**COMPLEJIDAD DE UN ALGORITMO**

**Dr. FLABIO A. GUTIERREZ SEGURA**

1. **Introducción**

En los problemas computacionales se habla de problemas “sencillos” y “problemas difíciles”. Ejemplos de problemas sencillos: Ordenamiento de valores, multiplicación de matrices. Ejemplos de problemas difíciles, el TSP (Travelling Salesman Problem), problemas combinatorios.

En un sentido amplio, dado un problema y un dispositivo donde resolverlo, es necesario proporcionar un método preciso que lo resuelva. A tal método lo denominamos *algoritmo*.

Hay dos aspectos muy importantes de los algoritmos, como son su diseño y el estudio de su eficiencia.

El primero se refiere a la búsqueda de métodos o procedimientos, secuencias finitas de instrucciones adecuadas al dispositivo que disponemos, que permitan resolver el problema. Por otra parte, el segundo nos permite medir de alguna forma el coste (en tiempo y recursos) que consume un algoritmo para encontrar la solución y nos ofrece la posibilidad de comparar distintos algoritmos que resuelven un mismo problema.

Este módulo está dedicado al segundo de estos aspectos: la eficiencia.

En la computación los algoritmos son más importantes que los Lenguajes de Programación (LP); la solución de un problema haciendo uso de las computadoras requiere por una parte un algoritmo o método de resolución y por otra un programa o codificación del algoritmo en un LP. Ambos componentes tienen importancia; pero la del algoritmo es absolutamente indispensable; sabemos que un algoritmo es una secuencia de pasos para resolver un problema, sus características son:

* Independiente: del LP y de la máquina.
* Definido: de pasos claros y concretos.
* Finito: en el número de pasos que usará.
* Preciso: cada paso arroja un cálculo correcto.
* Recibe datos: debe poseer datos de entrada.
* Arroja información: debe arrojar información de salida.

Para un mismo problema existen muchas alternativas de solución y estas alternativas pueden diferir por:

* Número de pasos
* Estructuras

Ahora que sabemos que existen muchas alternativas de solución para un problema, debemos seleccionar el algoritmo más eficiente, el mejor conjunto de pasos, el que tarde menos en ejecutarse, que tenga menos líneas de código.

Esta selección puede ser ejecutada a simple vista con sólo observar la cantidad de líneas del programa, pero cuando el programa crece se requiere una medición más exacta y apropiada, para esto se realizan ciertas operaciones matemáticas que establecen la eficiencia teórica del programa, al estudio de estos casos se denomina **Complejidad Algorítmica.**

**2. Tiempo de ejecución**

Una medida que suele ser útil conocer es el tiempo de ejecución de un programa en función de *n*, lo que denominaremos *T(n).* Esta función se puede medir físicamente (ejecutando el programa, reloj en mano), o calcularse sobre el código contando instrucciones a ejecutar.

Dado que es fácil cambiar de LP y que la potencia de los ordenadores crece a un ritmo vertiginoso (en la actualidad, se duplica anualmente), intentaremos analizar los algoritmos con algún nivel de independencia de estos factores; es decir, en el algoritmo se contará las instrucciones a ejecutar.

Teóricamente *T*(*n*) debe indicar el número de instrucciones ejecutadas por un ordenador idealizado. Debemos buscar por tanto medidas simples y abstractas, independientes del ordenador a utilizar.

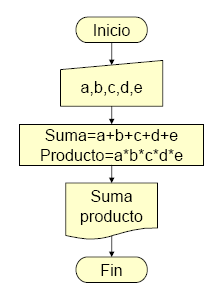
Operaciones Elementales (OE)

* Entradas/salidas
* Operaciones aritméticas básicas
* Asignaciones a variables de tipo predefinido
* Los saltos (llamadas a funciones y procedimientos, retorno desde ellos, etc.),
* Comparaciones lógicas
* Acceso a estructuras indexadas básicas, como son los vectores y matrices.

