Rev.Div.Amarun, **1**, 1–15, (2014)

AMARUN | Revista de Divulgación | **AMARUN** *www.amarun.org*

Problemas del Milenio: P vs NP

Xavier Alejandro Flores Cabezas[1]

Resumen

Los problemas de complejidad computacional pueden ser resueltos mediante un número finito de pasos que transforman las entradas en salidas. Dependiendo del número de operaciones en que el problema es resuelto, éste puede clasificarse dentro de ciertas clases llamadas *clases de complejidad*. La teoría de complejidad computacional estudia las clases de complejidad y de nuestro especial interés son las clases P y la clase NP que son las clases de problemas que pueden resolverse en un tiempo polinomial, y las clases que son *verificables* en tiempo polinomial respectivamente. Si estas dos clases son equivalentes se tiene la identidad $P = NP$ lo cual implicaría que todos los problemas que pueden ser *verificados* en tiempo polinomial, pueden *resolverse* en tiempo polinomial. La comprobación de esta identidad, o bien la comprobación de la relación $P \neq NP$ puede darse haciendo uso de la clase de complejidad $NP - completo$.

1. Introducción

Supongamos que una persona decide trabajar para una editorial en el puesto de repartidor de periódicos. Su trabajo es proveer de periódicos a 15 puntos de distribución ubicados en puntos distintos de la ciudad. La ciudad en la que vive, además, cuenta con un sistema de carreteras muy limitado, habiendo muy pocas de ellas. Por supuesto, la editorial busca que se gaste la menor cantidad de gasolina posible en su trayecto, así que no les conviene que pase por un centro de distribución más de una vez. De tal manera, han encontrado ellos una ruta que satisfaga esta condición, y se la han dado al repartidor para que la siga. Esta persona comienza a trabajar entregando periódicos en las mañanas a cada uno de los 15 puntos de distribución por la ruta de la cual le han provisto. Se da cuenta que en efecto, siguiendo aquella ruta, en su ciudad de muy pocas carreteras, pasa una única vez por cada punto de distribución y al final vuelve al punto de despacho. Mientras pasa el tiempo la persona se va acostumbrando a esta ruta y empieza a preguntarse si es que existirá alguna otra ruta más eficiente, o si es que es en realidad difícil el encontrar la ruta que está siguiendo… pero no le presta mucha importancia, después de todo, la ruta que lleva funciona bien y eso es lo que importa. Cierto día, justo antes de partir en su camión del despacho, le informan que 7 de los centros de distribución han cerrado abruptamente, y que justo aquel día 10 centros más de distribución serán abiertos en lugares distintos de la ciudad. Pues bien, como aún no se ha estudiado una ruta óptima para esta nueva configuración de centros de distribución en su ciudad de carreteras limitadas, le dan la orden de que vaya a cumplir con su trabajo sin importar la eficiencia de su ruta. Sin embargo, el repartidor ha pasado mucho tiempo en la ruta anterior y no le pareció nada especial el que ésta pasara por todos los puntos de distribución una única vez en la ruta, y que piensa que tal vez pueda determinar la ruta óptima para esta nueva configuración. Así que saca su mapa y piensa hacer un cálculo rápido (gracias a sus conocimientos en matemática) para determinar la ruta deseada. Pero con el mapa en frente se da cuenta

[1]xalejandroflores@gmail.com

de que ahora tiene 18 puntos de distribución; lo primero que piensa es por cual punto empezar. Tiene 18 posibilidades, los 18 puntos. Tiene que tratar de uno en uno, así que pone su dedo en el primer punto de distribución y se fija en las carreteras que salen de él. Hay 15 maneras de moverse desde aquí, solo hay 2 puntos a los que no puede llegar directamente. Pero luego ve que a uno de esos puntos se llega únicamente por otros dos de los cuales salen solamente unas 5 carreteras y en las que entran unas 3. Y en este punto es que el repartidor deja el mapa a un lado, se recuesta en su asiento, y se pregunta a sí mismo: "He hecho ese recorrido con cierta facilidad por algún tiempo, entonces ¿Qué tan difícil puede ser el hallar una ruta de ese tipo?" Este problema se relaciona estrechamente con el tema a tratar en este artículo, pues si es que se responde la pregunta central planteada de manera afirmativa, entonces tan fácil como se viaja de punto a punto en una ruta previamente dada, se podrá encontrar la ruta óptima (dado el método pertinente).

2. Problemas y Algoritmos

Para lograr un mejor entendimiento del problema P vs NP comencemos con las bases.

Digamos que tenemos algún problema aritmético, como por ejemplo el multiplicar dos números enteros. Ahora, la manera en la que se explica la multiplicación por primera vez es que es una suma iterada, es decir, el multiplicar un número x por un número y quiere decir sumar el número x a sí mismo un número $y-1$ de veces. La primera vez que nos encontramos con las tablas de multiplicar podemos intentar aplicar este razonamiento. Entonces si queremos multiplicar dos números naturales x y y seguimos los siguientes pasos:

1. Dados dos números naturales x y y.
2. Tomar x y sumarlo a sí mismo $y-1$ veces.
3. Retornar el número obtenido.

Al aplicar este proceso, nuestro problema de hallar la multiplicación de los dos números queda resuelto. Vemos que dentro de nuestro proceso hemos utilizado una operación que es la de sumar, a la que podemos llamar una operación básica o elemental. Así, si queremos multiplicar 5 y 8, tal como hemos definido nuestro proceso de multiplicación, lo que tendríamos que hacer es tomar el 5 y sumarlo a sí mismo 7 veces, en cuyo caso estaríamos efectuando 7 operaciones básicas de suma. Pero ¿qué ocurre si tomamos números más grandes? Digamos que queremos multiplicar 5 y 1000; en este caso tendríamos que tomar el 5 y sumarlo a sí mismo 999 veces, lo cual tomaría 999 operaciones básicas de suma. Ahora digamos que en cada operación básica nos tardamos una cantidad fija de tiempo t_b en "computar". Vemos que al multiplicar 5 por 1000 nos tardaríamos un tiempo igual a $999t_b$, pero si es que modificamos nuestro proceso de multiplicación podemos acortar este tiempo. Solo basta con elegir como el número que se suma a sí mismo al mayor de los dos:

1. Dados dos números naturales x y y.
2. Reconocer cuál de los dos números es mayor.
3. Tomar el número mayor y sumarlo a sí mismo un número de veces igual al número menor menos una unidad.
4. Retornar el número obtenido.

