INTRODUCCIÓN AL ANÁLISIS DE EFICIENCIA

Versión 20160407

ÍNDICE DE CONTENIDOS

PRESENTACIÓN

El presente documento contiene una introducción básica al análisis de eficiencia de los algoritmos desde la perspectiva de las ciencias de la computación. Esto, que es una parte importante en el diseño de un programa, a veces se deja un poco apartado por el cálculo matemático que lleva asociado.

Sin embargo, creo que es indispensable que cualquier persona que dedique a crear programas, es decir, que se dedica a crear o utilizar algoritmos ya creados, debe al menos conocer qué es el análisis de eficiencia y tener presente que en ocasiones es tan importante como obtener un algoritmo correcto, pues puede suponer el éxito o el fracaso del producto.

Para entender todo el contenido, recomendaría tener unos conocimientos mínimos de cálculo matemático (funciones con una variable, su representación, etc) así como de programación en cualquier lenguaje de programación en el paradigma imperativo u orientado a objetos (C,C++, Pascal, Java, ….) , así como, para el último apartado, conocer lo que es la recursividad y su aplicación a la programación (que es de plena aplicación en el paradigma funcional).

El autor es consciente de que el tratamiento que da al análisis, al ser básico, pretende ser más conceptual que formal. Para el segundo enfoque, existen otros documentos más adecuados que profundizan en el tema en cuestión, pudiendo partir de algunos de los indicados en la bibliografía.

El lector podría pensar, al terminar de leer el documento, que tiene pocos ejemplos. Esto es porque en esta primera versión se ha pretendido que sea una guía de conceptos que ayude a comprender el análisis de eficiencia.

Por último, animo al lector a que me proponga mejoras y posibles erratas a mi dirección de correo actual robertofl@aconute.es

Aviso de derechos de autor

CAPÍTULO 1.- INTRODUCCIÓN

OBJETIVOS

- Comprender la naturaleza de los algoritmos
- Distinguir entre algoritmo y programa
- Comprender los pasos en la creación de un algoritmo
- De los pasos anteriores, comprender en qué consiste el análisis de eficiencia
- Entender que el análisis de eficiencia es útil para elegir un algoritmo u otro.

PREREQUISITOS

- Representación de funciones de una variable.
- Conocimientos básicos de programación.

CAPÍTULO 1.- INTRODUCCIÓN

1.1.- ALGORITMOS

El término algoritmo fue acuñado por el matemático persa, del siglo IX, Mohamed Ben Musa, de sobrenombre Al-Jwârizmî. Podemos definir un algoritmo como un conjunto ordenado (secuencia) y finito de operaciones (acciones), que permiten hallar la solución (o soluciones) de un problema.

Para que esta **secuencia de acciones** se pueda considerar algoritmo, cada una de estas acciones **no debe permitir una interpretación ambigua**; han de ser exactas. Si esto no fuese así, podría darse el caso de no encontrar la solución aun cuando exista, o que la solución encontrada no fuese la correcta.

Ejemplo 1.1

Supongamos que estamos cocinando una receta nueva de cocina. Una de las acciones que nos dice la receta es añadir la sal; podríamos tener dos opciones:

a) Añadir 5gr de sal.- Es una acción exacta, luego si siempre añadimos la misma cantidad, el plato debería estar siempre igual de salado (o de soso).

b) Añadir una pizca de sal.- Esta es una acción subjetiva, ya que para cada cocinero una pizca de sal es una cantidad distinta y así el plato obtenido no será siempre el mismo.

--

Las acciones las ejecuta un **procesador.** Aquí procesador lo es en el sentido genérico, es indiferente si es una persona o si es una máquina.

En términos reales, cuando una persona ha de resolver un problema, lo que hace es planificar una *secuencia finita de acciones,* a realizar sobre un objeto para llegar a alguna solución de dicho problema.

Normalmente, cuando se trata de un problema diario (cuya característica es que no se suele repetir o, como mucho, en pocas ocasiones), buscamos esa secuencia de manera intuitiva, no solemos complicarnos: buscamos secuencias de acciones sencillas de aplicar. Probablemente, la suma del tiempo necesario en buscar una secuencia de acciones más elaborada (que nos permita solucionar el problema en menos tiempo) y el tiempo en realizar las acciones posiblemente no será rentable, ya que será superior al tiempo de la secuencia de acciones inicial. Sin embargo, si se trata de un problema repetitivo, sí nos puede interesar encontrar un mejor algoritmo, ya que a la larga nos permitirá ahorrar tiempo en el proceso (ver el ejemplo final de este capítulo).

La secuencia de acciones que hemos elegido se tiene que describir de alguna manera. Cuando la secuencia la va a realizar una persona, esta descripción suele hacerse en lenguaje natural. Una de las características del lenguaje natural es su alto grado de abstracción, luego en la descripción se presentarán acciones que una persona identifica como atómicas (indivisibles), pero que en realidad son acciones muy complejas.

Cuando tenemos la necesidad de hacer una descripción para una máquina, la situación es distinta. Las máquinas solo entienden acciones muy elementales, que además no admiten ningún tipo de matices. Cada tipo de máquina solo entiende un lenguaje, que suele ser formal y estricto (con poca capacidad expresiva, restricciones y peculiaridades). Así un algoritmo para una máquina hay que **implementarlo en un programa,** que no es otra cosa que una secuencia de acciones elementales (un algoritmo) descritas en el lenguaje que entiende esa máquina.

Así pues, para un problema podemos tener varios algoritmos que lo solucionan, y para cada uno de estos algoritmos, podremos tener varias implementaciones, cada una en el lenguaje que entienda cada procesador.

Ejemplo 1.2

Hace calor. Una posible solución es abrir la ventana. Una posible descripción de la secuencia de acciones podría ser simplemente: abrir la ventana. Esta acción, que para una persona sería atómica, implica una serie de acciones más elementales (como son las que entendería un robot, por ejemplo): detectar una temperatura, si la temperatura es

mayor a 24ºC moverse hasta la ventana, identificar el mecanismo de apertura, accionar el mecanismo, abrir la ventana y volver al punto de partida. Para el robot, hay que describir este algoritmo en el lenguaje formal que entienda, teniendo que pormenorizar cada acción (para moverse hasta la ventana tiene que sortear obstáculos, por ejemplo).
--

A partir de aquí, daremos por supuesto que el procesador va a ser un computador (una máquina). Su característica más relevante es que es capaz de procesar millones de acciones por segundo, y que además los trabajos que realiza son repetitivos. El objeto que procesa un computador son datos, llamados de entrada, que una vez procesados se convierten en datos de salida, que a su vez constituyen la solución (o soluciones) al problema.