Cada una de ellas contabilizará como 1 OE.

**Ejemplo 1.**

Hallar el T(n) para el algoritmo que 5 números por teclado y calcule la suma y el producto de los números.



**Las**

5 entradas = 5 0E

4 sumas = 4 OE

1 asignación = 1 OE

4 multiplicaciones = 4 OE

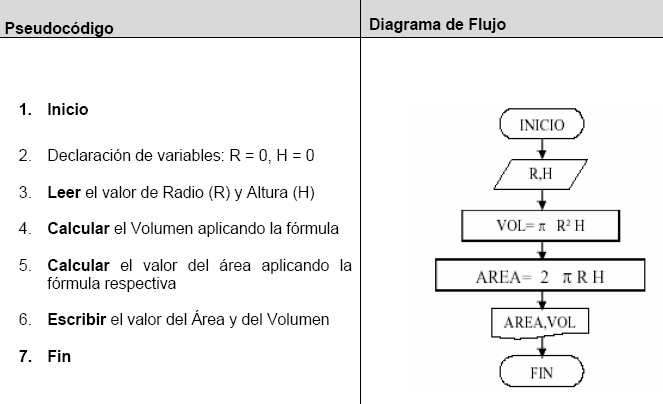
1 asignación = 1 OE

2 salidas = 2 OE

T(n) =5+ 4+1+4+1 +2 =17

**Ejemplo 2.**

Hallar el T(n) para el algoritmo que permita determinar el área y volumen de un cilindro dado su radio (R) y altura (H).



T(n) = 2 OE + 4 OE + 4 OE + 2 OE = 10

También es importante hacer notar que el comportamiento de un algoritmo puede cambiar notablemente para diferentes entradas (por ejemplo, lo ordenados que se encuentren ya los datos a ordenar). De hecho, para muchos programas el tiempo de ejecución es en realidad una función de la entrada específica, y no sólo del tamaño de ésta.

Así suelen estudiarse tres casos para un mismo algoritmo: *caso peor*, *caso mejor* y *caso medio*.

El caso mejor corresponde a la traza (secuencia de sentencias) del algoritmo que realiza menos instrucciones. Análogamente, el caso peor corresponde a la traza del algoritmo que realiza más instrucciones. Respecto al caso medio, corresponde a la traza del algoritmo que realiza un número de instrucciones igual a la esperanza matemática de la variable aleatoria definida por todas las posibles trazas del algoritmo para un tamaño de la entrada dado, con las probabilidades de que éstas ocurran para esa entrada.

**Ejemplo 3.**

Hallar el T(n) para el algoritmo que cambia el signo de un número, únicamente en caso que sea negativo.

Leer x (1 OE)

Si x <0 (1 OE)

y=-1\*x (2 OE)

Else

y=x (1 OE)

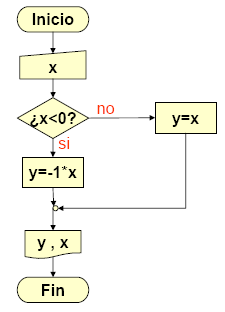
End

Escribir y (1 OE)

Escribir x (1 OE)

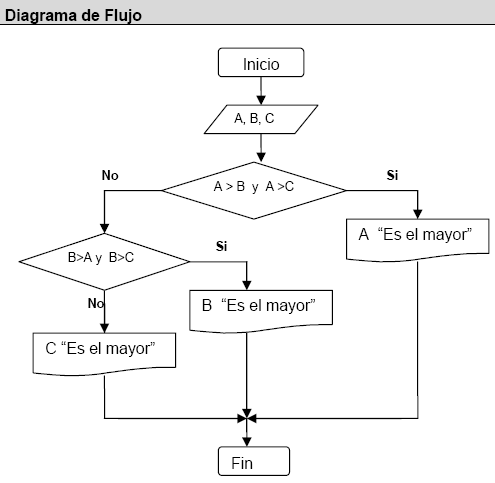
Caso mejor: T(n) = 1 OE + 1 OE + 1 OE + 1 OE + 1 OE = 5

Caso peor : T(n) = 1 OE + 1 OE +2 OE + 1 OE + 1 OE = 6



**Ejemplo 4.**

Hallar el T(n) para el algoritmo que permita ingresar tres valores diferentes y almacenarlos en las variables A, B y C respectivamente. El algoritmo debe imprimir cual es el mayor.