Y ahora vemos que si queremos multiplicar 5 por 1000, siguiendo nuestro proceso, tomaremos al mayor de los dos, en este caso 1000 y lo sumaremos a sí mismo 4 veces. Como vemos en esta ocasión solo nos toma un tiempo $5t_b$ el resolver el problema mediante este método. Pero ahora, ¿qué ocurre si ambos números naturales que se desea multiplicar son muy grandes? Digamos que queremos multiplicar 18×10^{25} y 2×10^{31} en este caso, aún habiendo modificado nuestro proceso no nos salvamos de tener que repetir un

gran número de veces la operación básica de la suma. En efecto, para realizar esta operación con el método definido tendríamos que realizar $18 \times 10^{25} - 1$ operaciones básicas de suma. Para darnos cuenta de lo poco práctico que es esto, demos un vistazo al tiempo en que nos demoraríamos en computar todas esas sumas. Digamos que resolvemos esto "a mano" y que el tiempo que nos demoramos en cada operación básica de suma es constante e igual a $t_b = \frac{1}{2}s$ (medio segundo). El tiempo total que nos demoraríamos en resolver el problema con nuestro método es $T = (18 \times 10^{25} - 1)t_b = (18 \times 10^{25} - 1)0{,}5s \approx 9 \times 10^{25}s$. Lo cual es alrededor de dos trillones de años (2×10^{18}). Si es que esto no parece un número muy grande de años, tengamos en cuenta que la edad del universo es del orden de 10^{19} años.

En la práctica, sin embargo, podemos resolver este problema con cierta rapidez utilizando el siguiente método:

1. Dados dos números naturales x y y expresados de la manera: $x = A \times 10^B$ y $y = C \times 10^D$ donde al menos A y C son números naturales.
2. Reconocer cuál de los dos números A o C es mayor.
3. Tomar el número mayor y sumarlo a sí mismo un número de veces igual al número menor menos una unidad. A este nuevo número lo llamaremos S.
4. Sumar los números B y D, a este nuevo número lo llamaremos E.
5. Retornar el número $S \times 10^E$

Dicho más simplemente, multiplicamos los coeficientes y sumamos los exponentes. Como podemos ver la cantidad de operaciones básicas realizadas dependen solamente de los coeficientes, en nuestro ejemplo al multiplicar 18×10^{25} y 2×10^{31} se tendría en total 2 operaciones básicas (1 por la multiplicación de coeficientes, y una en general por la suma de exponentes), lo cual siguiendo nuestro ejemplo nos tomaría apenas un segundo. Es muy clara la diferencia.

Bien, puede presentarse otro caso que nos causaría problemas, y es justamente la multiplicación de dos números grandes x y y que no cumplan con la condición primera de nuestro nuevo método, esto es, que no puedan expresarse de la forma $x = A \times 10^B$ y $y = C \times 10^D$ donde al menos A y C son naturales. Por ejemplo, si queremos multiplicar 548796 por 654112, vemos que no son buenos candidatos para la resolución por nuestro nuevo método, mientras que por nuestro método antiguo llevaría un tiempo considerable el llegar a una respuesta (alrededor de 3 días), y sin embargo un estudiante de bachillerato podrá resolver este problema a lo sumo en unos cuantos minutos. Esto es porque para computar la multiplicación de números de este estilo utilizamos la multiplicación enseñada en el colegio que consiste en multiplicar cifra a cifra desde la derecha hasta la izquierda:

1. Dados dos números naturales x y y, cada uno con n y m dígitos respectivamente.
2. Reconocer cuál de los dos números es menor.
3. Se toma el número menor y se toma el último dígito de aquel.
4. Se multiplica este dígito por el número mayor.
5. Llámese el número obtenido h_i , donde i indica que posición tiene el dígito en el número menor contando de derecha a izquierda.
6. A la derecha inmediata de h_i se añaden $i - 1$ ceros.
7. En caso de que ya no hayan más dígitos hacia la izquierda en el número menor, sumar todos los h_i y pasar al paso 9.
8. Escoger el siguiente dígito hacia la izquierda del número menor y repetir los pasos del 4 al 6.

9. Devolver el número obtenido

Ahora, veamos cuantas operaciones básicas se tiene. Podemos tomar en cuenta que el reconocer el menor de dos números es una operación básica, así como el tomar, modificar o el devolver un número, pero por el momento solo vamos a centrarnos en las operaciones aritméticas, esto es sumar. En los pasos del 1 al 3 no hay operaciones, en el paso 4 hablamos de la multiplicación de un dígito por un número. Como se trata de un dígito (número entero del 0 al 9), podemos utilizar uno de nuestros métodos anteriores, el cual se resolverá con a lo sumo 8 operaciones básicas (invitamos al lector a que compruebe esto). Por el paso 7, esto se repite tantas veces como dígitos tenga el número menor. Si es que el número de dígitos que tiene el número menor es n, entonces a lo sumo se tienen $8 \times n$ operaciones básicas de suma hasta este punto. Luego tenemos en el paso 8 una suma, en general será una suma de n elementos. Si es que tenemos en cuenta que una operación básica es la suma uno a uno entre dos números, entonces en el paso 8 se tendrían $n - 1$ operaciones básicas. Sumando, tendríamos en total, que en el peor de los casos (esto es, el número menor consiste solo de dígitos iguales a 9) se tiene un número de operaciones básicas igual a $8n+n-1$ que es $9n - 1$. En nuestro ejemplo de 548796 por 654112 vemos que $n = 6$ puesto que el menor número tiene 6 dígitos, en cuyo caso a lo sumo el proceso lleva un número de operaciones básicas de 53. De nuevo, si nos demoramos medio segundo en cada operación básica, en medio minuto ya se habría resuelto el problema. En comparación con los tres días que toma su resolución con el método anterior, podemos ver que hay un gran salto en lo que llamamos "eficiencia".

Observemos que el *problema* inicial siempre es el mismo: tenemos dos números, y queremos saber cuánto es la multiplicación de ambos. Sin embargo, podemos ver que dependiendo de los números que elijamos, se presentan dificultades específicas. El problema en sí se generaliza para dos números cualesquiera, pero si es que ponemos un ejemplo concreto de números que queremos multiplicar, a esto se le llama *instancia*. Nuestras instancias fueron 5 y 8, 5 y 1000, 18×10^{25} y 2×10^{31} , y por último 548796 y 654112. Ahora, luego de tener claro cuál es nuestro problema, procedimos a dar una lista de pasos a seguir para poder resolverlo. A esta lista (finita) de pasos a seguir se le llama *algoritmo*. Y por último, al llevar nuestras instancias a través de cada algoritmo, al final nos quedamos con lo que especificamos del problema, la solución a la multiplicación que de manera general se le llama *salida*.