Es necesarios matizar la relación entre datos de entrada y tareas repetitivas. Si el conjunto de datos de entrada fuese siempre el mismo, el resultado también sería el mismo: en este caso bastaría con procesar una única vez los datos de entrada y guardar directamente el resultado. Luego, el diseño de mejores algoritmos es interesante cuando se quiere encontrar la solución o soluciones de un mismo problema, que se da de forma repetida y sobre conjuntos distintos de datos de entrada.

1.2.- ALGORITMIA

Hemos dicho que podemos tener algoritmos sencillos, intuitivos y algoritmos más elaborados, que tardan menos en procesar la secuencia de acciones. Es decir, para un mismo problema podemos tener varios algoritmos distintos[1], pero ¿cuál es el bueno? Esto es una elección que depende de **cómo** y **cuándo** se vaya a utilizar para resolver ese problema.

En términos generales, la algoritmia se asocia al cálculo matemático, como ciencia del cálculo aritmético y algebraico. Un algoritmo que se procesa por un computador, no es más que eso, un cálculo aritmético, y por tanto es correcto utilizar esta ciencia para:

[1] Se ha dado por supuesto que hay uno o varios algoritmos que solucionan un problema. Hay muchos problemas para los que todavía no se ha encontrado un algoritmo que los solucione, como se muestra un poco más adelante.

- Diseñar nuevos algoritmos. La algoritmia nos va a ofrecer técnicas generales para la resolución de problemas, clasificando las posibles secuencias que los solucionan en familias de resolución de problemas, llamados **esquemas algorítmicos** (en problemas complejos)**.** La elección de uno de estos esquemas depende tanto del problema (un mismo problema se puede solucionar con varios esquemas distintos) como del uso que se prevé para el algoritmos que estamos diseñando.

- Estudiar de manera formal los distintos algoritmos (existentes o el nuevo diseñado) que solucionan un problema, analizando sus posibilidades, sus aplicaciones de uso potenciales y las limitaciones del computador en el que hay que implementarlo.

1.3.- CARACTERISTICAS DE UN ALGORTIMO

- **Eficacia.-** Un algoritmo es eficaz si es capaz de encontrar al menos una solución correcta al problema.
- **Corrección.-** Un algoritmo es correcto si, para cada dato de entrada, este termina su ejecución con la resolución del problema (encuentra la solución buscada). Se dice que resuelve el problema.
- **Eficiencia.-** Un algoritmo es más eficiente que otro si consume menos recursos[2] (ya sea de tiempo o de almacenamiento en memoria). Se mide como *coste computacional.*

Así pues, utilizaremos la algoritmia para diseñar algoritmos **EFICACES, CORRECTOS** y **EFICIENTES.**

1.4.- LIMITES DE LOS ALGORITMOS

Existen problemas para los que, con la definición dada para el mismo, no sería posible diseñar ningún algoritmo. Por ello, se permite flexibilizar la definición para poder añadir factores externos al comportamiento del mismo.

[2] Los recursos a consumir son tiempo de ejecución o consumo de memoria que , como se verá, son antagónicos.

- Algoritmos probabilistas.- Son aquellos que, para encontrar la solución, dependen de algún parámetro externo de probabilidad controlada. La ejecución de este tipo de algoritmo probablemente no dará la misma solución si se ejecuta dos veces seguidas. Hay dos tipos de algoritmos probabilistas:

    - Los que usan cálculo estadístico, con sus márgenes de confianza; tenemos una certeza relativa de que la solución es correcta.

    - Los que usan algún parámetro para ayudarse en el camino hacia la solución correcta (se utilizan cuando el tiempo para determinar el camino óptimo a la solución es prohibitivo).

- Algoritmos aproximados.- Un ejemplo claro es como damos un valor cuando este es infinito. En estos algoritmos se admite un margen de error para la solución.
  Un ejemplo de esto es la división 2/3 = 0,6666666…. En el algoritmo podemos decir que admitimos un error < 10^{-2}, por lo que el valor 0,66 (por truncamiento) es válido; también lo sería el valor 0,67 (por redondeo en exceso).

- Algoritmos heurísticos.- Existen problemas para los cuales los algoritmos que los solucionan no son viables en la práctica. En estos casos se extrae información adicional del problema, para posteriormente, durante la ejecución del algoritmo, utilizarla para guiar al algoritmo en la búsqueda de la solución. En estos casos se encuentra una solución, aunque no la más óptima. En estos casos el error introducido no se puede controlar, aunque quizás si estimar (en esta categoría también podríamos incluir los algoritmos probabilistas del segundo tipo antes mencionados).

1.5.- ESQUEMAS ALGORITMICOS

Hemos indicado que, para un problema concreto, se pueden dar varios algoritmos distintos y que la algoritmia nos ofrece técnicas para clasificar estos algoritmos dentro de alguna familia de resolución, llamados esquemas algorítmicos.

Un esquema algorítmico es una abstracción de control (es una plantilla): una guía de secuencias de acciones, caracterizada por unas funciones-tipo, y que son estas las que hay que particularizar para el problema concreto. Es decir, para implementar el algoritmo, este no se reescribe por completo, sino que se implementa el esquema donde sólo se definen estas funciones según el problema a resolver (esto facilita la modularidad).

Una distinción que podemos hacer entre **un esquema** algorítmico y un algoritmo, es que el primero **no se puede analizar en términos de coste computacional** (realmente no es una secuencia de acciones), mientras que **el algoritmo sí**.

Podemos clasificar los esquemas en tres grandes familias:

- Algoritmos voraces
- Algoritmos basados en "divide y vencerás"
- Algoritmos de exploración en grafos.

1.6.- SOLUCIONANDO UN PROBLEMA

Para el diseño de un algoritmo debemos realizar un proceso metódico y formal, utilizando principalmente los esquemas algorítmicos antes mencionados. Las fases del diseño serán:

- Abstracción
- Diseño
- Especificación y análisis de eficacia
- Análisis de eficiencia
- Implementación en el lenguaje elegido.

1.6.1 Abstracción

Para el diseño de un algoritmo lo primero que tenemos que saber es cual es la o las soluciones del problema. Este se debe analizar para extraer las características más relevantes del mismo.

1.6.2. Diseño

Si hemos identificado bien las características del problema, lo más probable es que podamos identificarlo con alguno de los **esquemas algorítmicos.** Si hay más de un esquema aplicable, habrá que elegir el que sea más fácil de implementar y que tenga un menor coste computacional.

