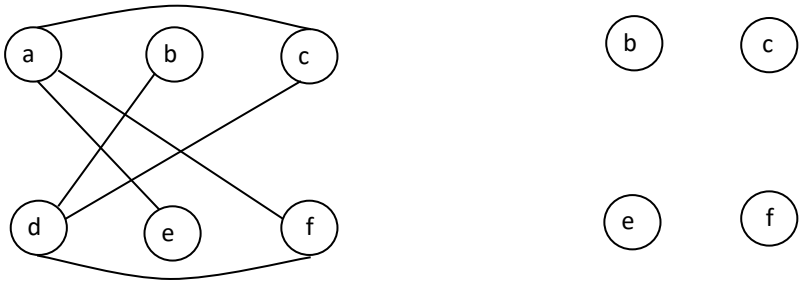
Por lo tanto, en el algoritmo para el clique máximo que mostramos más arriba, el ciclo indicado en el paso 3 deberá hacer 2^n iteraciones (una por cada una de las posibles combinaciones del conjunto *S*) y comprobar cada una de ellas por separado para determinar si efectivamente conforma un clique en el grafo *g*.

Sólo para el ciclo del paso 3 el tiempo de ejecución es entonces $O(2^n)$ (lo cual ya es muy malo…) y todavía queda determinar el tiempo de ejecución de la consulta efectuada en el paso 3.1 en cada vuelta del ciclo para saber si la combinación tomada es o no un clique. Si el grafo está implementado en forma matricial, esto último básicamente requiere recorrer en forma completa la matriz de adyacencias del grafo, que tiene *n* filas y *n* columnas, llevando un tiempo $O(n^2)$ cada comprobación… con lo que el tiempo total de ejecución es un

escalofriante $O(n^2 * 2^n)$. Y en esencia, este es el único algoritmo que se conoce para el problema del clique máximo (salvo por variaciones que se aplican en casos particulares para intentar acelerar el tiempo de respuesta en esos casos).

Supongamos ahora un segundo problema muy conocido, también del mundo de los grafos: el *Problema del Subconjunto Independiente Máximo* (que abreviaremos como *SIM*) [2]: como antes, se tiene un grafo *G2* y se quiere saber ahora cuál es el máximo subconjunto *C* de vértices de *G2* para el cual *ninguno* de sus vértices está conectado con los demás en *C*. En la *Figura 4*, la gráfica de la izquierda representa un grafo *G2* original, y en la gráfica de la derecha mostramos un subgrafo *C* que constituye un *SIM* para *G2*:

Figura 4: Un grafo y un subconjunto independiente máximo para ese grafo.



Grafo G2 original — Un *SIM* C para G2

El subconjunto de vértices *C* de la derecha constituye un *SIM* para *G2*, ya que es el mayor subconjunto de vértices de *G2* tal que ninguno de esos vértices está enlazado directamente con los demás.

Lo primero que debemos observar en estos dos ejemplos, es que ambos problemas están muy relacionados entre sí. Para rescatar esa relación, note que (con toda intención…) se eligió el grafo *G2* de forma que sea el *grafo complemento o inverso* de *G1 (*se denota: *G2 = G1')*: todo arco que está en *G1* no está en *G2*, y todo arco que falta en *G1* está en *G2*. Y ahora veamos que si se tiene el grafo *G1* y se calcula un *clique máximo* para él, entonces ese mismo conjunto de vértices también constituye un *SIM* pero para el *grafo inverso* de *G1* (que en nuestro caso es *G2*…) La *Figura 5* muestra las matrices de adyacencia de ambos grafos.

Figura 5: Representación matricial de un grafo *G1* y de su grafo inverso *G2 = G1'*.

	a	b	c	d	e	f
a		1		1		
b	1		1		1	1
c		1			1	1
d	1				1	
e		1	1	1		1
f		1	1		1	

Grafo G1 original

	a	b	c	d	e	f
a			1		1	1
b				1		
c	1			1		
d		1	1			1
e	1					
f	1			1		

Grafo G2 = G1'

Es intuitivamente simple de demostrar la relación *(G1, Clique) = (G1', SIM) = (G2, SIM)*: como todo arco *(x, y)* que está en *G1* no está en *G1'*, entonces cualquier subconjunto de vértices de *G1* que forme un *clique* (todos unidos con todos) necesariamente tendrá todos esos

vértices desconectados en *G1'*, formando un *subconjunto independiente* en *G1'*. Y si el *clique* en *G1* es máximo, entonces también es máximo el subconjunto independiente en *G1'*.

Por lo tanto, el *Problema del Clique Máximo* puede *reducirse* al *Problema del SIM* (¡y viceversa!) simplemente invirtiendo los valores de todos los casilleros de la matriz de adyacencia del grafo de entrada (y obviamente manteniendo la diagonal principal con valores *0* o *false*). Ese proceso requiere dos ciclos anidados de *n* iteraciones cada uno, por lo cual el tiempo empleado para hacer la *reducción* es $O(n^2)$ quedando así, una *reducción polinómica*.

Ahora bien: hemos visto que el problema del *Clique Máximo* posiblemente sea *intratable*. Pero por lo dicho en el párrafo anterior, si se tuviese una solución de tiempo polinómico/razonable para el problema del *SIM*, entonces esa solución serviría también para el *Clique Máximo*… que dejaría de ser intratable. La mala nueva (por lo que se sabe hasta ahora) es que el problema del *SIM también es intratable*: los únicos algoritmos que se conocen son variaciones simples del mismo algoritmo de *Fuerza Bruta* de $O(2^n)$ que permite calcular el *Clique Máximo*.

4.] Máquinas Deterministas y Máquinas No Deterministas.

Independientemente de la existencia real y concreta de computadoras modernas y potentes en la actualidad, las Ciencias de la Computación han avanzado a lo largo de las décadas también en base al planteo de *máquinas teóricas*, planteadas de manera formal para poder ser aplicadas al estudio de distintos escenarios y tipos de problemas [1] [2].