**Inicio**

**Leer A, B, C** 3 OE

**Si** A > B y A > C **Entonces 3 OE**

**Escribir** A “Es el mayor” 1 OE

**Sino**

**Si** B > A y B > C **Entonces 3 OE**

**Escribir** B “Es el mayor” 1 OE

**Sino**

**Escribir** C “Es el mayor” 1 OE

**Fin Si**

**Fin Si**

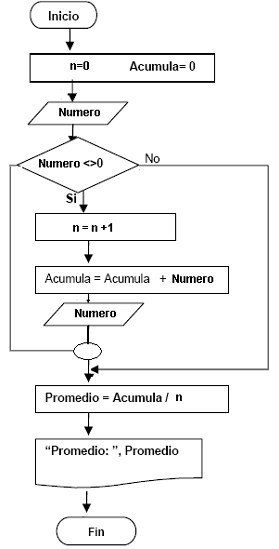
**Fin**

Caso mejor: T(n) = 3 OE + 3 OE + 1 OE = 7

Caso peor : T(n) = 3 OE + 3 OE + 3 OE + 1 OE = 10

**Ejemplo 5.**

Hallar el T(n) para el algoritmo que permita calcular Promedio de Notas. El algoritmo finaliza cuando Numero = 0.



**Inicio**

n= 0, Acumula= 0 2 OE

Leer Numero 1 OE

**Mientras** Numero <> 0 hacer 1 OE

n = n + 1 2 OE

Acumula = Acumula + Numero 2 OE

Leer Numero 1 OE

**Fin Mientras**

Promedio = Acumula/n 2 OE

Imprimir Promedio 1 OE

**Fin**

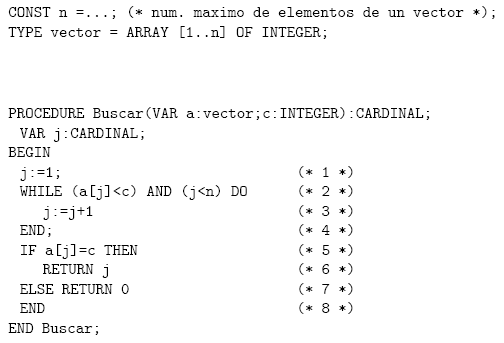
Caso mejor: T(n) = 2 + 1 + 1 + 2 + 1 = 7

Caso peor: T(n) =

T(n) = 3 + (6 n + 1) + 3= 6n+7

**Ejemplo 6.**

**Hallar el T(n) para el siguiente algoritmo**



Para determinar el tiempo de ejecución, calcularemos primero el número de operaciones elementales (OE) que se realizan:

– En la línea (1) se ejecuta 1 OE (una asignación).

– En la línea (2) se efectúa la condición del bucle, con un total de 4 OE (dos comparaciones, un acceso al vector, y un *AND*).

– La línea (3) está compuesta por un incremento y una asignación (2 OE).

– La línea (5) está formada por una condición y un acceso al vector (2 OE).

– La línea (6) contiene un *RETURN* (1 OE) si la condición se cumple.

– La línea (7) contiene un *RETURN* (1 OE), cuando la condición del *IF* anterior es falsa.

* En el *caso mejor:*

Para el algoritmo, se efectuará la línea (1) y de la línea (2) sólo la primera mitad de la condición, que supone 2 OE (suponemos que las expresiones se evalúan de izquierda a derecha, y con “cortocircuito”, es decir, una expresión lógica deja de ser evaluada en el momento que se conoce su valor, aunque no hayan sido evaluados todos sus términos). Tras ellas la función acaba ejecutando las líneas (5) a (7). En consecuencia, T(n)=1+2+3=6.