Hay que hacer hincapié en que esta es la secuencia: análisis – elección de esquema, y no al contrario, ya que una elección esquema a priori puede llevar a un callejón sin salida o a que el esquema utilizado no sea el más adecuado. Por otro lado, elegir el esquema por eliminación de los otros, puede significar que no se extrajeron todas las propiedades relevantes, es decir, no se analizó adecuadamente el problema.

Una vez que hemos identificado el esquema que vamos a implementar, el siguiente paso es determinar el **dominio de definición** del problema, es decir, definir todos los casos posibles de datos de entrada para este algoritmo. Cada uno de estos casos es lo que se llama un **ejemplar** del problema (otros autores lo llaman instancia). Ahora, ya podemos particularizar las funciones tipo del esquema elegido, ajustándolas a los detalles específicos del problema.

Un detalle importante a la hora de diseñar el algoritmo es la elección de una estructura para los datos. La elección de una u otra estructura de datos puede ser relevante para la eficiencia del algoritmo, luego su elección es tan importante como la del esquema.

Si se han cumplido todos estos pasos, implementar el algoritmo se reduce a elegir el esquema como cuerpo del programa y completar las funciones tipo con sus particularidades, dependiendo de las estructuras de datos elegidas.

1.6.3. Especificación, corrección y análisis de eficacia

La especificación no es más que la formalización de cómo queremos resolver el problema. ¿Para qué? Hemos dicho anteriormente que necesitamos determinar cual es el dominio de definición del problema; el número de casos posibles para algunos problemas pueden ser unos pocos, pero en otros muchos problemas, el número de ejemplares será

infinito. En el primer caso, comprobar que el algoritmo funciona para todos los casos posibles no nos llevará mucho tiempo, pero en el segundo caso ¿Cuál es el procedimiento para comprobar que el algoritmo es correcto para todos y cada uno de los ejemplares si estos son infinitos (contables)? De manera empírica esto es imposible, pero de manera formal sí lo es.

Para la especificación formal tenemos

- Para los algoritmos.- las técnicas pre/post, basadas en la inducción matemática y la lógica de predicados.

- Para las estructuras de datos.- la especificación algebraica de los Tipos de Datos Abstractos (TDA, que son conjuntos de datos con sus operaciones asociadas).

1.6.4. Análisis de eficiencia e implementación

Una vez demostrado que el algoritmo es eficaz y correcto, el siguiente paso es analizar la *eficiencia* del algoritmo. La medida de eficiencia se hace en función del **tamaño del problema** (no confundirlo con el dominio de definición del mismo)**.** Este tamaño vendrá dado por el número de datos de entrada o el tamaño de estos datos para cada ejemplar. El resultado, normalmente, será una medida del tiempo de ejecución para llegar a la solución; en algunas ocasiones podrá ser una medida de la necesidad de almacenamiento en memoria durante su ejecución.

Una manera de analizar la eficiencia podría ser, una vez implementado el algoritmo, crear unos vectores de prueba (conjuntos de datos de entrada), que simularan distintos conjuntos de datos y luego medir los tiempos de ejecución. Esta aproximación tiene varias limitaciones importantes:

1. La más significativa, es que el tamaño del problema que podríamos probar estaría limitado por las prestaciones de la máquina en la que se hagan las pruebas. Cada vez que las máquinas fueran más potentes, tendríamos que volver a ejecutar, incluso rehacer, los vectores de prueba para volver a analizar la eficiencia.

2. Para cada posible algoritmo que se diseñara, habría que diseñar una batería de datos de prueba, lo que, en términos de coste de trabajo, es inviable.

3. Habría que tener en cuenta la influencia del lenguaje y del compilador utilizado para analizar el algoritmo.

Otra aproximación es realizar el análisis de manera formal, como se hizo en la especificación (y que aquí no se aborda). Esto nos va a permitir abstraer el algoritmo de la implementación, es decir, de las limitaciones de la máquina y de la influencia del lenguaje y su compilador asociado. El análisis del comportamiento del algoritmo se realiza una única vez y la elección del lenguaje de implementación (cada uno tiene sus restricciones y peculiaridades) se hará teniendo en cuenta el comportamiento de aquel y su aplicación.

Es igual de importante realizar este análisis tanto de la secuencia de acciones como de la posible estructura de datos elegida, ya que esta última influye mucho en el comportamiento global del algoritmo.

Antes hemos dicho que el resultado puede ser una medida de tiempo de ejecución o en consumo de recursos de memoria; pues bien, estos dos tipos de resultados son antagonistas: Un algoritmo que solucione un problema concreto puede tardar muy poco en ejecutarse, pero esto lo hará en detrimento del consumo de memoria; otro algoritmo podría consumir más tiempo, pero consumir menos memoria. Tanto la memoria como el tiempo son recursos críticos en los computadores, por lo que hay que llegar a una solución de compromiso entre estas dos medidas.

Existe un enfoque híbrido, teórico-práctico, en el que se realiza una batería de pruebas para una máquina y, con los datos obtenidos, efectuar cálculos estadísticos para generalizar los resultados a tamaños de problema mayores. Este enfoque es muy crítico, pues se puede partir de premisas erróneas a la hora de efectuar los cálculos estadísticos y por lo tanto dar por válido cualquier resultado.

Al análisis de eficiencia también se le llama **análisis de coste computacional y análisis de complejidad**.

1.7.- ELIGIENDO UN ALGORITMO

Lo primero que tenemos que determinar es la frecuencia de uso del algoritmo (intuitivamente ya se dijo al principio): si esta es esporádica o muy pequeña es posible que no sea rentable invertir tiempo en buscar un algoritmo más eficiente, pues el tiempo utilizado sería superior al invertido en ejecutar el algoritmo "poco eficiente".

También hay que tener en cuenta el tamaño real del problema. Se pueden dar casos que, dados dos algoritmos, el primero sea menos eficiente que otro para tamaños grandes del problema, pero para casos no tan grandes (los que se van a dar en realidad), el primero puede ser más eficiente que el segundo, por lo que lo adecuado será elegir el primero con la solución elegida.

Una limitación que hay que tener en cuenta es que hay que llegar a un compromiso entre la sencillez de lectura por parte del equipo de programación y la eficiencia del algoritmo. Un algoritmo muy eficiente pero difícil de entender conlleva que su mantenimiento será muy costoso (es un tiempo que también hay que tener en cuenta). El equipo que lo realiza puede ser distinto del que desarrolló el algoritmo, y por tanto puede ser muy costoso entender el funcionamiento del algoritmo antes de acometer modificaciones (el tiempo que ganamos con el algoritmo se pierde con el tiempo en realizar el mantenimiento).