Las computadoras existentes hoy en día están todas basadas en un modelo teórico designado como *Máquina de Turing*, el cual define a su vez un tipo de máquina que se conoce como *Máquina Determinista*. Básicamente, una *máquina determinista* es aquella en la cual dado su estado actual, se puede determinar con certeza el siguiente estado en el cual estará en el próximo paso. Todas las computadoras existentes hasta hoy, son máquinas deterministas.

En contrapartida, existe un modelo teórico de máquina designada como *Máquina No Determinista*, que es aquella en la que no puede predecirse el estado en el que estará en el siguiente paso sólo conociendo el estado actual. Debe quedar claro que el *no determinismo* no implica *funcionamiento aleatorio*: esencialmente, una máquina *no determinista* resuelve correctamente un problema (de la misma forma que una máquina determinista), pero a diferencia de una máquina determinista, las no deterministas *ven* (o *adivinan*) la respuesta correcta y eligen en *tiempo polinómico* el mejor camino entre varios posibles (si se presentan), permitiendo que puedan resolverse en tiempo de ejecución polinomial algunos problemas para los que una máquina determinista emplearía un tiempo exponencial.

Como dijimos, las máquinas no deterministas son *máquinas teóricas* (o *abstractas*): está definido en términos formales/matemáticos el comportamiento de tales máquinas y se sabe en términos teóricos lo que pueden o no pueden hacer, pero no existen aparatos concretos y funcionando que sean no deterministas: ninguna computadora existente es no determinista, y por lo tanto ninguna tiene el sorprendente poder del no determinismo.

Muy básicamente, una máquina no determinista podría ejecutar el mismo juego de instrucciones y funciones que una determinista (podría usar los mismos lenguajes de programación con las mismas instrucciones y funciones que una determinista). Los

problemas para los que ya existen buenos algoritmos en una máquina determinista, también se resolverían en forma eficiente en una no determinista: *todo lo que una máquina determinista ya hace bien, también será bien resuelto por una no determinista*.

Pero las máquinas no deterministas tendrían una ventaja casi increíble sobre las deterministas[2]: dentro de su juego de instrucciones, las no deterministas incluirían una instrucción especial que podría designarse como *if-better* (que podría entenderse como: *si corresponde*, o bien como: *si es lo mejor*), la cual funcionaría en forma similar a una instrucción condicional normal, pero con las siguientes propiedades generales [2]:

i. En forma similar a una instrucción condicional normal, la instrucción *if-better* dispondría de dos ramas o caminos a seguir: el *mejor* u *óptimo* y el *no óptimo*:

```
if-better:
    # procesos que están en el camino óptimo a ejecutar.
else:
    # procesos que no están el camino óptimo a ejecutar.
```

ii. Una instrucción condicional normal chequea una proposición lógica por *True* o *False*, y hace ese chequeo en *tiempo constante* (*O(1)*). La instrucción *if_better* no chequearía ninguna proposición, pero lo que sea que tuviese que hacer para determinar el próximo estado al que debiera pasar el programa, lo haría en *tiempo polinómico* ($O(n^k)$).

iii. La instrucción *if-better* sería capaz de analizar el estado del programa hasta ese momento, y *decidir por sí misma* (sin ningún dato adicional ni proposición lógica alguna a controlar ni proceso adicional agregado por el programador) qué es lo que corresponde hacer.

Es difícil formarse una idea de lo potente que resultaría una instrucción tan especial como esta en una computadora, si pudiese construirse alguna con esa capacidad. Si un programa tuviese que analizar múltiples caminos a seguir, de forma que cada uno de ellos a su vez se abriese en otros múltiples subcaminos y recorridos, una máquina determinista normal debería explorar uno por uno todos esos caminos hasta descubrir cuál es el óptimo y decidir quedarse con él, lo cual llevaría un *tiempo exponencial* (como vimos en el pseudocódigo del problema del clique máximo).

Pero una máquina no determinista, dotada de la fantástica instrucción *if-better*, simplemente *sabría* qué camino a seguir es el óptimo (o qué combinación es la mejor) sin más datos que la presencia del inicio de ese camino frente a ella. Sin tener que recorrer ese camino o analizar esa combinación, la máquina no determinista escogería el camino óptimo, y demoraría un *tiempo polinomial* (es decir, un tiempo aceptable) en hacerlo, en lugar del tiempo exponencial que demandaría la máquina determinista en analizar todos los caminos, uno por uno. A modo de ejemplo, el siguiente sería el pseudocódigo de un algoritmo para calcular un clique máximo en un grafo *g* de *n* vértices y retornar su cantidad de vértices, pero para una máquina no determinista dotada de la instrucción *if-better* [2]:

[2] Las ideas y principios que siguen en cuanto al funcionamiento conceptual de una máquina no determinista, están inspirados en los materiales del curso *CS313 - Intro to Theoretical Computer Science* (a cargo del profesor *Ph.D. Sebastian Wernicke*) disponible en la plataforma *Udacity* (https://www.udacity.com/course/intro-to-theoretical-computer-science--cs313)

Figura 6: Pseudocódigo básico del algoritmo *no determinista* para el problema del Clique Máximo.

```
def max_clique_nondeterministic(g):
    1.) Sea n la cantidad de vertices del grafo g.
    2.) Sea c = [] la lista de vértices que contendrá al clique máximo.
    3.) Para cada vértice v de los n vértices de g:
        3.1.) if-better [o sea: si v está en un clique máximo de g]:
            3.1.1.) Añadir v a c.
    4.) Retornar el tamaño de c.
```

Este algoritmo no necesita analizar cada una de las 2^n posibles combinaciones de vértices (como hacía el algoritmo de fuerza bruta determinista de la *Figura 3*, *página 598*): sólo recorre el conjunto de *n* vértices del grafo, y con cada vértice aplica la fantástica instrucción *if-better* para saber si ese vértice sirve o no. Como el ciclo hace *n* iteraciones (hasta aquí *O(n)*), y en cada iteración *if-better* hace su trabajo en tiempo polinómico ($O(n^k)$ para algún valor $k > 0$)), el resultado combinado también es polinómico y de la forma: $O(n) * O(n^k) = O(n^{k+1})$.