* En el *caso peor*:

Se efectúa la línea (1), el bucle se repite n–1 veces hasta que se cumple la segunda condición, después se efectúa la condición de la línea (5) y la función acaba al ejecutarse la línea (7). Cada iteración del bucle está compuesta por las líneas (2) y (3), junto con una ejecución adicional de la línea (2) que es la que ocasiona la salida del bucle. Por tanto



* En el *caso medio*:

El bucle se ejecutará un número de veces entre 0 y *n*–1, y vamos a suponer que cada una de ellas tiene la misma probabilidad de suceder.

Como existen *n* posibilidades (puede que el número buscado no esté) suponemos a priori que son equiprobables y por tanto cada una tendrá una probabilidad asociada de 1/*n*. Con esto, el número medio de veces que se efectuará el bucle es de



Tenemos pues que



Es importante observar que no es necesario conocer el propósito del algoritmo para analizar su tiempo de ejecución y determinar sus casos mejor, peor y medio, sino que basta con estudiar su código.

# 3. Orden de Complejidad

En el análisis de algoritmos se considera usualmente el caso peor, si bien a veces conviene analizar igualmente el caso mejor y hacer alguna estimación sobre un caso promedio. Para independizarse de factores coyunturales tales como el lenguaje de programación, la habilidad del codificador, la máquina soporte, etc. se suele trabajar con un cálculo asintótico que indica cómo se comporta el algoritmo para datos muy grandes.

Es necesario acotar de alguna forma la diferencia que se puede producir entre distintas implementaciones de un mismo algoritmo, ya sea del mismo código ejecutado por dos máquinas de distinta velocidad, como de dos códigos que implementen el mismo método. Esta diferencia es la que acota el siguiente principio:

**Principio de Invarianza**

Dado un algoritmo y dos implementaciones suyas *I1* e *I2*, que tardan *T1(n)* y *T*2(*n*) segundos respectivamente, el *Principio de Invarianza* afirma que existe una constante real *c* > 0 y un número natural *n*0 tales que para todo *n* ≥ *n0*  se verifica que *T*1(*n*) ≤ *cT*2(*n*).

Es decir, el tiempo de ejecución de dos implementaciones distintas de un algoritmo dado no va a diferir más que en una constante multiplicativa.

Una familia de funciones que comparten un mismo comportamiento asintótico será llamada un *Orden de Complejidad*. Estas familias se designan con *O( )*, existiendo una infinidad de ellos.

* La familia *O(f(n))* define un Orden de Complejidad. Elegiremos como representante de este Orden de Complejidad a la función *f(n)* más sencilla perteneciente a esta familia.
* Las funciones de complejidad algorítmica más habituales en las cuales el único factor del que dependen es el tamaño de la muestra de entrada n, ordenadas de mayor a menor eficiencia son:

|  |  |
| --- | --- |
| *O(1)* | *Orden constante* |
| *O(log n)* | *Orden logarítmico* |
| *O(n)* | *Orden lineal* |
| *O(n log n)* | *Orden cuasi-lineal* |
| *O(n2)* | *Orden cuadrático* |
| *O(n3)* | *Orden cúbico* |
| *O(na)* | *Orden polinomico* |
| *O(2n)* | *Orden exponencial* |