Ejemplo 1.3

Llega la navidad y es tiempo de escribir las postales de felicitación, esta vez serán 27. Pero resulta que estamos a 22 de diciembre (el día de la salud) y todavía no las he enviado. Mi problema es que quiero que las reciban antes del día de navidad, pero los servicios postales tienen más trabajo en estas fechas y no llegarían. Como todos mis conocidos están cerca, decido no usar estos servicios y llevarlas yo mismo puerta por puerta. Así que, escribo las postales, salgo de casa y las voy introduciendo en los buzones sin ningún tipo de orden ni preferencia. Solo me lleva una tarde (4 horas) y un buen paseo ¡No está mal!

- Mi problema: enviar las postales de manera que lleguen antes del 25 de diciembre
- Mi algoritmo: llevarlas en persona, entregándolas sin ningún tipo de orden ni preferencia.
- Tamaño actual del problema: 27 postales.

Pero ¿qué ocurriría si los servicios postales utilizaran mi algoritmo? Pues que se necesitarían unos enormes recursos humanos y materiales, pues la cantidad de envíos que maneja son inmensos. Así, para el mismo problema pero con, por ejemplo, un tamaño del problema de 5 millones de postales, con mi algoritmo tardarían 2 años con miles de empleados.

Por tanto, para los servicios postales, se hace necesario buscar un algoritmo más elaborado que mejore el tiempo que tardan el llegar las postales. Por ejemplo: si se ordenan las postales por ciudades y calles, se mejoraría el algoritmo, de una manera drástica: para un tamaño de 5 millones de envíos, conseguiríamos hacerlo en ¡3 días!. El tiempo que se invertiría en ordenar por calles y ciudades es mucho menor que el que se consumirían si se utilizara el primer algoritmo. Sin embargo para mí, puede suponer una tarde completa ordenarlas y decidir una ruta y otra tarde más repartirlas, mientras que, con el primer algoritmo, tan solo era una; luego el segundo algoritmo a mí no me sirve.

Supongamos que se tardan 24 horas en encontrar un tercer algoritmo más eficiente todavía que el segundo, y que, como con el primer algoritmo, yo siga tardando una tarde en hacer mis 27 envíos, es más, solo tardo una hora en hacerlo . Sin embargo, este tercer algoritmo no mejora que los servicios postales tarden 3 días para 5 millones de envíos. En resumen, hemos mejorado el segundo algoritmo pero solo para un número de envíos es pequeño.

¿Cuál es el algoritmo elegido? Para mí, si con el primer algoritmo tardaba 4 horas y con el tercero ahora tardo una, gano 3 horas de cada vez, por lo que este tercer algoritmo me será rentable si hago lo mismo durante más de 8 navidades (hasta amortizar las 24 horas de su diseño), mientras que para los servicios postales el adecuado es el segundo. La siguiente figura nos nuestra el crecimiento del tiempo de

ejecución en función del tamaño del problema de los tres algoritmos de nuestro ejemplo.

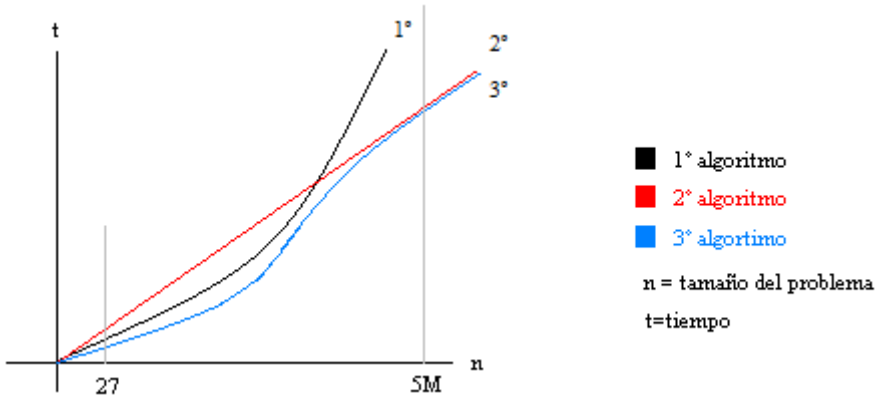


Figura 1.1. Muestra simbólica del tiempo consumido para cada solución

--

CAPÍTULO 2.- PRINCIPIOS DEL ANÁLISIS

OBJETIVOS

- Distinguir ente caso peor y caso mejor en la ejecución de un algoritmo
- Comprender en qué consiste el criterio asintótico y los órdenes de complejidad
- Entender en qué afecta una constante al coste computacional de un algoritmo
- Entender cómo se elige un algoritmo

PREREQUISITOS

- Teoría de conjuntos (operaciones y relaciones)
- Representación de funciones
- Conocer los tipos de funciones matemáticas típicas

CAPÍTULO 2.- PRINCIPIOS DEL ANÁLISIS

2.1.- ANALISIS DE EFICIENCIA DE ALGORITMOS

Al analizar la eficiencia de un algoritmo, lo que pretendemos hacer es *caracterizar*, predecir, el comportamiento del mismo con respecto al consumo de recursos (tiempo y/o memoria principal). Es decir, lo que queremos saber es **como crece el consumo de recursos en función del tamaño del problema**.

Para ello, lo que se va a hacer es contar el número de acciones (instrucciones elementales) que va a realizar el algoritmo. Imaginemos que nuestro algoritmo tiene 15.000 acciones. Analizar de manera pormenorizada el tiempo que tarda en realizar cada una de estas es una tarea ímproba, pues habría que hacerlo para cada ejemplar del problema (y eso si el número de ejemplares es finito).

Como lo que queremos es *caracterizar* el comportamiento, podremos ignorar ciertas medidas y parámetros, por lo que será válido un **análisis aproximado del algoritmo, pero lo suficientemente exacto para representar el comportamiento,** que es lo que realmente nos interesa. Para ello lo que buscamos es una función matemática aproximada que identifique el **orden de crecimiento**[3] de los recursos (tiempo o espacio) en función del tamaño del problema (o tamaño de los ejemplares), que llamamos ***n***. Además, con esta función, nos centraremos en valores grandes de n (ya vimos que con el análisis empírico esto no se podía hacer).

Aunque a partir de ahora se hable de tiempo de ejecución, hay que tener en cuenta que es igualmente aplicable al consumo de memoria.

[3] Es normal encontrar varios sinónimos: orden de crecimiento, orden de complejidad, orden del coste computacional

2.2.- ¿QUÉ ES ESO DEL TAMAÑO DEL PROBLEMA?