Así de simple. Los únicos algoritmos conocidos para el problema del clique máximo y para muchos otros en una máquina determinista son de tiempo exponencial $O(2^n)$, pero en una hipotética máquina no determinista se podrían plantear algoritmos que resuelvan esos mismos problemas en tiempo polinomial $O(n^k)$.

La *teoría de la complejidad* trata entonces de estudiar los distintos tipos de problemas que pueden surgir y clasificar a esos problemas según su *complejidad inherente*, lo que en última instancia *equivale a clasificarlos de acuerdo al tipo de máquina (teórica o concreta) que permitiría resolverlos dentro de ciertos rangos de tiempo de ejecución y/o consumo de memoria* [1].

Algunos problemas podrán ser resueltos en tiempo aceptable (o sea, polinomial) por máquinas deterministas (y diremos que esos problemas son *inherentemente fáciles* de resolver). Y otros podrán ser resueltos *también en tiempo polinomial y aceptable* pero por increíbles máquinas no deterministas (y diremos que esos problemas son *inherentemente difíciles*).

Y si bien podría parecer que eso es todo, tenga cuidado y no caiga en la trampa: muchos problemas *no podrán ser resueltos en tiempo aceptable ni siquiera con nuestras mágicas máquinas no deterministas*: algunos problemas requieren tiempo exponencial se use la máquina que se use. Por ejemplo, suponga que se nos pide simplemente listar (mostrar por pantalla) todos y cada uno de los subconjuntos de un conjunto *C* de *n* componentes. Como vimos, la cantidad de subconjuntos de un conjunto de *n* elementos es igual a 2^n, y si se nos está pidiendo mostrarlos a todos, ese algoritmo forzosamente requerirá un ciclo de 2^n iteraciones y ejecutará en tiempo $O(2^n)$ nos guste o no, tanto en máquinas deterministas como en máquinas no deterministas. El no determinismo nos permite elegir en forma rápida un camino óptimo entre muchos posibles, pero no puede evitar que se recorran cada uno de los caminos posibles si justamente ese es el problema: mirarlos y mostrarlos a todos, uno por uno.

A modo de curiosidad (y sólo como ejercicio a nivel intelectual), mostramos el algoritmo no determinista de la *Figura 6*, pero ahora escrito en un hipotético "Python No-Determinista 1.0" (por favor recuerde: ***no existe*** la instrucción *if-better* ni tampoco un lenguaje *Python No-Determinista 1.0*):

```
# hipotético programa en Python usando if-better...
def max_clique_nondeterministic(g):
    n = len(g)
    c = []
    for v in range(n):
        if-better:
            c.add(v)
    return len(c)
```

5.] Clases de Complejidad.

Como se dijo, la *teoría de la complejidad* se centra en estudiar y clasificar problemas de acuerdo a su complejidad interna inherente. Como resultado, se han planteado diversos conjuntos de problemas, agrupados según lo que se sabe de ellos en cuanto a consumo de recursos y posibilidades de ser resueltos con diversos tipos de máquinas (teóricas o concretas).

Esos conjuntos de problemas similares se llaman en general "*clases de complejidad*" [1] [2]. Al menos una de esas clases es muy importante en todo estudio introductivo de la teoría de la complejidad (y puede decirse que todo el andamiaje de la teoría comienza con ella…) y se designa como *clase de complejidad P* (o simplemente, *clase P*), que se define en forma esencial como sigue:

- *Clase P*: Conjunto de todos los *problemas de decisión* que pueden ser resueltos con *algoritmos de tiempo polinomial* usando una *máquina determinista*.

Si bien la definición hace referencia específicamente a *problemas de decisión* (problemas en los que se espera como única salida un valor lógico *true* o *false*), no hay mayores inconvenientes en extrapolar la discusión que sigue a problemas de otros tipos. Los fundamentos esenciales de la teoría de la complejidad están basados en *problemas de decisión*, pero las conclusiones generales de la teoría son aplicables a otros tipos de problemas contenidos en otras clases de complejidad (como la *clase FP* que es similar a *P*, pero sin la restricción de que los problemas sean de decisión). Por otra parte, cuando se habla de tiempo polinomial, se da por entendido que se hace referencia a que el tiempo de ejecución del algoritmo es una función polinómica del tamaño de la entrada para ese algoritmo.

La *clase P* (cuyo nombre es abreviatura de *Polynomial-Time*) es el conjunto de los problemas para los cuales se conocen soluciones consideradas eficientes, o al menos aceptables, desde el punto de vista del tiempo de ejecución en una máquina determinista (insistimos: no se preocupe por el detalle de si los problemas son o no de decisión). Puede decirse entonces que *P* es la clase en la que todo programador desearía que esté cualquier problema, ya que un problema en *P* tiene al menos una solución conocida y "rápida" en cualquier computadora actual. Por caso, pertenecen a *P* problemas como el *ordenamiento* de un arreglo (sabemos que existen algoritmos de ordenamiento que ejecutan en *tiempo polinomial*, y no importa demasiado en este momento si existen algoritmos "mejores" o "peores"), o la *búsqueda* de un elemento en un arreglo o una lista. La *clase P* entonces, agrupa a los problemas que hemos identificado como *inherentemente fáciles* de resolver.

Sin embargo, y como también vimos en la sección anterior, existen muchos otros problemas para los cuales simplemente no se conocen soluciones que puedan correr en un tiempo de ejecución razonable en ninguna computadora como las que hoy disponemos (todas

máquinas deterministas). En otras palabras: para muchos problemas ***existen*** algoritmos conocidos, pero esos algoritmos (normalmente basados en estrategia de *fuerza bruta*) tienen tiempos de *ejecución exponencial* y por lo tanto resultan tremendamente ineficientes incluso en una computadora moderna, y aún para tamaños relativamente pequeños del conjunto de datos a procesar. Algunos problemas muy estudiados que tienen estas características (y sólo por citar unos pocos) son el *Problema del Viajante* (dadas *n* ciudades y el conjunto de caminos que permiten vincularlas, calcular el recorrido más corto que permita salir de una de ellas y visitar a todas las otras, sin repetir ninguna), el *Problema de Clique Máximo* y el *Problema del Máximo Subconjunto Independiente* (que vimos en esta misma ficha).