Podemos ver cómo algunos de nuestros algoritmos funcionan en menos tiempo que otros pero el problema es siempre el mismo: multiplicación de dos números naturales. Como vemos, existen varios algoritmos (métodos) para solucionar el problema, mientras menos tiempo y menos recursos ocupe un algoritmo se dice que es más *eficiente*. En nuestro ejemplo es claro que los primeros dos algoritmos presentados son bastante ineficientes, mientras que los dos últimos son mucho más eficientes dependiendo el tipo de número que se tenga para multiplicar. Como es natural, para cada problema se busca el algoritmo que lo resuelva ocupando la menor cantidad de recursos en el menor tiempo posible, y es justamente esto de lo que se ocupa la teoría de complejidad computacional.

Nótese que hasta ahora hemos utilizado el término *computar* muy seguido; contrariamente a lo que se pueda pensar, cuando hablamos de una computación no nos referimos necesariamente a un proceso en el que toma parte una computadora como las que conocemos y utilizamos hoy en día; en efecto, el concepto de computar se tenía desde muchos años antes a la aparición de la primera computadora electrónica en el siglo pasado. Entendemos por *computar*, el proceso de producir una salida, dado un conjunto de entradas en un número finito de pasos. Un problema de computación puede ser algo tan simple como tomar un par de números y sumarlos, o algo un poco menos trivial como: dada la localización de dos ciudades y varias redes de carreteras que las conectan, encontrar el camino más corto entre las ciudades.

Pero quedémonos en nuestro ejemplo. Como pudimos ver, el número de operaciones básicas que se llevan a cabo en cada algoritmo siempre depende del tipo de instancia, o más específicamente del tamaño de la entrada. Tomemos el último algoritmo que tratamos, el de la multiplicación por el método enseñado en el colegio. Hicimos un análisis del algoritmo y pudimos ver que en el peor de los casos, si es que el número de dígitos que tiene el número menor es n, se llevarán a cabo $9n - 1$ operaciones. Esto es, en el caso en que

todos los dígitos del número menor sean solo 9. Pero de manera general los dígitos pueden ser diferentes dependiendo la instancia. Ya que, al desarrollar un algoritmo no podemos controlar las instancias que se computen, no tenemos idea cuántas operaciones básicas se llevarán a cabo cada vez, lo único que podemos asegurar es que no será un número mayor a $9n - 1$. Esto es, el número de operaciones básicas de nuestro algoritmo se encuentra acotado por $9n - 1$, donde n es el número de dígitos que tiene el menor número entero. En el caso del primer y segundo algoritmo, pudiéramos decir que el número de operaciones básicas se encuentran acotadas por $n-1$, e incluso se llega a la igualdad, es decir, sea cual sea la entrada se llevaran a cabo $n-1$ operaciones básicas. La gran diferencia (recordarán que estos algoritmos eran extremadamente ineficientes) es que aquí n representa el valor del número menor, y entonces se llevarán a cabo muchas más operaciones que en el caso en que n es el número de dígitos de un número. Sin embargo, en ambos casos podemos decir que el número de operaciones que ocurren para cada problema dependen del "tamaño" de la entrada.

Como vimos también, ya sea una computadora electrónica o un ser humano, en cada operación básica se demora cierto tiempo que podemos considerar constante por cada operación. Entonces, si un algoritmo incurre en un número muy grande de operaciones básicas, el tiempo que le llevará el ejecutarse será muy grande en comparación a un algoritmo que tiene un número muy bajo de operaciones básicas. Podemos ver entonces, que el número de operaciones básicas que realiza un algoritmo va ligado estrechamente al tiempo en que este algoritmo "corre".

3. Verificando un Problema

Consideremos un problema un poco más complicado, como por ejemplo el problema de ordenamiento, el cual dice: dado un cierto arreglo de números reales, queremos ordenarlos de manera ascendente. En este caso tenemos:

Entrada:	arreglo de reales $(m_1, m_2, m_3, ..., m_n)$
Salida:	arreglo de reales $(m'_1, m'_2, m'_3, ..., m'_n)$ que es una permutación del arreglo de entrada y cumple con $m'_1 \leq m'_2 \leq m'_3 \leq ... \leq m'_n$

Hay varios algoritmos que resuelven este problema en particular, a los cuales se los llama *algoritmos de ordenamiento*. Dependiendo del algoritmo de ordenamiento utilizado el tiempo que se demora en entregar la salida puede variar, pero tenemos que preguntarnos: ¿Cómo sabemos que el algoritmo utilizado devuelve una salida satisfactoria?, dicho de otra manera: ¿Cómo verificamos que el algoritmo cumple con los criterios del problema? La única manera que tenemos de verificar que el algoritmo utilizado para resolver el problema, resuelve efectivamente el problema, es el analizar la salida del algoritmo. Decimos entonces que el algoritmo procesa las instancias y devuelve un certificado (salida). El analizar el certificado de cierto problema constituye un nuevo problema en sí mismo, y como tal deberá ser tratado con el algoritmo correspondiente. Si tomamos el problema de ordenamiento, verificar el certificado significa comprobar que en efecto se tiene que $m'_1 \leq m'_2 \leq m'_3 \leq ... \leq m'_n$. Esto se puede llevar a cabo mediante el siguiente algoritmo que verifica de uno a uno los elementos adyacentes del arreglo:

1. Fijamos el entero i , y lo inicializamos en $i = 2$
2. Comprobamos que $m_{i-1} \leq m_i$
3. Si es que la comprobación no es satisfactoria, retornar un certificado de FALSO y detener el algoritmo.
4. Si es que se tiene que i es igual al tamaño del arreglo, devolver un certificado de VERDADERO y detener el algoritmo.
5. Si es que la comprobación es satisfactoria, e $i + 1$ es menor que el tamaño del arreglo, fijar $i = i + 1$ y volver al paso 2.

Podemos ver que este algoritmo difiere de los anteriores. Antes que nada vemos que hemos hecho uso de un número i que es de uso solamente dentro de nuestro algoritmo, es decir este número que hemos utilizado como *contador* no figura ni en las entradas ni en la salida. Luego, fijémonos que mientras que los algoritmos anteriores eran utilizados para devolver un resultado numérico, en este caso lo que queremos hacer es verificar que el arreglo está ordenado ascendentemente, no queremos que retorne nada en particular, solo pedimos la confirmación o la negación de que en efecto el arreglo se encuentra ordenado de tal manera. Por esta razón vemos en la línea 3 que si para cualquier par de elementos adyacentes, el actual es mayor que el siguiente, entonces el arreglo no se encuentra ordenado ascendentemente y devuelve un certificado de FALSO, mientras que si se comprueba que todo par de elementos adyacentes del arreglo se encuentra ordenado ascendentemente, entonces llega a la línea 4 y termina devolviendo un certificado VERDADERO que nos indica que todo el arreglo se encuentra ordenado ascendentemente. Analicemos el algoritmo: en este caso sí vamos a tomar en cuenta que el hecho de comparar un par de elementos cuenta como una operación elemental. De esta manera, vemos que la línea 2 en que se realiza una comparación entre elementos del arreglo se repite un número $n-1$ de veces, donde n es el número de elementos que tiene el arreglo.