* Es más, se puede identificar una jerarquía de órdenes de complejidad que coincide con el orden de la tabla anterior; jerarquía en el sentido de que cada orden de complejidad superior tiene a los inferiores como subconjuntos. Si un algoritmo **A** se puede demostrar de un cierto orden *O1*, es cierto que también pertenece a todos los órdenes superiores (la relación de orden cota superior de' es transitiva); pero en la práctica lo útil es encontrar la "menor cota superior", es decir el menor orden de complejidad que lo cubra.
* *O(1):* Complejidad constante. Cuando las instrucciones se ejecutan una vez.
* *O(log n)*: Complejidad logarítmica. Esta suele aparecer en determinados algoritmos con iteración o recursión no estructural, ejemplo la búsqueda binaria.
* *O(n)*: Complejidad lineal. Es una complejidad buena y también muy usual. Aparece en la evaluación de bucles simples siempre que la complejidad de las instrucciones interiores sea constante.
* *O(n log n)*: Complejidad cuasi-lineal. Se encuentra en algoritmos de tipo divide y vencerás como por ejemplo en el método de ordenación quicksort y se considera una buena complejidad. Si *n* se duplica, el tiempo de ejecución es ligeramente mayor del doble.
* *O(n2)*: Complejidad cuadrática. Aparece en bucles o ciclos doblemente anidados. Si *n* se duplica, el tiempo de ejecución aumenta cuatro veces.
* *O(n3)*: Complejidad cúbica. Suele darse en bucles con triple anidación. Si *n* se duplica, el tiempo de ejecución se multiplica por ocho. Para un valor grande de *n* empieza a crecer dramáticamente.
* *O(na)*: Complejidad polinómica (*a* > 3). Si *a* crece, la complejidad del programa es bastante mala.
* *O(2n)*: Complejidad exponencial. No suelen ser muy útiles en la práctica por el elevadísimo tiempo de ejecución. Se dan en subprogramas recursivos que contengan dos o más llamadas internas.

***Algoritmos Polinomiales:*** Aquellos que son proporcionales a *na*. Son en general factibles o aplicables, los problemas basados en estos algoritmos son solucionables.

***Algoritmos Exponenciales***: Aquellos que son proporcionales a *kn*. En general no son factibles salvo un tamaño de entrada *n* exageradamente pequeño, pero generalmente pertenecen a un universo de problemas de los cuales el cómputo se hace imposible.

# 4.- Impacto Práctico

**Ejemplo 7.**

Contamos con una computadora capaz de procesar datos en 10-4 seg. En esta computadora se ejecuta un algoritmo que lee registros de una base de datos, dicho algoritmo tiene una complejidad exponencial 2*n, ¿Cuánto tiempo se tardará en procesar una entrada n de datos?*

|  |  |  |
| --- | --- | --- |
| ***n*** | **Tiempo** | **TIEMPO** |
| 10 | 210(0.0001) = 0.1024 seg | ≈ 1 décima de segundo |
| 20 | 220(0.0001) = 104.8576 seg | ≈ 2 minutos |
| 30 | 230(0.0001) = 107374.1824 seg = 1.242757 dias | > 1 día |
| 40 | 240(0.0001) = 109951162.776 seg= 3.53495251 años | > 3 años |
| 50 | 250(0.0001) =1.125899906842624 × 10^11seg. = 3619.791367 años | ≈ 3 3620 años |
| 100 | 2100(0.0001) = 1.267650600228229401496703205376 × 10^26 seg.  = 4.0755227630794412342357999144032921810699588477366255 × 10^15 milenios | = 4,075,522,763,079,441 milenios |

Ahora se tiene la misma computadora capaz de procesar datos en 10-4 seg. Se ejecuta un algoritmo que hace el mismo trabajo antes citado, pero este algoritmo tiene una complejidad cúbica *n3.*

*¿Cuánto tiempo se tardará en procesar una entrada* ***n*** *de datos?*

|  |  |  |
| --- | --- | --- |
| ***n*** | **TIEMPO** | **TIEMPO** |
| 10 | 103 (0.0001) = 0.1 seg. | = 1 décima de segundo |
| 20 | 203 (0.0001) =0.8 seg. | = 8 décimas de segundo |
| 100 | 1003 (0.0001) = 100 seg. | = 1.7 minutos |
| 200 | 2003 (0.0001) = 800 seg | = 13.3 minutos |
| 1000 | 10003 (0.0001) = 100000 seg = 1.1574 dias | ≈ 1 día |

**Ejemplo 8:**

Sea un problema que sabemos resolver con algoritmos de diferentes complejidades. Para compararlos entre sí, supongamos que todos ellos requieren 1 hora de ordenador para resolver un problema de tamaño n=100.