La *teoría de la complejidad* también debe lidiar con estos problemas, y también intenta clasificarlos de acuerdo a su complejidad inherente. Si bien podría pensarse en que todo problema que no esté en *P* podría entrar simplemente en otra clase "No-P", el hecho es que esa sobre-simplificación nos diría muy poco de las propiedades de los problemas analizados: por ejemplo, el hecho de que un problema *x* no pertenezca a *P* nos dice que no se conocen soluciones de tiempo polinomial para *x* en una computadora normal actual, pero no se dice nada respecto de posibles buenas soluciones para *x* en *otro* tipo de computadoras (existan estas o no…)

Los ya citados problemas del *viajante*, del *clique máximo* y del *máximo subconjunto independiente* no pertenecen a *P* (o al menos eso es lo que hasta ahora se supone), pero es interesante recordar que para los tres existen algoritmos que *ejecutarían* en tiempo polinomial en una máquina no determinista... y no es relevante para la teoría de la complejidad si las máquinas no deterministas existen o no en la práctica. La teoría hace suposiciones y plantea hipótesis que son válidas en distintos escenarios basados en máquinas concretas (como las deterministas) o en máquinas teóricas (como las no deterministas). Y todavía más: como muchos problemas no tendrían una solución eficiente *ni siquiera* en una máquina tan poderosa como una no determinista, este tipo de problemas merecería ser clasificado incluso en una clase de complejidad *diferente* a la que se use para el *viajante* o el *clique*.

Por lo anterior, y en forma más ajustada, la teoría de la complejidad plantea entonces una segunda clase, conocida como *clase de complejidad NP* (o simplemente, *clase NP*), cuya definición general es la que sigue:

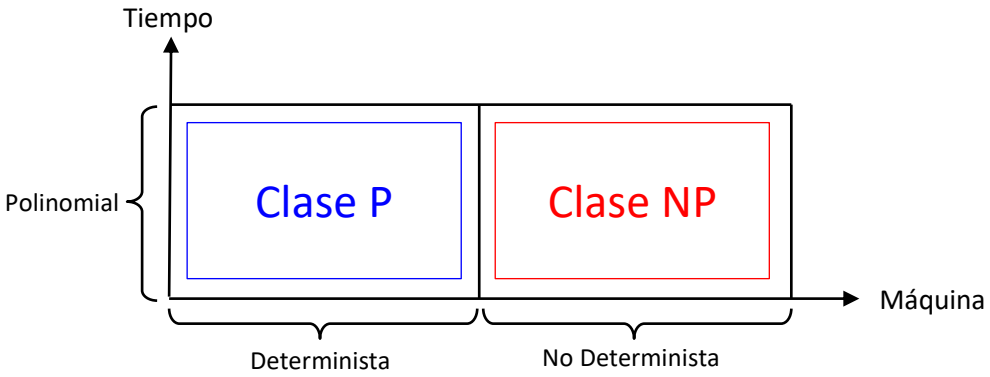
- *Clase NP*: Conjunto de todos los *problemas de decisión* que pueden ser resueltos con *algoritmos de tiempo polinomial* usando una *máquina no determinista*.

Otra vez, se habla de problemas de decisión pero no habrá inconveniente con ello en la explicación general que sigue, incluso considerando problemas de otros tipos.

La *clase NP* (cuyo nombre proviene de las iniciales de *Nondeterministic Polynomial-Time*) abarca problemas de decisión que en una máquina determinista requerirían tiempo exponencial, pero que en una máquina *no determinista* insumirían tiempo polinomial. Entre muchos otros, los ya citados problemas del *viajante*, del *clique máximo* y del *subconjunto independiente máximo* pertenecen a *NP*: los únicos algoritmos conocidos para los tres son de tiempo exponencial en una computadora moderna común (determinista), pero podrían resolverse con algoritmos de tiempo polinomial en una máquina no determinista (si existiese alguna).

Hasta aquí hemos presentado dos clases de complejidad que son fundamentales en la teoría. De alguna manera, esas dos clases marcan una primera división entre lo que se sabe en relación a muchos problemas: ambas abarcan problemas que pueden resolverse en tiempo polinómico pero con distintos tipos de máquinas [2]:

Figura 7: Las clases P y NP.

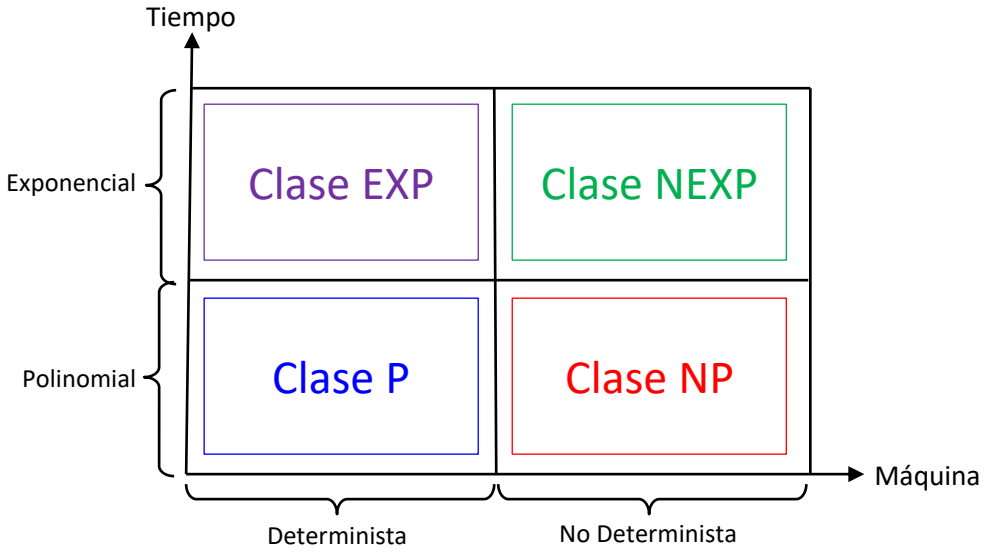


Como vemos, la clasificación de problemas se hace en base al *tipo de máquina* que podría resolverlos con determinada precisión y eficiencia en cuanto al tiempo y/o al espacio requerido. Y como hemos visto que existen problemas que no estarían contenidos ni en *P* ni en *NP*, es de suponer que existen otras clases de complejidad para clasificar a estos problemas extremos. Inmediatamente surgen otras dos que podemos agregar al panorama general (aunque luego no nos ocuparemos demasiado de ellas: simplemente las nombramos para completar la visión de conjunto) [2]:

- *Clase EXP*: Conjunto de todos los *problemas de decisión* que pueden ser resueltos con *algoritmos de tiempo* ***exponencial*** usando una *máquina determinista*. El nombre proviene de *Exponencial-Time*.
- *Clase NEXP*: Conjunto de todos los *problemas de decisión* que pueden ser resueltos con *algoritmos de tiempo* ***exponencial*** usando una *máquina no determinista*. El nombre proviene de *Nondeterministic Exponential-Time*.

El gráfico podría entonces expandirse de la siguiente forma:

Figura 8: Las clases P, NP, EXP y NEXP.

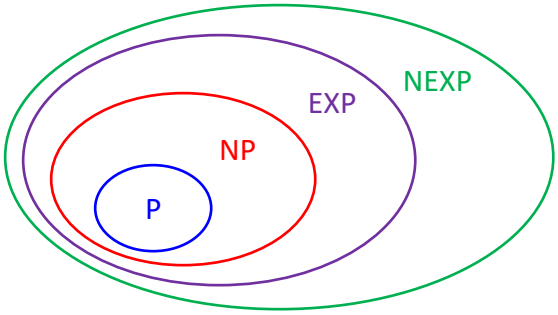


Además, nada impide que un mismo problema pueda ser clasificado en dos o más clases diferentes de complejidad: de hecho, es fácil ver que todos los problemas que están en *P*, también están en *NP*: si una máquina determinista puede resolver un problema en tiempo polinomial, es completamente obvio que una máquina no determinista también puede hacerlo. Por lo tanto, *P* es un subconjunto de *NP* (o sea: $P \subset NP$). Razonamientos similares (que dejamos para ser verificados por el estudiante) llevan a concluir que:

$$P \subset NP \subset EXP \subset NEXP$$

Con lo que queda la siguiente gráfica de subconjuntos [2]:

Figura 9: Relación de subconjuntos entre las clases P, NP, EXP y NEXP.



6.] La cuestión *P vs NP*. Problemas *NP-Complete*.

En la sección anterior hemos visto que la clase *P* es subconjunto de *NP*: todo problema en *P* pertenece también a *NP*. Pero la clase *NP* contiene muchos otros problemas que son los que esencialmente definen la razón de ser de *NP*: aquellos que tienen *solución de tiempo exponencial* en *P* pero de *tiempo polinomial* en *NP* (como el *clique máximo*, el *viajante* o el *SIM*).

Ahora bien: los problemas que están clasificados directamente en *NP* y no en *P* se caracterizan por el hecho de que *no se conocen* algoritmos eficientes, de tiempo polinomial, para resolverlos en una computadora normal. Pero si para algún problema en *NP* se encontrase una solución eficiente, automáticamente ese problema pasaría a pertenecer a *P*.

El hecho de estar en *NP* y no en *P* es una cuestión que depende de conocer o no un algoritmo adecuado que ponga a un problema dado en *P*. Por lo tanto, *si se encontrasen* soluciones de tiempo polinomial para ***todos*** los problemas de *NP*, tendríamos que todos los problemas de *NP* estarían también en *P* (o sea, $NP \subset P$). Y como ya sabemos que $P \subset NP$, inmediatamente esto implicaría que *P* y *NP* serían en realidad la misma clase: $P = NP$.

La realidad es que ***nadie sabe*** si la relación ***P = NP es cierta o falsa***. Nadie ha podido demostrar ni lo uno ni lo otro: se desconoce si todo problema de *NP* podría resolverse con algoritmos de tiempo polinomial y hacer entonces que la proposición sea cierta ($P = NP$), y nadie ha sido tampoco capaz de mostrar un problema específico de *NP* para el cual se pruebe que no está a su vez en *P*, con lo que ese problema serviría de contra-ejemplo y probaría que la relación es falsa ($P \neq NP$).

El problema de demostrar si $P = NP$ es cierta o falsa se designa como el ***Problema P vs NP*** (se lee "*P versus NP*") y constituye uno de los problemas abiertos más famosos del campo de las matemáticas y las ciencias de la computación. El problema es tan complejo y sus

implicancias (tanto si se demuestra por cierto o por falso) son tan amplias que incluso está clasificado como uno de los *Siete Problemas del Milenio*[3] por el *Clay Mathematics Institute*, de los Estados Unidos, que en el año 2000 propuso una lista de siete problemas considerados fundamentales en el campo de las matemáticas, y ofreció un premio de u$s1000000 (un millón de dólares) por cada uno de estos problemas para quien pudiera resolver cualquiera de ellos[4]. Al momento de confeccionarse esta ficha de estudios (año 2015), el *Problema P vs NP* permanecía sin resolverse, pero todo indica que la mayor parte de los científicos y especialistas del área suponen que la proposición *es falsa* (o sea, suponen que $P \neq NP$).

Si pudiera demostrarse que $P = NP$, la implicancia inmediata sería que cualquier máquina determinista es al menos tan poderosa como una no determinista, ya que ambas podrían resolver en forma eficiente los mismos problemas, y sólo sería necesario encontrar el algoritmo correcto (que podría seguir "oculto", pero al menos se tendría la certeza en cuanto a su existencia). Por otra parte, también implicaría que muchas de las clases de la teoría son equivalentes entre sí, con lo que la propia teoría se simplificaría de forma notable.

Si en cambio se demostrase lo contrario, es decir, que $P \neq NP$, entonces definitivamente se confirmaría que el no determinismo es realmente una herramienta extremadamente poderosa. Para muchos problemas hoy considerados intratables se sabría que no existe una solución eficiente y se podría simplemente dejar de buscarla, intentando en cambio resolver esos problemas en base a soluciones aproximadas o sub-óptimas (cosa que, de paso, hoy ya se está haciendo).