4. Tipos de Problemas y Tiempos

Podemos ver hasta ahora que hay por lo menos dos tipos de problemas abstractos:

Problemas de Decisión.

En estos problemas existe un conjunto de soluciones de solamente dos elementos: $\{0, 1\}$ en donde podemos pensar en 0 como un "no" y en 1 como un "sí". Existe un conjunto de instancias, digamos I, para las que se comprueba que su solución es 1. Un problema de decisión consiste en comprobar si es que una instancia en particular i, pertenece o no al conjunto I. Esto es, si es que la solución de esta instancia es "si" o "no". Un ejemplo puede ser el comprobar si es que un número es mayor o igual que cero; en este caso se tiene $I = [0; +\infty[$; si es que el número que queremos analizar se encuentra dentro de este conjunto, entonces nuestro algoritmo arroja "si" y comprueba, en efecto, que es mayor o igual que cero. Caso contrario arroja "no".

Problemas Funcionales

Estos problemas son muy parecidos a los problemas de decisión en el sentido en que transforman un conjunto de instancias en una sola salida; lo distinto es que la salida ya no es únicamente un 0 o un 1 (un certificado booleano), sino que pueden ser resultados mucho más complejos en su estructura. Un ejemplo de este tipo de problemas es la factorización de un entero: dado un entero queremos encontrar todos los factores primos que lo componen. Podemos ver que la salida de este problema será una cadena de números primos y no solamente un"sí" o un "no", pero aún a pesar de esto, un problema funcional siempre puede expresarse como un problema de decisión. En nuestro ejemplo de factorización de un entero podemos reformular el problema de la siguiente manera: dado un número entero y una cadena específica de números primos, decidir si es que el número entero se puede factorizar completamente en la cadena de números primos.

En efecto, como veremos más adelante, las clases P y NP corresponden solamente a problemas de decisión, pero el hecho de que siempre podamos expresar un problema funcional como un problema de decisión hace tanto más relevante la existencia de estas clases.

Tiempos de Ejecución

Antes de entrar en clases de complejidad, hablemos de tiempos. Ya vimos que el hablar del tiempo en que corre un algoritmo es lo mismo que hablar del número de operaciones elementales que este algoritmo

realiza. Entonces, digamos que cierto algoritmo A tiene un número de operaciones básicas dado por una función:

$$\begin{aligned} f: \quad & \mathbb{N} \to \mathbb{N} \\ & n \mapsto f(n) : f(n) \text{ es el número de operaciones básicas que ejecuta el algoritmo} \end{aligned}$$

en donde n es el tamaño de la entrada. Esta función se llama *tiempo de ejecución* del algoritmo A.

Tomemos como ejemplo el último algoritmo de multiplicación que definimos. Vimos que en el peor de los casos, había $9n-1$ operaciones básicas ejecutadas. En realidad, lo único que nos interesa (casi siempre) es el peor de los casos en que corre un algoritmo, ya que necesitamos generalizar su eficiencia. Entonces podemos ver que para este algoritmo $f(n) \leq 9n-1$, es decir que f se encuentra acotado por $9n-1$ en donde podemos ver que $9n-1$ tiene la forma de un polinomio de grado 1. De manera general, se clasifican los problemas dependiendo de cómo se encuentran acotados los tiempos de ejecución de sus algoritmos de resolución más eficientes. Nos interesa el tipo de función que acota al tiempo de ejecución, mas no la cota exacta. Entonces podemos decir que nuestro algoritmo ejemplo se encuentra acotado de manera general por un polinomio de grado 1:

$$(\exists \alpha > 0)\ (\forall n > 0) : f(n) \leq \alpha n$$

Hablando de manera general, hacemos uso de la notación de crecimiento asintótico utilizada usualmente para determinar los tiempos de ejecución:

$$f(n) = O\big(g(n)\big) \iff (\exists \alpha > 0)\ (\exists n_0 > 0)\ (\forall n > n_0) : f(n) \leq \alpha g(n)$$

Y decimos que $f(n)$ corre en un tiempo $g(n)$. En nuestro ejemplo anterior $g(n)$ es un polinomio de grado 1, y decimos que $f(n)$ corre en tiempo polinomial. Pero démonos cuenta de algo en esta notación: nos está diciendo que el tiempo de ejecución no tiene que estar acotado por $g(n)$ para cualquier tamaño de instancia necesariamente. Nos dice que tiene que existir algún tamaño n_0 desde el cual el tiempo de ejecución se encuentra acotado por $g(n)$, sin importar lo que suceda con tamaños de datos de entrada menores. Esto nos dice que lo que nos interesa es que el tiempo de ejecución esté acotado por alguna función determinada para grandes tamaños de datos de entrada, es decir, no es de gran importancia cómo un algoritmo se comporta con datos de entrada de pequeño tamaño, lo que nos interesa es que a medida que el tamaño de la instancia incremente, el tiempo de ejecución se vaya "regularizando".

Podemos entonces clasificar los problemas dependiendo de sus tiempos de ejecución más eficientes. Por la manera en que crecen las funciones mientras el tamaño de las instancias aumenta, se tiene que las funciones logarítmicas ($\log(n)$) son las más eficientes, seguidas por las polinómicas (n^k), y luego las exponenciales (2^n). Para ver esto más claramente consideremos una relación entre número de operaciones básicas realizadas por algoritmos con diferente tiempo de ejecución:

	n					
	5	**10**	**50**	**100**	**500**	**1000**
$log(n)$	2	3	6	7	9	10
n	5	10	50	100	500	1000
n^2	25	100	2500	10000	250000	1000000
n^3	125	1000	125000	1000000	$1{,}3x10^8$	10^9
2^n	32	1024	$1{,}126 \times 10^{15}$	$1{,}3 \times 10^{30}$	3×10^{150}	$1{,}072 \times 10^{301}$