¿Qué ocurre si disponemos del doble de tiempo? Nótese que esto es lo mismo que disponer del mismo tiempo en un ordenador el doble de potente, y que el ritmo actual de progreso del hardware es exactamente ese:

"duplicación anual del número de instrucciones por segundo".

¿Qué ocurre si queremos resolver un problema de tamaño 2n?

|  |  |  |  |
| --- | --- | --- | --- |
| **O(f(n))** | **n=100** | **t=2h** | **n=200** |
| **log n** | 1 h | 10000 | 1.15 h |
| **n** | 1 h | 200 | 2 h |
| **n log n** | 1 h | 199 | 2.30 h |
| **n2** | 1 h | 141 | 4 h |
| **n3** | 1 h | 126 | 8 h |
| **2n** | 1 h | 101 | 1030 h |

Los algoritmos de complejidad O(n) y O(n log n) son los que muestran un comportamiento más "natural": prácticamente a doble de tiempo, doble de datos procesables.

Los algoritmos de complejidad logarítmica son un descubrimiento fenomenal, pues en el doble de tiempo permiten atacar problemas notablemente mayores, y para resolver un problema el doble de grande sólo hace falta un poco más de tiempo (ni mucho menos el doble).

Los algoritmos de tipo polinómico no son una maravilla, y se enfrentan con dificultad a problemas de tamaño creciente. La práctica viene a decirnos que son el límite de lo "tratable".

Sobre la tratabilidad de los algoritmos de complejidad polinómica habría mucho que hablar, y a veces semejante calificativo es puro eufemismo. Mientras complejidades del orden O(n2) y O(n3) suelen ser efectivamente abordables, prácticamente nadie acepta algoritmos de orden O(n100), por muy polinómicos que sean. La frontera es imprecisa.

Cualquier algoritmo por encima de una complejidad polinómica se dice "intratable" y sólo será aplicable a problemas ridículamente pequeños.

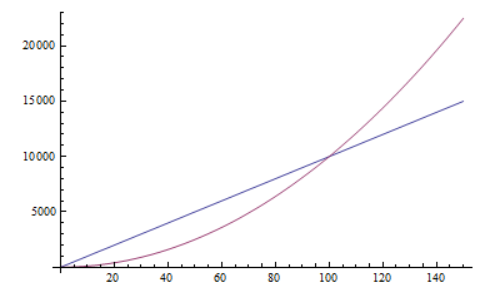
A la vista de lo anterior se comprende que se busquen algoritmos de complejidad lineal. Es un golpe de suerte encontrar algo de complejidad logarítmica. Si se encuentran soluciones polinomiales, se puede vivir con ellas; pero ante soluciones de complejidad exponencial, más vale seguir buscando.

**Ejemplo 9:**

Si disponemos de dos algoritmos para el mismo problema, con tiempos de ejecución respectivos:

|  |  |  |
| --- | --- | --- |
| **algoritmo** | **tiempo** | **complejidad** |
| f | 100 n | O(n) |
| g | n2 | O(n2) |

Asintóticamente, "f" es mejor algoritmo que "g"; pero esto es cierto a partir de N > 100.