Cada ejemplar va a tener un tamaño. En un computador ese tamaño será el espacio en memoria que ocupa el mismo. Pero, desde un punto de vista formal, en nuestro análisis, este enfoque no es válido ya que al determinar un número de bits, estamos incluyendo implícitamente las limitaciones de la máquina (detalles de implementación). Por tanto, no es válido.

Siguiendo en nuestro enfoque aproximado, el tamaño del problema dependerá de la naturaleza del mismo y no del número de instrucciones que tenga el algoritmo:

- Ordenar un número de elementos: el tamaño será el número de elementos a la entrada

- Operaciones con enteros grandes: el tamaño será el valor de la entrada.

- Operaciones en grafos: número de nodos y/o aristas.

Así, para el análisis de los algoritmos que dan solución a un mismo problema, habrá que especificar la forma de determinar el tamaño del problema.

Ahora nos queda saber qué es el tamaño del problema utilizado para el análisis. Para cada ejemplar, el algoritmo tendrá un comportamiento distinto, con tiempos de ejecución distintos. Ya hemos dicho que no vamos a analizar el algoritmo para cada ejemplar, sino que vamos a elegir un ejemplar representativo del problema. Podremos tener tres opciones:

- Escoger el mejor caso, el óptimo.- Es una mala elección porque este ejemplar suele ser trivial y, por tanto, poco representativo.

- Escoger el caso promedio.- Normalmente nos encargarán realizar un algoritmo para un problema nuevo y concreto, en cuyo caso no tenemos estadísticas para poder determinar un caso promedio. Si por el contrario, se nos pide mejorar uno ya existente, calcular un caso promedio puede ser un trabajo estadístico de

gran envergadura que además puede fallar; los algoritmos pueden cambiar su comportamiento según aumenta n. Es más, en muchas ocasiones, el caso promedio se acerca al caso peor.

- Escoger el peor caso posible.- Esta es la elección más acertada. Tomando el caso peor para tamaños grandes del problema será la forma más fácil de predecir el comportamiento.

2.3.- CRITERIO ASINTOTICO

Ya se ha dicho que lo que buscamos es una función matemática que represente el tiempo de ejecución en función del tamaño del problema. Vamos a llamar **T(n) la función representativa del tiempo consumido por nuestro algoritmo en el peor caso** (en función del tamaño del problema)**.** Esta función, será positiva para valores de n grandes.

Hay veces que es necesario hacer un análisis de requisitos de memoria, además del tiempo. Hay que tener en cuenta que hay algoritmos que mejoran su eficiencia aumentando su ocupación en memoria principal; este comportamiento puede llevar al agotamiento de esta, necesitando el uso de la memoria secundaria (en los sistemas con memoria virtual). Los tiempos de respuesta de la memoria secundaria son de varios órdenes de magnitud superior a la de la memoria principal, por lo que el análisis que se efectúe no será válido.

El diccionario de la RAE define asíntota como una curva que se acerca de continuo a una recta u otra curva, sin llegar nunca a encontrarla (el concepto matemático es el ya conocido). Para caracterizar nuestro análisis lo que vamos a buscar son **unas funciones que acoten asintóticamente nuestra función T(n) para el peor caso**[4]. Aquí asintótico se utiliza para indicar que el criterio elegido es en el límite del tiempo conforme aumenta el tamaño del problema, cuando este es muy grande (tendiendo hacia el infinito); es a los que nos referimos con “sin llegar nunca a tocarla”.

[4] Se ha dado esta definición porque estamos acostumbrados a ver las asíntotas como rectas y no es el caso.

Esto significa que puede darse el caso de funciones T(n) que sean secantes con la que se considera asintótica para valores pequeños de n, pero asintótica para valores de n suficientemente grandes (esto se explicará más adelante).

2.4.- OPERACION ELEMENTAL

Es aquella cuyo tiempo de ejecución es una constante, y será función de la implementación que se vaya a adoptar (prestaciones del procesador, lenguaje utilizado….).

Como normal general identificamos como operaciones elementales:

- Operaciones aritméticas: Sumas, restas, multiplicaciones y divisiones (valor pequeño de los operadores)

- Asignación y comparación

- Operaciones booleanas y de módulo (paso de parámetros)

Esta generalización, como todas, hay que tomarla con reservas: puede haber casos en los que no se puedan tomar como elementales:

- Operaciones aritméticas.- Hay que tener en cuenta que, para cualquier implementación, su coste computacional aumenta linealmente con el tamaño de los operandos, por lo que, cuando estos son grandes, hay que tenerlo en cuenta (dejarían de ser elementales)

- Asignación y comparación.- Si esta operación se realiza sobre variables estructuradas (vectores, matrices o registros) y sus elementos, hay que tener cuidado de que estas operaciones no estén relacionadas con el tamaño del problema. Por ejemplo: en un vector, el acceder al un elemento del mismo no es una operación elemental si para ello primero hay que ordenarlo.

- Operaciones.- Hay que tener en cuenta que no estén relacionadas con el tamaño del problema.

Todas estas consideraciones también hay que tenerlas en cuenta a la hora de especificar el problema, para que esta sea también correcta.

Dadas varias implementaciones (máquina y/o lenguaje) y dada una operación elemental, esta tendrá un tiempo constante distinto en cada una de aquellas. Llamaremos C el mayor valor de los tiempos de esta operación, de manera que será una cota superior al tiempo de operación elemental en cualquier implementación.

$$T_{\text{operación elemental}} \leq C$$

Este valor C denota el tiempo máximo que puede tomar una operación elemental. Con el objeto de independizar el coste computacional del algoritmo de la o las posibles implementaciones que podamos tener del mismo (que como vimos era uno de los inconvenientes del análisis empírico), vamos a suponer que **las operaciones elementales tienen un coste unitario**, ya que, dado que nuestro análisis es aproximado y para casos muy grandes, este valor C será insignificante en comparación con el tiempo total del algoritmo. Para saber el tiempo de cada implementación, solo habrá que multiplicar el resultado de nuestro análisis por la constante C que corresponda (luego se demuestra formalmente).

Así el análisis de eficiencia[5] consiste en 'contar' el número de operaciones elementales con el ejemplar que represente el peor caso, de manera que este será el tiempo máximo que tardará el algoritmo para cualquier ejemplar del problema. Este tiempo máximo estará representado por la función T(n), donde $n \in \mathbb{N}$ y $T(n) \geq 0$ (es obvio que sea no negativa; recordar que es para valores de n suficientemente grandes, hecho que se mostrará en el siguiente capítulo).