Podemos ver que la diferencia entre un algoritmo de tiempo logarítmico y uno de tiempo exponencial es bastante notoria. Pero para tener un poco más claro la diferencia entre tiempos, digamos que cada operación básica toma $0{,}001$ segundos en resolver, en este caso tendríamos:

	n					
	5	**10**	**50**	**100**	**500**	**1000**
$log(n)$	$0{,}002s$	$0{,}003s$	$0{,}006s$	$0{,}007s$	$0{,}009s$	$0{,}01s$
n	$0{,}005s$	$0{,}01s$	$0{,}05s$	$0{,}1s$	$0{,}5s$	$1s$
n^2	$0{,}02s$	$0{,}1s$	$2{,}5s$	$10s$	4 min	16 min
n^3	$0{,}1s$	$1s$	2 min	16 min	$34h$	11 días
2^n	$0{,}03s$	$1s$	35702 años	4×10^{16} milenios	1×10^{137} milenios	4×10^{287} milenios

La comparación entre una centésima de segundo y 4×10^{287} milenios para procesar un mismo tamaño de datos de entrada es bastante considerable. Pero pudiéramos decir que a medida que pase el tiempo las computadoras se seguirán volviendo más y más potentes hasta poder realizar algoritmos de tiempos exponenciales en tiempos más razonables, ¿cierto? Veamos la siguiente tabla, la que nos indica cómo es que mejorarían los tiempos de ejecución de cada caso a medida que mejora la tecnología de computación.

La siguiente tabla muestra qué tan grande es la instancia que se puede resolver en una hora de tiempo, y como este tamaño va cambiando a medida que las computadoras se vuelven más y más rápidas, dicho de otra manera, cuántas operaciones básicas pueden realizarse en una hora a medida que las computadoras incrementen su velocidad de computación:

	Hoy	**10x más rápidas**	**100x más rápidas**	**1000x más rápidas**
$log(n)$	N_1	N_1^{10}	N_1^{100}	N_1^{1000}
n	N_2	$10N_2$	$100N_2$	$1000N_2$
n^2	N_3	$3{,}16N_3$	$10N_3$	$31{,}16N_3$
n^3	N_4	$2{,}15N_4$	$4{,}64N_4$	$10N_4$
2^n	N_5	$N_5 + 3{,}32$	$N_5 + 6{,}64$	$N_5 + 9{,}97$

Ahora la situación es mucho más notable, porque vemos que mientras los algoritmos polinomiales incrementan el número de instancias que se pueden realizar en una hora por un factor que las multiplica, en el caso de las exponenciales se incrementan solamente un número fijo de operaciones; *10 operaciones más, con computadoras 1000 veces más rápidas*.

En efecto a los problemas que son resueltos por un algoritmo que corre en tiempo polinomial o más rápido se lo dice *tratable*, mientras que a los problemas que no se pueden resolver vía un algoritmo polinomial o más eficiente les llamamos *intratables*.

5. Clases de Complejidad

Como se puede esperar, hay un sinnúmero de problemas abstractos que pueden ser computados. Se puede además hacer una clasificación de estos problemas en base a sus tiempos de ejecución y recursos que ocupan (cantidad de memoria). Las categorías en que se encuentran estos problemas se llaman clases de complejidad. Unos ejemplos de clases de complejidad son las siguientes:

Clase de Complejidad EXP

Un problema de decisión C se dice que pertenece a la clase de complejidad EXP, si es que se resuelve en tiempo exponencial. Esto significa que a lo mucho se han encontrado algoritmos cuyos tiempos de ejecución se encuentran acotados por una función exponencial. Ahora bien una función exponencial es de la forma:

$$e(n) = a^{f(n)}$$

De manera general $a \in \mathbb{R}^+$ y f es una función que depende del tamaño de las instancias n. Podemos pedir sin perder generalidad que $f(n)$ no sea logarítmica, ya que de serlo $e(n)$ no es necesariamente exponencial. En complejidad computacional, sin embargo, al hablar de tiempos de ejecución exponenciales nos referimos por lo general a tiempos de ejecución acotados por una función del tipo:

$$e(n) = 2^{p(n)} \text{ donde } p(n) \text{ es un polinomio.}$$

Entonces decimos que un problema C pertenece a EXP si es que cumple con los siguientes criterios:

- C es un problema de decisión.
- C se puede resolver en tiempo exponencial. Esto es, existe un algoritmo que resuelve C cuyo tiempo de ejecución es $f(n)$ tal que:

$$(\exists \alpha > 0)\ (\forall n > n_0) : f(n) \leq \alpha 2^{p(n)}, \text{ donde } p(n) \text{ es un polinomio.}$$

Una función de este tipo se comporta de tal forma que mientras n sigue creciendo, el tiempo incrementa rápidamente. Es decir, mientras más grande es la entrada, mayor tiempo se demorará el algoritmo en arrojar un resultado. A decir verdad, no es necesario un tamaño muy grande en las instancias para que el tiempo comience a crecer de manera muy rápida; para un tiempo ejecución exponencial "simple" de $e(n) = 2^n$, vemos que para un valor n de 5, nuestra función $e(n)$ llevará a cabo 32 iteraciones, pero basta con fijar el tamaño de la instancia en $n = 10$ y vemos que ahora tenemos 1024 iteraciones. Veamos un problema que se encuentra dentro de esta clase:

Satisfacibilidad Booleana

Podemos pensar en una variable booleana como una variable que acepta solamente dos tipos de valores: un afirmativo o un negativo, a los cuales les podemos asignar los números 1 y 0 respectivamente. Las variables booleanas pueden conectarse mediante operadores lógicos como $\wedge$ (conjunción) $\vee$(disyunción) $\neg$(negación), y se operan siguiendo las reglas de la lógica matemática[2]. Una expresión booleana es una expresión algebraica cuyo resultado es o bien afirmativo o bien negativo.

El problema de satisfacibilidad booleana dice: dada una expresión booleana con variables booleanas, decidir si existe una instancia de sus variables que hacen que la expresión retorne un valor afirmativo.

Para que esto quede más claro pongamos un ejemplo puntual: sea la expresión booleana

$$\big(a \wedge b\big) \vee \big(\neg c \vee (d \wedge \neg a)\big),$$

donde a, b, c y d son variables booleanas. Queremos ver si alguna combinación de valores para cada variable resulta en que la expresión nos de un resultado verdadero. Intuitivamente este problema no parece muy complicado aún más tratándose de un ejemplo de tan pocas variables booleanas, pero veremos que en efecto este es un problema intratable debido a su largo tiempo de resolución a medida que el número de variables incrementa.

La forma de resolver este problema es haciendo uso de *tablas de verdad* (para más detalles véase [4]). Como sabemos, cada variable booleana puede tener dos posibles valores: 1 o 0, luego si fijamos para una variable su valor, queremos comprobar para la siguiente variable sus dos valores posibles, y así sucesivamente hasta haber fijado valores para todas las variables. Entonces, si es que tenemos n variables booleanas, tendremos un total de 2^n combinaciones.