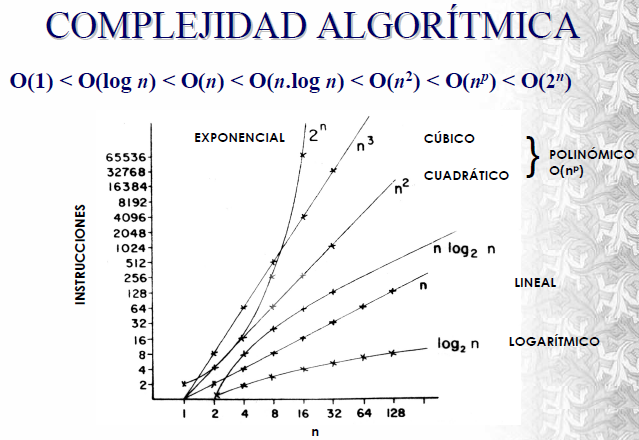


Si nuestro problema no va a tratar jamás problemas de tamaño mayor que 100, es mejor solución usar el algoritmo "g".

El ejemplo anterior muestra que las constantes que aparecen en las fórmulas para T(n), y que desaparecen al calcular las funciones de complejidad, pueden ser decisivas desde el punto de vista de ingeniería. Pueden darse incluso ejemplos más dramáticos:

|  |  |  |
| --- | --- | --- |
| **algoritmo** | **tiempo** | **complejidad** |
| f | n | O(n) |
| g | 100 n | O(n) |

Aun siendo dos algoritmos con idéntico *comportamiento* asintótico, es obvio que el algoritmo “f" es siempre 100 veces más el " rápido que g" y candidato primero a ser utilizado.



# 5. Problemas P, NP y NP-completos

Hasta aquí hemos venido hablando de algoritmos. Cuando nos enfrentamos a un problema concreto, habrá una serie de algoritmos aplicables. Se suele decir que el orden de complejidad de un problema es el del mejor algoritmo que se conozca para resolverlo. Así se clasifican los problemas, y los estudios sobre algoritmos se aplican a la realidad.

Estos estudios han llevado a la constatación de que existen problemas muy difíciles, problemas que desafían la utilización de los ordenadores para resolverlos.

**Clase P**

Los algoritmos de complejidad polinómica se dice que son tratables en el sentido de que suelen ser abordables en la práctica. Los problemas para los que se conocen algoritmos con esta complejidad se dice que forman la clase P. Aquellos problemas para los que la mejor solución que se conoce es de complejidad superior a la polinómica, se dice que son problemas intratables. Seria muy interesante encontrar alguna solución polinómica (o mejor) que permitiera abordarlos.

Ejemplos de algoritmos de complejidad P: Búsqueda Binaria, método de ordenación quicksort, multiplicación de matrices, calcular el determinante de una matriz, algoritmos de programación lineal (método simplex).

**Clase NP**

Algunos de estos problemas intratables pueden caracterizarse por el curioso hecho de que puede aplicarse un algoritmo polinómico para comprobar si una posible solución es válida o no. Esta característica lleva a un método de resolución no determinista consistente en aplicar heurísticos para obtener soluciones hipotéticas que se van desestimando (o aceptando) a ritmo polinómico. Los problemas de esta clase se denominan NP (la N de no-deterministas y la P de polinómicos).

Ejemplos de algoritmos de complejidad NP: el TSP (Travelling Salesman Problem), problemas combinatorios, algoritmo para hallar la raíz cuadrada, Programación Lineal Entera

**Clase NP-completos**

Se conoce una amplia variedad de problemas de tipo NP, de los cuales destacan algunos de ellos de extrema complejidad. Podemos decir que algunos problemas se hayan en la "frontera externa" de la clase NP. Son problemas NP, y son los peores problemas posibles de clase NP. Estos problemas se caracterizan por ser todos "iguales" en el sentido de que si se descubriera una solución P para alguno de ellos, esta solución sería fácilmente aplicable a todos ellos. Actualmente hay un premio de prestigio equivalente al Nobel reservado para el que descubra semejante solución ... ¡y se duda seriamente de que alguien lo consiga!

Es más, si se descubriera una solución para los problemas NP-completos, esta sería aplicable a todos los problemas NP y, por tanto, la clase NP desaparecería del mundo científico al carecerse de problemas de ese tipo. Realmente, tras años de búsqueda exhaustiva de dicha solución, es hecho ampliamente aceptado que no debe existir, aunque nadie ha demostrado, todavía, la imposibilidad de su existencia.