2.5.- ÓRDENES DE COMPLEJIDAD (DE CRECIMIENTO)

Hemos dicho que en análisis de eficiencia queremos predecir el comportamiento, lo que realmente estamos diciendo es que queremos saber cómo crece el consumo de los recursos en función del tamaño del problema.

[5] Como ya se indicado, todo lo que se diga para el tiempo se puede aplicar a la memoria

Para comparar cada posible algoritmo que soluciona un problema, necesitamos alguna herramienta que nos permita comparar la función T(n) de cada uno de ellos.

Para ello, lo primero que vamos a hacer es clasificar todas las posibles funciones T(n) con arreglo al **orden de crecimiento** del consumo de tiempo con respecto del tamaño del problema. Cada una de estas funciones podrá estar caracterizada por el comportamiento siguiente:

- tiempo de ejecución constante (unitaria para nuestro análisis)
- crecimiento logarítmico(log (n))
- crecimiento lineal (n)
- crecimiento cuadrático (n^2)
- crecimiento polinómico (n^a , con a >2)
- crecimiento exponencial (a^n)
- crecimiento factorial (n!)
- crecimiento polinómico exponencial (n^n).

Esta lista de órdenes de crecimiento constituye en sí una **jerarquía creciente de órdenes de complejidad.** Un orden de complejidad es un conjunto de funciones, representado por la función más sencilla de las incluidas en él (las típicas se indican en la lista anterior[6]), cuya característica es que, cada una de ellas, y en los términos aproximados de este análisis, no supera el orden de crecimiento de la función que lo representa. Así, podemos definir la jerarquía en términos de conjuntos:

$$\text{Orden}(1) \subset \text{Orden}(\log n) \subset \text{Orden}(n) \subset \text{Orden}(n^2) \subset \text{Orden}(n^a) \subset$$
$$\text{Orden}(a^n) \subset \text{Orden}(n!) \subset \text{Orden}(n^n)$$

¿Cómo elegimos la función representativa? Dado que estamos analizando para valores de n muy grandes, en una función polinómica podemos ignorar los términos de orden inferior y también el coeficiente multiplicativo para el término de mayor grado, ya que ambas son despreciables cuando n es suficientemente grande (en la notación asintótica se

[6] Esta lista no es exclusiva; también hay otras como n log (n)

verá cuando no se puede aplicar este método). En esta situación el valor de los términos de orden inferior es insignificante y el coeficiente multiplicativo se puede eliminar pues (luego se mostrará formalmente) la diferencia entre las funciones será una constante multiplicativa *C*, que podemos eliminar por ser nuestro análisis aproximativo. Como podemos ver en la jerarquía antes definida, ya se han elegido las funciones típicas representativas de cada uno de los conjuntos "Orden de". Las condiciones formales de pertenencia a uno de estos órdenes de complejidad se detallan en el siguiente capítulo.

Ejemplo 2.1

Supongamos las funciones $T_1(n)$ y $T_2(n)$, representativas de dos algoritmos que solucionan un mismo problema. Si

$$T_1(n) = 2n^2 + n + 1 \quad \text{y} \quad T_2(n) = 47n^2 + 7$$

y eliminamos los términos de grado inferior, n y 1 en $T_1(n)$ y 7 en $T_2(n)$, y los coeficientes multiplicativos, 2 en $T_1(n)$ y 47 en $T_2(n)$, podemos ver que

$$T_1(n) \approx n^2 \text{ y } T_2(n) \approx n^2, \text{ luego}$$

$$T_1(n), T_2(n) \in \text{Orden } (n^2)$$

--

2.6.- COMPARANDO ÓRDENES DE COMPLEJIDAD

Podemos decir que, para valores de n suficientemente grandes (luego veremos qué significa esta puntualización), un algoritmo tiene un peor comportamiento computacional si su T(n) pertenece a un orden superior al orden del segundo.

Ejemplo 2.2

Supongamos dos funciones $T_1(n)$ y $T_2(n)$ que son representativas de dos algoritmos que solucionan un mismo problema. Si

$$T_1(n) = 2n^3 + n + 1 \quad \text{y} \quad T_2(n) = 47n^2 + 7 \text{ entonces}$$

$$T_1(n) \approx n^3 \text{ y } T_2(n) \approx n^2, \text{ luego}$$

$$T_1(n) \in \text{Orden } (n^3) \text{ y } T_2(n) \in \text{Orden } (n^2)$$

Así podremos decir que $T_2(n)$ tiene un mejor comportamiento computacional que $T_1(n)$, para valores de n suficientemente grandes.

--

Hay que hacer notar que, aplicando la definición dada para orden de complejidad (por inclusión de conjuntos), podríamos decir también que $T_2(n) \in$ Orden (n^3), ya que $T_2(n)$ tiene un orden de crecimiento inferior a n^3. Aunque esto es correcto bajo este enfoque de conjuntos no lo es desde el enfoque asintótica, por lo que se considerará que la función T(n) de un algoritmo pertenece al conjunto "Orden de" con el menor orden de complejidad posible, ya que es este el que la caracteriza (es la que la acota asintóticamente); en el ejemplo $T_2(n) \in$ Orden de (n^2). Esta condición se verá formalmente con la notación asintótica.

Hemos visto que, como normal general, se eliminan las constantes multiplicativas (coeficiente del término de mayor grado de la función). Esta eliminación hay que hacerla con ciertas reservas, pues esta puede llevar a un análisis erróneo.

Suponer que hacemos un análisis para tamaños muy grandes del problema pero que, en realidad, ese tamaño nunca se va a dar, siendo, por el contrario, sensiblemente menor. En esta situación el análisis de la función con la constante multiplicativa sí puede ser importante. Como norma general, siempre que las constantes multiplicativas son significativas, es necesario hacer el análisis teniéndolas en cuenta, para saber a partir de qué tamaño del problema es bueno uno u otro algoritmo. Este es uno de los dos puntos que esconde la frase *para un tamaño del problema suficientemente grande* (el otro punto lo veremos en la notación asintótica).