Y bien: el *Problema P vs NP* está abierto y a la espera de ser resuelto, por la afirmativa o la negativa. En ese sentido, entonces, ¿qué es lo que se puede hacer para intentar probar la veracidad o la falsedad de la proposición $P = NP$? Obviamente, si se pudiese probar que un problema *x* en particular que pertenezca a *NP* no puede resolverse con *ningún algoritmo determinista en tiempo polinomial* se habrá probado que la relación es falsa ($P \neq NP$), pues es suficiente con encontrar ***un problema*** para el que pueda probarse que *definitivamente* esté fuera de *P* pero que a su vez esté en *NP*.

Demostrar que la relación es cierta ($P = NP$) requeriría en principio bastante más trabajo: habría que demostrar que ***todos los problemas*** de *NP* pueden resolverse en tiempo polinomial mediante algoritmos deterministas… En la práctica esto parece simplemente imposible de hacer: hay muchos problemas en *NP*, todos muy diferentes entre ellos, con diferentes tipos de entradas y diferentes objetivos y salidas, e incluso podría ponerse peor: nada garantiza que no sigan apareciendo *nuevos* problemas en *NP*.

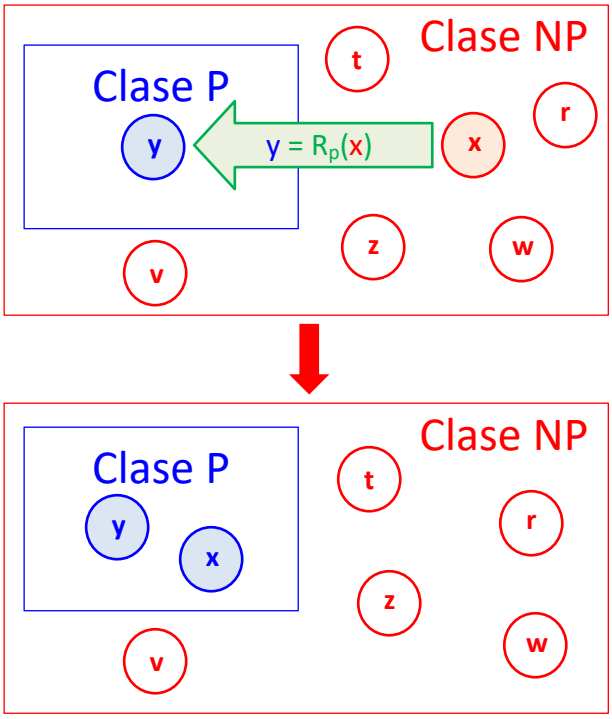
Sin embargo, las cosas no son en realidad tan oscuras. El concepto de *reducción polinómica* (que hemos introducido en esta misma Ficha, *página 594* y siguientes) permite intentar vías de demostración que simplifican bastante la cuestión [1] [2]: supongamos que se tiene un problema *x* que se sabe que pertenece a *NP* (es decir, se sabe que *x* puede resolverse en tiempo polinómico con una máquina no determinista) pero no se sabe si *x* pertenece a *P* o no. Supongamos además que tenemos otro problema *y* que se sabe que pertenece a *P* (o sea, se sabe que puede resolverse en tiempo polinómico con una máquina determinista). Y

[3] Fuente: https://en.wikipedia.org/wiki/Millennium_Prize_Problems.

[4] Fuente: http://www.claymath.org/.

supongamos finalmente que se logra encontrar una *reducción polinómica* R_p que transforme las entradas de *x* en entradas de *y*. Gráficamente:

Figura 10: Esquema y efecto de una reducción polinómica entre problemas de NP y P.



La reducción $y = R_p(x)$ implicaría que cualquier solución de *y* sería también una solución de *x*, por lo que si *y* está en *P* (puede resolverse en tiempo polinómico en una máquina determinista) entonces *x* también está en *P* por efecto de la reducción. La reducción $y = R_p(x)$ aumentó el tamaño de la clase *P*, pues agregó a ella un nuevo problema.

Es obvio que no podemos aplicar este procedimiento con ***todos*** los demás problemas de *NP*, uno por uno, por lo que este camino no serviría para intentar demostrar que *P = NP*. Pero hagamos ahora una suposición mucho más radical: *supongamos* que existiese en *NP* un problema ***q*** tan especial, que se pudiese demostrar de manera formal que ***todos*** los demás problemas de *NP* pueden ser reducidos a ***q*** con reducciones polinómicas.

Si ese fuese el caso, entonces ese sorprendente problema ***q*** podría ser la clave para probar que *P = NP*: bastaría con encontrar la manera de reducir ***q*** a uno cualquiera ***y*** de los problemas de *P*, ya que entonces al resolver ***y*** se resolvería ***q***, y al resolver ***q*** se resolverían todos los demás en *NP*.

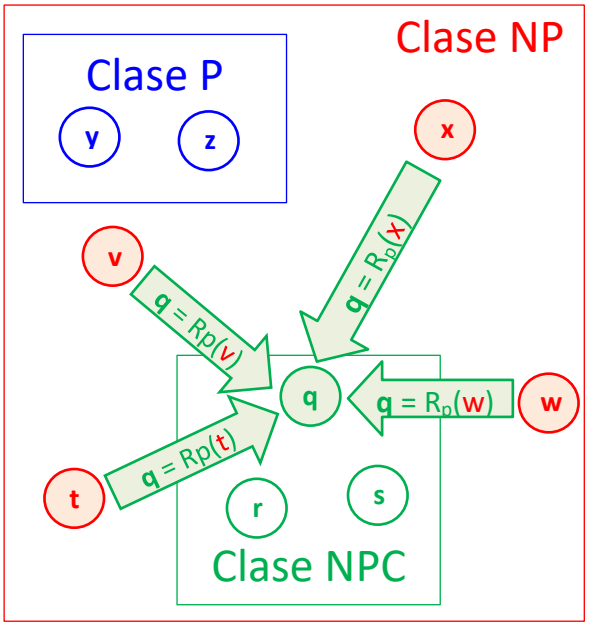
Por increíble que parezca, no sólo existe un problema así, sino que existen efectivamente muchos problemas como ese. Se designan genéricamente como *Problemas NP-Complete*, y de hecho, forman parte de una subclase de *NP* llamada *NPC* (abreviatura de *NP-Complete*). Esencialmente, la clase *NPC* es un subconjunto de *NP* y abarca a todo problema ***q*** que cumpla con las dos propiedades siguientes [1] [2]:

i. El problema ***q*** debe ser parte de *NP* (es decir: $q \in NP$).

ii. Todo problema *x* que pertenece a *NP*, debe poder reducirse a ***q*** mediante una *reducción polinómica* R_p. (Es decir: $\forall x \in NP \Rightarrow \exists R_p \,/\, q = R_p(x)$).