**Ejemplo 10:**

Se pide comprobar si un número determinado X es la raíz cuadrada de Z

Alternativas

1. Calcular la raíz de Z y comparando con X (proceso lento y engorroso)
2. Elevar al cuadrado a X y comparar con Z (simple multiplicación X·X)

La complejidad de la función “elevar al cuadrado” (Clase P) es más simple que calcular la raíz cuadrada.

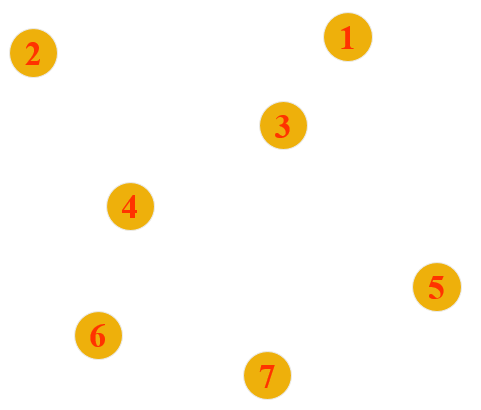
En algunos problemas comprobar la solución es más eficiente que calcularla.

**Esta idea es el fundamento para el desarrollo de las heurísticas y metaheuristicas.**

La **clase de complejidad NP** contiene problemas que **no pueden resolverse** en un tiempo polinómico. Cuando se dice que un algoritmo no puede obtener una solución a un problema en tiempo polinómico siempre se intenta buscar otro procedimiento que lo consiga mejorar.

**Ejemplo 11:** El problema del viajante de comercio

Travelling Salesman Problem (TSP)



Se tiene una red de nodos, que pueden ser ciudades.

**Se parte de un lugar inicial, y debe recorrerse todos sin pasar más de una vez por cada lugar, volviendo al lugar inicial, y con una distancia total recorrida mínima**

Para cada arco, se tiene un valor Cij, que indica la distancia o el costo de ir del nodo i al nodo j

Sea n= número de ciudades

Para n=7, hay 7! = 5040 posibles rutas ***(***permutaciones) posibles.

Para n=10, hay 10! = 3 628 800 rutas posibles.

Para n=12, hay 12! = 479 001 600 rutas posibles.

Para n=20 hay 2432 9020 0817 6640 000 rutas posibles.

Explorando 10000 permutaciones por segundo, una búsqueda exhaustiva demandaría un tiempo estimado de:

|  |  |  |
| --- | --- | --- |
| n | Rutas | Tiempo estimado |
| 7 | 5040 | 0.504 seg |
| 10 | 3 628 800 | 362.88 seg. aprox 6.048 minutos |
| 12 | 479 001 600 | 47 900.2 seg= 798.333 minutos  13.3055 horas |
| 20 | 2432 9020 0817 6640 000 | 243290200817664 seg.  4.05484xE10 minutos  7.82183xE6 años |

Aplicaciones

* El camino más corto entre varios puntos,
* Un plan de mínimo coste para repartir mercancías a clientes,
* Una asignación óptima de trabajadores a tareas a realizar,
* Una secuencia óptima de proceso de trabajos en una cadena de producción,
* Una distribución de tripulaciones de aviones con mínimo coste,
* El mejor enrutamiento de un paquete de datos en Internet

# 6.- Conclusión

* Se puede concluir, que solo un algoritmo eficiente, con un orden de complejidad bajo, puede tratar grandes volumen de datos, se razona que un algoritmo es:
  + - Muy eficiente si su complejidad es de orden *log* *n*
    - Eficiente si su complejidad es de orden *na*
    - Ineficiente si su complejidad es de orden *2n*
* Se considera que un problema es tratable si existe un algoritmo que lo resuelve con complejidad menor que 2*n*, y que es intratable o desprovisto de solución en caso contrario.