Formalmente, dados dos algoritmos que solucionan un mismo problema y cuyas funciones representativas T(n) tienen un orden de crecimiento distinto, las curvas que representan a estas funciones se pueden cortar en algún punto a partir del cual la T(n) de orden superior tiene un crecimiento más rápido, es decir

$$\forall n > n_o \rightarrow T_1(n) > T_2(n)$$

Para valores superior a n_o el algoritmo representado por $T_2(n)$ tiene un mejor comportamiento computacional que el representado por $T_1(n)$ (consume menos tiempo). Veamos un ejemplo de esto

Ejemplo 2.3

Supongamos dos funciones $T_1(n)$ y $T_2(n)$ que son representativas de dos algoritmos que solucionan un mismo problema. Si

$$T_1(n) = 2n^3 + n + 1 \quad y \quad T_2(n) = 100n^2 + 51 \text{ entonces}$$

$$T_1(n) \approx n^3 \text{ y } T_2(n) \approx n^2, \text{ luego}$$

$$T_1(n) \in \text{Orden } (n^3) \text{ y } T_2(n) \in \text{Orden } (n^2)$$

Así podremos decir que $T_2(n)$ tiene un mejor comportamiento computacional que $T_1(n)$, para valores de n suficientemente grandes. El valor de n para el cual esto es cierto es

$$T_1(n) = T_2(n) \text{ luego } 2n^3 + n + 1 = 100n^2 + 51$$

La solución válida es n=50

Es decir, que $T_2(n)$ tiene un mejor comportamiento computacional que $T_1(n)$, para valores de n superior a 50. Para valores inferiores a este valor es mejor el comportamiento de $T_1(n)$.

--

Una aproximación visual al comportamiento de estas dos funciones podría ser la siguiente (aproximación porque estas dos curvas no representan gráficamente las funciones del ejemplo anterior)

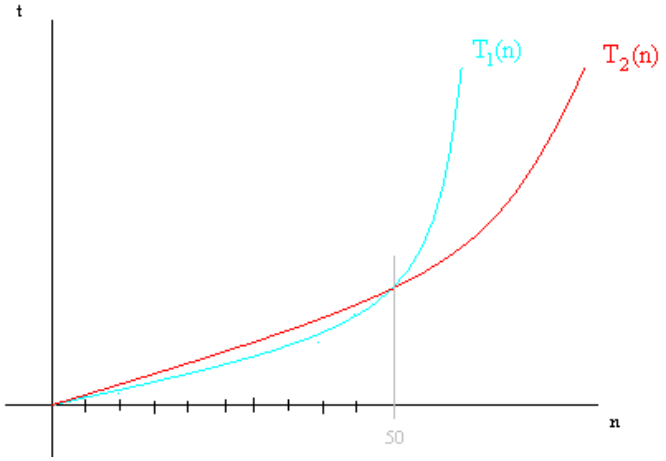


Figura 2.1. Aproximación visual $T_1(n)$ y $T_2(n)$

En resumen, el caso general es válido salvo que se den, en las funciones, constantes multiplicativas grandes y/o tamaños de problema moderados. También hay que ser cuidadosos a la hora de eliminar coeficientes en T(n), pues estos pueden ser determinantes a la hora de elegir uno u otro algoritmo, dada la implementación real que se vaya a realizar.

2.7.- PRINCIPIO DE PERSISTENCIA (INVARIACIÓN)

En la vida real los sistemas son escalables, esto es, con el tiempo va aumentando el tamaño del problema, con lo que la eficiencia se va degradando. Entonces se nos plantea la siguiente cuestión ¿es mejor reducir el orden de complejidad de un algoritmo o, por el contrario, mejorar la máquina, es decir, la implementación?

Cuando hablamos de las operaciones elementales, ya dijimos que el tiempo que consume una operación elemental en cada implementación está acotado superiormente por una constante C; si generalizamos esta afirmación para un algoritmo completo, se formula el **principio de persistencia o invariación** que dice: "dadas dos implementaciones cualesquiera distintas de un mismo algoritmo, la eficiencia de ambas no variará en más de una constante multiplicativa (normalmente positiva)". Es decir, que cualquier implementación de un mismo algoritmo está en el mismo orden de crecimiento.

Formalmente, dado un problema, tenemos un conjunto A de algoritmos y, a su vez, para cada uno de ellos, a_i, tenemos un conjunto M_i de implementaciones; así para cada algoritmo podríamos acotar los costes con sendas constantes C_1 y C_2:

$$\text{Dados } A = \{ a_i \mid a_i = \text{algoritmo}\}, \text{ y } M_i = \{m_i \mid m_i = \text{implementaciones de } a_i\}$$

$$\forall a_i \in A; \exists m_j, m_k \in M_i ; T(n) \geq 0 \rightarrow \exists C_1, C_2 > 0 \text{ tal que}$$

$$Tm_j(n) \leq C_1 \, Tm_k(n)$$

$$Tm_k(n) \leq C_2 \, Tm_j(n)$$

Para valores de n suficientemente grandes, la eficiencia está doblemente acotados por C_1 y C_2, que solo dependen de la implementación.

Con este principio ya podemos contestar a la pregunta que nos hacíamos al principio de este apartado y que reformulamos ¿es mejor reducir el orden de complejidad o reducir en una constante de tiempo de ejecución? Lo haremos con un ejemplo:

Ejemplo 2.4

Una fábrica especializada produce un producto muy exclusivo. El equipo de cálculo que se utiliza es muy potente, y utiliza un algoritmo cuya eficiencia viene dada por

$$T_1(n) = 3^n \quad \text{(segundos)}$$

El número de productos máximo que se puede producir al año es de 15 unidades completas.

Sin embargo, esta producción es insuficiente para la demanda que se tiene, por lo que la junta directiva decide adquirir un nuevo equipo de cálculo, con una capacidad 50 veces mayor que la del equipo antiguo, con un coste de medio millón de euros (el algoritmo que va a utilizar va a ser el mismo).

Por el principio de persistencia sabemos que

$$T_{vieja1}(n) \leq 50\, T_{nuevaa1}(n)$$

Con lo que, resolviendo la inecuación, resulta que con este nuevo equipo, se puede llegar a una producción de 19 unidades.

La competencia, que también tiene la primera máquina con el mismo algoritmo, decide contratar una empresa de informática para estudiar la mejora del mismo, encontrando uno cuyo comportamiento está caracterizado por:

$$T_{a2}(n) = 50n^4 + 3 \text{ (segundos)}$$

Y cuyo coste ha sido también de medio millón de euros. Con este algoritmo ahora la segunda empresa podrá producir 28 unidades al año.

Como podemos ver, la segunda empresa, con el mismo gasto, ha conseguido casi duplicar la producción, mientras que la primera solo aumenta su producción aproximadamente una cuarta parte (la primera posiblemente cerraría porque el aumento de su producción no cubriría la demanda, y por tanto, tampoco los gastos producidos por la nueva máquina).