[2]El lector puede encontrar una amplia bibliografía concerniente a la teoría lógica matemática en cualquier biblioteca o fácilmente en internet, por ejemplo [4]

Ilustremos esto con nuestro ejemplo:

$(a$	$\wedge$	$b)$	$\vee$	$(\neg$	c	$\vee$	$(d$	$\wedge$	$\neg$	$a))$
V		V			V		V			V
V		V			V		F			V
V		V			F		V			V
V		V			F		F			V
V		F			V		V			V
V		F			V		F			V
V		F			F		V			V
V		F			F		F			V
F		V			V		V			F
F		V			V		F			F
F		V			F		V			F
F		V			F		F			F
F		F			V		V			F
F		F			V		F			F
F		F			F		V			F
F		F			F		F			F

En esta tabla de verdad la letra V sale de "Verdadero" y representa una respuesta afirmativa y la letra F sale de "Falso" y representa una respuesta negativa. Vemos que cada fila tiene una configuración distinta de los valores de las distintas variables; hay un total de 16 filas ya que son 2^n casos y tenemos que $n = 4$ por tener 4 variables booleanas. Para verificar cual de estas combinaciones nos da una salida afirmativa tenemos que hacer un análisis exhaustivo caso por caso de cada fila y verificar su resultado. Este tipo de análisis exhaustivo es la manera menos eficiente de resolver un problema, pero esta es la manera en que se resuelve el problema de satisfacibilidad booleana y no hay aún un método más eficiente. Podemos ver que como hay 2^n casos, tenemos que hacer cierto número de operaciones fijo 2^n veces, (en nuestro ejemplo de 4 variables tenemos que realizar las operaciones 16 veces) lo cual nos define un tiempo exponencial, y por lo tanto este problema se encuentra acotado por una función exponencial y pertenece a la clase de complejidad EXP.

Existen alrededor de 500 clases de complejidad [3]. Cabe notar que cuando hablamos de comprobar un problema se refiere a un algoritmo utilizado para verificar el certificado emitido al resolver el problema (esto se explicará más adelante).

El hecho de que un problema pueda ser resuelto en cierto tiempo significa que existe al menos un algoritmo que lo resuelva en esas condiciones. Pueden existir un sinnúmero de algoritmos de tiempo exponencial para resolver cierto problema, pero basta con que exista un algoritmo que corra en tiempo polinomial para que estemos en el derecho de decir que el problema se puede resolver en tiempo polinomial. Nuestra atención se concentrará en las clases P y NP.

Clase de complejidad P

Decimos que un problema D se encuentra en la clase de complejidad P si es que cumple con los siguientes criterios:

- D es un problema de decisión.
- D se puede *resolver* en tiempo polinomial.

Aquí "tiempo polinomial" nos da la noción de que el problema puede resolverse con un algoritmo que corra en un tiempo relativamente rápido. Formalmente el que un algoritmo D que depende del tamaño de la entrada n corra en tiempo polinomial significa que:

[3] https://complexityzoo.uwaterloo.ca/Complexity_Zoo

$(\exists \alpha > 0)\ (\exists n_0 > 0) : D(n) \leq \alpha_1 p(n)\ ;\ (\forall n \geq n_0)$, donde $p(n)$ es un polinomio.

Podemos hacer uso de la notación usual antes explicada:

$$D = O\big(n^k\big)$$

Y entonces decimos que un algoritmo D corre en *tiempo polinomial*.

Diremos que un algoritmo que corre en tiempo polinomial es "eficiente", pero por supuesto, algún proceso acotado por un polinomio de grado muy alto (10000 por ejemplo) difícilmente se lo puede considerar como "rápido" o "eficiente". Por lo general los problemas que se resuelven en tiempo polinomial lo hacen con un grado relativamente pequeño. Un ejemplo de este tipo de problema es el problema de la multiplicación entre dos números naturales que comenzamos tratando en este artículo, pues como vimos, el tiempo de ejecución se encontraba acotado por un polinomio de grado uno.

Clase de Complejidad NP

Decimos que un problema E se encuentra en la clase de complejidad NP si es que cumple con los siguientes criterios:

- E es un problema de decisión.
- E es *verificable* en tiempo polinomial.

Recordemos que con "verificar"nos referimos a comprobar si el certificado emitido por el algoritmo cumple con los requerimientos del problema. Un problema pertenece a la clase NP si el algoritmo usado para verificar su certificado corre en tiempo polinomial. Como un ejemplo de un problema que se encuentre en la clase de complejidad NP podemos nombrar al problema de ordenamiento antes mencionado donde vimos cómo la verificación del resultado se encuentra acotada por un polinomio de grado 1.

Ahora, de todos los algoritmos de ordenamiento podemos ver que mientras algunos corren en tiempo polinomial como el llamado *bubble sort*[4], en principio podríamos definir un algoritmo de ordenamiento que corra en tiempo exponencial, pero basta con que exista uno que se pueda resolver en tiempo eficiente para que este problema pertenezca a la clase P. De manera general, se tiene que si un problema puede ser resuelto en tiempo polinomial, entonces también puede ser comprobado en tiempo polinomial, es decir: $P \subseteq NP$. Surge, entonces la pregunta: ¿Es esta contenencia estricta? O bien ¿Si es que un problema es verificable en tiempo polinomial, se puede encontrar su solución también en tiempo polinomial?

Para explorar esta pregunta y dar una mejor idea de como responderla tenemos que definir una clase de complejidad que se encuentra dentro de NP llamada la clase de complejidad $NP - Completo$.

Clase de Complejidad $NP - Completo$

Un problema D es $NP - Completo$ si cumple con los siguientes dos criterios:

- D pertenece a la clase de complejidad NP.
- Cualquier problema en NP (incluidos los mismos problemas en $NP - Completo$) se puede reducir polinomialmente en D.

El primer criterio es claro y con este establecemos la relación $NP - Completo \subseteq NP$. Lo que nos dice el segundo criterio es que para cualquier problema F dentro de NP, existe una función llamada *reducción polinomial* que toma las instancias de F y las transforma en instancias de D, tal que una instancia de F

[4]Para una explicación de este algoritmo y otros algoritmos de ordenamiento se puede revisar [1, p. 140-196]

devuelve una salida afirmativa si y solamente si su reducción polinomial en D devuelve una salida afirmativa. Además la reducción polinomial debe de computarse en tiempo polinomial.

Dicho de otra manera, una *reducción polinomial* es un algoritmo que corre en tiempo polinomial y convierte las entradas de un problema de decisión en entradas de otro problema de decisión tal que, el hecho de que se comprueben para uno de los problemas, significa necesariamente que se comprueba para el otro.