Gráficamente, la aparición de la clase *NPC* podría verse así:

Figura 11: Esquema de las clases *P*, *NP* y *NPC* (o *NP-Complete*).



Cualquier problema *NP-Complete* (en el gráfico, por ejemplo, los problemas *q*, *r* y *s*) entonces podría ser usado para demostrar que *P = NP*: una solución en *P* para cualquiera de ellos, pondría a toda la clase *NP* dentro de *P*, lo cual evidentemente es un camino prometedor.

Hasta aquí hemos hablado de la clase *NPC* (o *NP-Complete*) en forma muy general, y hemos dicho que contiene no uno, sino muchos problemas. Los ya mencionados problemas del *clique máximo* y del *subconjunto independiente máximo* son problemas de la clase *NP-Complete*: como veremos, se puede demostrar que todo otro problema en *NP* se puede reducir a cualquiera de estos dos. Y estos no son los únicos: ya en 1972 un investigador de las ciencias de la computación llamado *Richard Karp* publicó una lista de 21 problemas para los cuales demostró que son *NP-Complete*[5].

Ahora bien: sabemos cuáles son las propiedades de un problema *NP-Complete*, pero ¿cómo hacer para comprobar si un problema cualquiera *q* pertenece a *NPC*? ¿Cuál es el procedimiento para poder incluir un problema en la clase *NPC*? Acabamos de decir que el problema del *clique máximo* es *NP-Complete*, pero ¿cómo se demuestra eso?

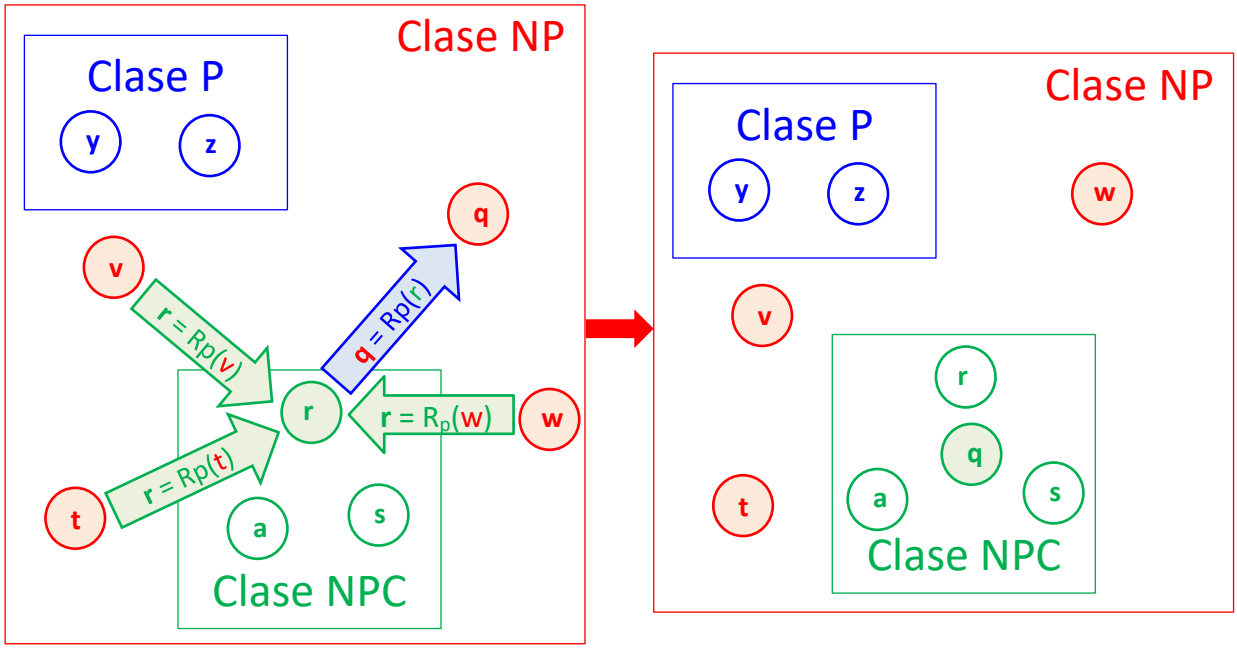
Hay dos caminos formales para demostrar que un problema *q* cualquiera es *NP-Complete*:

- **a.)** Demostrar a través de un teorema que todo problema de *NP* (sea cual sea ese problema) se puede reducir tiempo en polinomial a *q*.
- **b.)** Tomar un problema *r* del cual ya se sepa que es *NP-Complete* ($r \in NPC$) y reducir polinómicamente *r* a *q* (o sea, $q = R_p(r)$).

La forma *b* expresada en el párrafo anterior es obviamente la más simple. El siguiente gráfico muestra cuál es la idea:

[5] La lista se conoce como *Los 21 problemas NP-Complete de Karp*, y puede consultarla en la Wikipedia con el siguiente enlace: https://en.wikipedia.org/wiki/Karp's_21_NP-complete_problems.

Figura 12: Reducción de un problema NP-Complete a otro problema externo a NPC.



El concepto es directo: si todo problema de *NP* se puede reducir a *r* (ya que *r* es *NP-Complete*), entonces una solución para *r* sería una solución para todo problema en *NP*. Pero si luego *r* se reduce a otro problema *q* de *NP*, entonces cualquier solución para *q* sería una solución para *r*… y esto implica que esa misma solución lo sería para todos en *NP* (ya que todos pueden reducirse a *r*).