Quizá se pueda ver ya lo importante que son los problemas en esta clase. El punto clave aquí es que para cualquier problema dentro de NP existe una reducción polinomial en un problema $NP - Completo$, esto quiere decir que siempre podemos tomar un problema arbitrario dentro de NP y tratarlo como un problema $NP - Completo$. Luego, si es que podemos encontrar un algoritmo que corra en tiempo polinomial y resuelva un problema $NP - Completo$, entonces, por reducción polinomial, estaríamos diciendo que cualquier problema dentro de NP puede resolverse en tiempo polinomial. Así, es ahora mucho más claro la manera en que se orienta la demostración del problema $PvsNP$.

Para ver un ejemplo de un problema en $NP - Completo$ volvamos al problema de la satisfacibilidad booleana. Hemos probado que este ejemplo se encuentra en la clase EXP pues su tiempo de ejecución se encuentra acotado por una función exponencial, pero para verificar el certificado (esto es, las combinaciones en donde la expresión booleana retorna un valor positivo) dado que tenemos m cadenas de valores para cada variable en donde supuestamente la expresión retorna un valor afirmativo, tenemos que realizar las operaciones lógicas un número m de veces, una vez para cada caso, ya que solo queremos comprobar el resultado obtenido por las cadenas ingresadas. Entonces podemos ver que el tiempo de ejecución viene dado por la expresión $f(m) = m$ la cual representa un polinomio de orden 1. Como el certificado del problema de satisfacibilidad booleana es verificable en tiempo polinomial, decimos entonces que este problema pertenece a la clase NP. Pero además, en [3] Stephen Cook comprueba que cualquier problema dentro de NP se lo puede expresar como una reducción polinomial, y entonces el problema de satisfacibilidad booleana pertenece a la clase $NP - Completo$.[5]

6. ¿$P = NP$?

En mayo del año 2000 el *Clay Mathematics Institute* (CMI) de Cambridge, E.E.U.U. anunció los siete problemas del milenio; estos serían siete problemas matemáticos abiertos que son de gran importancia en su respectivo ámbito. La junta del CMI ofrece un premio de 7 millones de dólares a quien fuere capaz de proveer una solución formal y válida a estos problemas, un millón dirigido a cada problema. En particular, y motivo principal de este artículo es la conjetura P vs NP la cual plantea la pregunta: ¿Se cumple que $P = NP$?

Al día de hoy, se tiene una inclusión $P \subseteq NP$ solamente. Es decir, se sabe que si un problema puede resolverse en tiempo polinomial, entonces su certificado es verificable en tiempo polinomial, pero no se ha podido comprobar la validez de la afirmación $NP \subseteq P$, es decir, no sabemos aún si para un problema cuyo certificado es verificable en tiempo polinomial, se puede encontrar un algoritmo que lo resuelva en tiempo polinomial. Si es que esto fuera cierto, se tendría la identidad $P = NP$. Y es justamente este uno de los siete problemas del milenio. Lo que se conjetura es que $P \neq NP$, es decir que existen problemas de certificado verificable en tiempo polinomial que no se pueden resolver en tiempo polinomial. La clase de problemas $NP - Completo$ son los que más probablemente no se encuentren en P. Sabiendo que cualquier problema en NP lo podemos transformar polinomialmente en un problema de $NP - Completo$ podemos ver que la manera de acercarse a esta conjetura es clara:

- $P = NP$ si y solo si $P \cap NP - Completo \neq \emptyset$

Por un lado, si $P = NP$ entonces todos los problemas en NP se pueden resolver en tiempo polinomial, y como $NP - Completo$ se encuentra contenido en NP entonces los problemas en $NP - Completo$ se pueden

[5] Para más detalles de esta clase de complejidad revisar [5].

resolver también en tiempo polinomial y por lo tanto $P \cap NP - Completo \neq \emptyset$ ya que los problemas en $NP - Completo$ pertenecen también a P.

Por otro lado, si $P \cap NP - Completo \neq \emptyset$, entonces existe al menos un elemento en $NP - Completo$ que puede resolverse en tiempo polinomial, pero como ya vimos, cualquier problema en NP lo podemos reducir polinomialmente a cualquier problema en $NP - Completo$. Entonces podemos reducir polinomialmente todos los problemas en NP al problema en la intersección entre P y $NP - Completo$, el cual se resuelve en tiempo polinomial, y como tal, todos los problemas en NP se pueden resolver en tiempo polinomial. Luego $P = NP$ pues $P \subseteq NP$ y todos los elementos de NP estarían en P.

Entonces vemos que para comprobar que $P = NP$ debemos solamente encontrar un problema que se encuentre en la intersección de P y $NP - Completo$. Esto es, un problema que se resuelva en tiempo polinomial y al cual cualquier problema en NP pueda ser reducido polinomialmente.

Para comprobar que $P = NP$, sin embargo, lo que tenemos que hacer es comprobar que no existe ningún problema que se resuelva en tiempo polinomial y al cual cualquier problema en NP pueda ser reducido polinomialmente. Podemos pensar en esto como una prueba exhaustiva de problemas, lo cual es mucho más complicado que el caso anterior.

Obviamente, hasta la fecha no se han encontrado problemas que se encuentren en la intersección de P y $NP - Completo$, por lo cual este es aún un problema abierto. Los problemas ubicados en $NP - Completo$ son considerados los problemas NP más "difíciles" de resolver, y esta noción de "dificultad" puede generalizarse para otras clases, habiendo subclases "Completas" para otras clases que representan las subclases que contienen a los problemas que pueden reducirse polinomialmente los demás problemas de la clase principal.

Luego de que hemos visto la naturaleza del problema en sí, es natural preguntarnos: ¿Por qué este problema amerita tanto interés?

7. La brecha entre verificar y resolver

Para poder empezar a contestar esta pregunta, tengamos en cuenta de que la conjetura es que $P \neq NP$, mas no que $P = NP$, lo que significa que aún sin tener pruebas concretas, se cree y se espera que $P \neq NP$. ¿Por qué es eso? Empecemos por lo más natural: como vimos, para comprobar que $P = NP$ nos bastará solamente con encontrar un algoritmo polinomial que resuelva un problema $NP - Completo$, lo cual sería relativamente simple si es que en efecto se cumpliera que $P = NP$. Pero, para comprobar que $P \neq NP$ lo que tenemos que comprobar es que no existe ningún elemento en la intersección de P y $NP - Completo$, lo cual representa mayor dificultad. El problema fue planteado en 1971 y desde entonces ha habido un gran esfuerzo por llegar a una solución, incluso está el incentivo del millón de dólares por parte del Clay Mathematics Institute[6] , y aún así no se ha podido llegar a una solución. Ciertamente el problema es bastante difícil de resolver, lo cual de alguna manera refleja la realidad hipotética que $P \neq NP$. El hecho de ser $NP - Completo$ quiere decir que el tiempo en que se demora su resolución crece rápidamente conforme se incrementa el número de instancias, siendo a menudo completamente impráctico el intentar resolverlo directamente.

Existe un sinnúmero de problemas $NP - Completos$ aplicados a diferentes ramas de la ciencia y la ingeniería. Existen problemas $NP - Completos$ para:

[6]Para ver una explicación del problema $PvsNP$ de parte del Clay Mathematics Institute ver [2]

Grafos

- **Problema de inspección de rutas:** Encontrar el camino cerrado más corto que pase por todos los puntos de un grafo no dirigido.
- **Problema del ciclo Hamiltoniano:** Determinar si en un grafo (dirigido o no) existe un camino que visite a todos los nodos una única vez. Justamente es este problema que empezamos planteando en el ejemplo dado en la introducción. Como vemos encontrar si es que tal camino existe no es un asunto trivial.
- **Problema del camino más largo:** Encontrar el camino más largo que pase por todos los puntos de un grafo no dirigido.

Programación matemática

- **Problema de 3-partición:** Dado un conjunto de enteros, decidir si este puede ser particionado en tripletas que sumen la misma cantidad.
- **Emparejamiento numérico en 3 dimensiones:** Dados tres conjuntos de enteros X, Y y Z con el mismo número de elementos y una cota d, encontrar k tripletas (x, y, z) en donde $x \in X$, $y \in Y$ y $z \in Z$, cada elemento se presenta en una sola tripleta y se cumple que $x + y + z = d$ para todas las tripletas.
- **Problema de suma de subconjuntos:** Dado un conjunto de números enteros decidir si existe un subconjunto no vacío cuya suma es cero.

Procesamiento de cadenas

- **Problema de la subsecuencia común más grande:** Dado un conjunto de secuencias, encontrar la subsecuencia más grande que sea común a todas las secuencias del conjunto (revisar [6] para más detalles).
- **Problema de corrección cadena a cadena:** Dadas dos cadenas distintas, determinar el número mínimo de ediciones que se deben aplicar a una cadena para cambiarla en la otra (revisar [7] para más detalles) .

Juegos

La resolución de los siguientes juegos:

- **Hundir la flota:** localizar la ubicación de todos los barcos de longitud $1 \times n$ del contrincante en una cuadrícula de dimensión $m \times m$.
- **Toros y vacas:** determinar el número de 4 dígitos oculto del contrincante mediante una serie de intentos en donde el contrincante deberá responder si es que en el intento hay dígitos coincidentes con el número y si hay dígitos que aparecen en el número en un orden distinto.
- **Sudoku generalizado:** dada una cuadrícula de 81×81 y un mínimo de 17 pistas iniciales (para más detalles en el número mínimo de pistas iniciales para un Sudoku válido ver [10]) llenar cada cuadro con dígitos de tal manera que cada subcuadrícula principal de 9×9 conste de los nueve dígitos sin repeticiones.
- **Tetris:** maximizar el número de filas limpiadas al jugar la siguiente pieza dada.

Además se puede revisar [9] para aprender más acerca de un problema de este tipo en ajedrez.

Lógica

- **Satisfacibilidad Booleana:** Dada una expresión booleana con variables booleanas y sin cuantificadores, encontrar la combinación de estados para las variables que devuelven un resultado afirmativo.

Optimización combinatoria

- **k-centro métrico:** Dadas n ciudades con distancias especificadas construir k almacenes en diferentes ciudades tal que se minimize la distancia máxima entre una ciudad y un almacén.
- **Problema de ruteo de vehículos con recolección y entrega:** Dada una cantidad de bienes que deben ser movidos de puntos de recolección a puntos de entrega, encontrar la ruta óptima para llevar esto a cabo (Se puede revisar [8] para más detalles acerca de este problema.).
- **Problema de la mochila:** Dada una mochila que tiene un límite en el peso que puede cargar, y ciertos objetos a los cuales está asociados un valor y un peso, se introducen estos objetos en la mochila. Maximizar el valor de los objetos dentro de la mochila sin exceder el peso máximo de ella.

El hecho de comprobar que $P = NP$, es decir encontrar un algoritmo polinomial para algún problema $NP - Completo$, se tendría, por la misma definición de $NP - Completo$, que esto significaría el encontrar un algoritmo polinomial para todos los problemas $NP - Completos$ y todos los problemas NP en general. Al encontrar que $P = NP$ no se estaría resolviendo solamente un problema matemático teórico, sino que se estaría dando una solución razonable a miles de otros problemas tanto prácticos como teóricos. Aquí yace la importancia del problema del milenio. Del problema del millón de dólares.

Referencias

[1] Cormen Thomas H., Leiserson Charles E., Rivest Ronald L., *Introduction to Algorithms*, The MIT Press, Cambridge, Massachusetts, Londres, Inglaterra. 1era Edición, 1990

[2] Cook Stephen, *The $PvsNP$ Problem*, Clay Mathematics Institute of Cambridge, Massachusetts (CMI), 2000

[3] Cook Stephen, *The Complexity of Theorem-Proving Procedures*, Universidad de Toronto, 1971

[4] Pérez Raposo Álvaro, *Lógica, conjuntos, relaciones y funciones*, Universidad Autónoma de San Luis Potosí, Universidad Politécnica de Madrid, 2010

[5] Garey Michael R , David S. Johnson, *Computers and Intractability; A Guide to the Theory of NP-Completeness*, W.H. Freeman and Company, San Francisco, 1979

[6] Maier David, *The Complexity of Some Problems on Subsequences and Supersequences*, (JACM) Volume 25 Issue 2, April 1978, 322-336

[7] Wagner Robert A. Fischer Michael J., *The String-to-String Correction Problem*, (JACM) Volume 21 Issue 1, Jan. 1974, 168-173

[8] Psaraftis, H.N., *Dynamic vehicle routing problems*, in: Vehicle Routing: Methods and Studies, eds. B. Golden and A. Assad (North-Holland, 1988).

[9] Fraenkel Aviezri S., Lichtenstein, *Computing a Perfect Strategy for $n \times n$ Chess Requires Time Exponential in n.* Academic Press, Inc., 1981

[10] MacGuire Gart, Tugemann Bastian, Civario Gilles, *There is no 16-Clue Sudoku: Solving the Sudoku Minimum Number of Clues Problem via Hitting Set Enumeration.* Cornell University Library, 2013