La alternativa *b* para demostrar que un problema es *NP-Complete* resulta la más simple y obvia, pero hay un inconveniente que el estudiante ya debe haber notado: para poder aplicar esta estrategia, debemos usar un problema *r* que ya esté en *NPC*. Sabemos que al menos el *clique máximo* y el *SIM* están en *NPC*… pero ¿cuál fue el primer problema *NP-Complete* de todos? ¿Cuál fue el problema a partir del cual pudo comenzar a aplicarse la estrategia *b*?

Ese problema se conoce como el *Problema de la Satisfactibilidad Booleana* (también conocido como el problema *SAT*) [1]. Se tiene una fórmula o expresión compuesta por un conjunto de *n* variables booleanas (variables que sólo pueden valer *true* o *false*) y se quiere saber si existe alguna combinación de valores de esas variables que hagan que la fórmula entregue un resultado *true*.

En 1971, un científico de la computación llamado *Stephen Cook* aplicó la estrategia *a* sobre este problema: mediante un teorema riguroso, probó que efectivamente todo problema de *NP* (conocido o por conocerse) puede ser reducido al *SAT* mediante una reducción polinómica, y el *SAT* se convirtió así en el primer "habitante" de la clase *NPC*. El teorema hoy se conoce como *Teorema de Cook* (y sus fundamentos están fuera del alcance de estas notas) [2]. Hoy se sabe que aproximadamente en las mismas fechas en que *Cook* demostró y presentó su teorema, otro científico soviético llamado *Leonid Levin* demostró en forma independiente el mismo teorema (aunque su publicación fue posterior, en 1973) [2]. Por

este motivo, el *Teorema de Cook* ha pasado posteriormente a designarse como *Teorema de Cook-Levin*[6].

Llegados a este punto, tenemos muy poco más que agregar a estas notas introductivas. No podemos menos que hacer notar que se han escrito muchísimos ensayos y papers orientados a demostrar la verdad o falsedad del *Problema P vs NP*. Existe al menos una página en la web dedicada a enumerar y controlar cada una de estas publicaciones, mostrando para cada una de ellas los motivos por los cuales se consideran erróneas. Se conoce como "*The P versus NP page*" (https://www.win.tue.nl/~gwoegi/P-versus-NP.htm) y recomendamos entrar en ella cada tanto, para estar al día en cuanto a nuevos intentos de solución (o en cuanto a fiascos y soluciones inaceptables… que sólo se explican por el deseo de muchos "investigadores" de ganar el millón de dólares del *Clay Mathematics Institute*).

Y para terminar, digamos que en esta Ficha hemos presentado algunas clases de complejidad que son típicamente las más comunes en cualquier discusión sobre el tema. Pero la teoría de la complejidad es mucho (pero mucho…) más extensa. Existe un número sorprendente de clases de complejidad que se han propuesto en función de diversas necesidades de investigación y clasificación. Ese número actualmente ronda en cerca de 500 clases diferentes, y puede el estudiante echarles un vistazo en la página "*Complexity Zoo*" (cuya dirección es: https://complexityzoo.uwaterloo.ca/Complexity_Zoo).

Toda esta ficha trató sobre temas avanzados, acercándonos al límite de lo que las computadoras pueden hacer, por lo que no hemos incluido la típica sección de *Temas Avanzados*.

Hasta aquí llegamos. Esperamos haber podido contribuir en forma clara, precisa y profunda con sus necesidades de estudio.

Créditos

El contenido general de esta Ficha de Estudio fue desarrollado por el *Ing. Valerio Frittelli* para ser utilizada como material de consulta general en el cursado de la asignatura *Algoritmos y Estructuras de Datos* – Carrera de Ingeniería en Sistemas de Información – UTN Córdoba, en el ciclo lectivo 2019.

Actuaron como revisores (indicando posibles errores, sugerencias de agregados de contenidos y ejercicios, sugerencias de cambios de enfoque en alguna explicación, etc.) en general todos los profesores de la citada asignatura como miembros de la Cátedra, que realizaron aportes de contenidos, propuestas de ejercicios y sus soluciones, sugerencias de estilo de programación, y planteo de enunciados de problemas y actividades prácticas, entre otros elementos.

[6] En la época en que tanto Cook como Levin hicieron sus descubrimientos en forma paralela e independiente, estaba en plena vigencia la Guerra Fría entre Estados Unidos y la Unión Soviética, y este no era un detalle menor (incluso en el campo de la ciencia). Lo tenso de las relaciones entre ambos países puso al mundo al borde la guerra nuclear varias veces desde el final de la Segunda Guerra Mundial hasta la caída del Muro de Berlín en 1989. Un acercamiento al sentir de esa época y los graves problemas que surgían (y no siempre se conocían) puede verse en la película de 2015 llamada *Bridge of Spies* (o *El Puente de los Espías*) dirigida por *Steven Spielberg* y protagonizada por *Tom Hanks*. Un espía soviético es capturado por agentes del FBI, y desde el gobierno pretenden intercambiarlo por un piloto estadounidense a su vez capturado por los soviéticos. La turbia negociación tiene lugar en Berlin Oriental (en el lado Este del Muro), y finalmente se produce el intercambio en el puente *Glienicke* que une la localidad de Potsdam con la ciudad de Berlin. Y a partir de ese primer intercambio de espías, se realizaron otros muchos intercambios en el mismo puente, lo que le valió en la prensa el nombre de "Puente de los Espías".

Bibliografía

[1] R. Sedgewick, Algoritmos en C++, Reading: Addison Wesley - Díaz de Santos, 1995.

[2] S. Wernicke, "Intro to Theoretical Computer Science," Udacity, 2015. [Online]. Available: https://www.udacity.com/course/intro-to-theoretical-computer-science--cs313. [Accessed October 2015].

[3] V. Frittelli, Algoritmos y Estructuras de Datos, Córdoba: Universitas, 2001.

[4] M. A. Weiss, Estructuras de Datos en Java - Compatible con Java 2, Madrid: Addison Wesley, 2000.