¿Qué comportamiento tiene el nuevo algoritmo para valores pequeños de producción? Pues procedemos a comparar las funciones representativas de ambos algoritmos

$$\forall n > n_o \rightarrow 3^n + n > 50n^4 + 3$$

solucionando la inecuación, tenemos que $n_o = 13$, es decir, para valores de producción inferiores a 13, es mejor utilizar el algoritmo antiguo.

--

En los ejemplos que se han propuesto hasta ahora, se han dado funciones T(n) concretas, con unidades en segundos, días o años, es decir, para una implementación concreta. Pero, si lo que pretendemos es hacer un análisis independiente de la implementación ¿qué unidades utilizaremos?

Ya hemos dicho que lo que hacemos en el análisis es contar acciones elementales, luego por la misma definición del método, no tendremos ninguna unidad de medida. Esta la dará la constante multiplicativa asociada al procesador que ejecute el algoritmo. Lo único que indicaremos es la pertenencia de T(n) a alguno de los órdenes de complejidad antes expuestos: diremos que "T(n) está en el orden de ". Como *C* es la cota superior de tiempo de una operación elemental para una implementación dada m,

$$T_{\text{operación elemental}} \leq C_e$$

Aplicando el principio de persistencia podemos afirmar que una implementación es proporcional a la función representativa

$$T(n) \leq C\, t(n)$$

2.8.- ELIGIENDO UN ALGORITMO (y II)

Para elegir un algoritmo u otro, por regla general elegiremos el más eficiente, pero sin dejar de tener en cuenta:

- Si el algoritmo se va a utilizar pocas veces, el coste de buscar uno más eficiente puede ser superior al que tiene el procesamiento menos eficiente, pero más directo, y por tanto es mejor elegir este último (véase el ejemplo final del primer capítulo).

- Si el algoritmo se va a utilizar con tamaños pequeños o moderados del problema, hay que tener en cuenta las constantes multiplicativas del término más significativo de la función para elegir el adecuado.

- Cuidado con los algoritmos crípticos en su escritura. Pueden ser muy eficientes pero imposibles de mantener (e incurrir en otros gastos superiores a la ineficiencia).

- Estas técnicas están pensadas para caracterizar el comportamiento en la memoria principal. Cuidado con los algoritmos que mejoran su eficiencia a costa de consumir memoria (y si es secundaria, como ya se ha dicho, el estudio carece de validez por mezclar distintos órdenes de magnitud).

CAPÍTULO 3.- NOTACIÓN ASINTÓTICA

OBJETIVOS

- Conocer la notación asintótica
- Comprender bien la notación O y distinguirla del resto
- Comprender la utilización de las distintas reglas que se utilizan en el análisis de eficiencia
-

PREREQUISITOS

- Teoría de conjuntos (operaciones y relaciones)
- Límites y logaritmos
- Representación de funciones

CAPÍTULO 3.- NOTACIÓN ASINTÓTICA

3.1.- ORDEN DE COMPLEJIDAD (DE CRECIMIENTO)

Recordar que hemos dicho que T(n) es una función, con un dominio de definición $n \in \mathbb{N}$, que nos muestra el comportamiento del algoritmo en el caso peor y que pertenece a un orden de complejidad determinado. T(n), por regla general (hay excepciones que veremos luego), se define como una función Real no negativa; formalmente

$$T: \mathbb{N} \rightarrow \mathbb{R}^{+} \cup \{0\}, donde\ \mathbb{R}^{+} \cup \{0\} \equiv \mathbb{R}^{\geq 0}$$

Cuando decimos que T(n) ha ser positiva, lo que queremos decir es que lo sea para valores de n suficientemente grandes (para valores pequeños de n esta función podría ser nula o incluso parcial- no estar definida-).

También hemos mostrado (muy a la ligera) como se puede simplificar una función polinómicaT(n) para representar el orden de complejidad (según hemos definido anteriormente), pero ¿formalmente cómo sabemos que T(n) pertenece a un orden de crecimiento dado? Para ello vamos a definir las **medidas asintóticas**.

Dada una T(n) representativa del comportamiento del algoritmo y una función f(n)

$$f: \mathbb{N} \rightarrow \mathbb{R}^{+} \cup \{0\}$$

representativa de un orden de complejidad, diremos que **T(n) está en el orden de complejidad representado por f(n)** si, para un n suficientemente grande, existe una constante Real positiva que cumpla con la definición y que se explica a continuación:

$$\exists C \in \mathbb{R}^{+}, \forall n \geq n_0 \in \mathbb{N} \mid 0 \leq T(n) \leq C\, f(n) \rightarrow T(n) = O(f(n))$$

O en términos de conjuntos,

$$O\left(f(n)\right) = \{t: \mathbb{N} \rightarrow \ \mathbb{R}^{+} \cup \{0\} \mid \exists C \in \mathbb{R}^{+}, \forall n \geq n_0 \in \mathbb{N} \rightarrow T(n) \leq C\, f(n)\}$$

Donde O (f(n)), leído *"O de f de n"*, es el orden de complejidad representado por f(n). Dijimos que el orden de complejidad era un conjunto, pero como se ve en la primera definición, no se utiliza la pertenencia a conjuntos (t(n) ∈ O (f(n))) como sería de esperar, sino una expresión algebraica de igualdad. Esta es la notación tradicional y es así porque de esta manera se simplifica la notación en expresiones algebraicas (al revés no funciona). Antes de explicar esto, enunciaremos algunas reglas.

¿Qué podemos deducir de esta definición? Que, superado un umbral n_0, la función representativa, multiplicada por una constante, se convierte en una curva asintótica que acota por encima a la función T(n). Como esta definición lo es sólo para la función por encima del umbral n_0, T(n) puede ser negativa o incluso no estar definida por debajo de este valor y, aun así, seguir perteneciendo a O(f(n)).

Ejemplo 3.1

Supongamos una $T(n) = 2n^2 - n - 10$

Si resolvemos la ecuación T(n) = 0, tiene dos raíces: 5/2 y -2, de las cuales la segunda no pertenece a ℕ, por lo que no es válida. Tenemos que T(n) es no positiva para $n \leq 5/2$, luego el umbral deberá ser superior a 5/2 si queremos cumplir la definición (función no negativa).

Como T(n) es un polinomio, suprimiendo los términos de menor grado y el coeficiente multiplicativo (para n muy grande son insignificativos), podemos afirmar que su orden de complejidad es $O(n^2)$. Pero vamos a demostrarlo. Si elegimos un valor de umbral $n_0 = 3$, aplicando la definición

$$\forall\ n \geq 3,\ \exists\, C \in R^{+} /\quad 0 \leq 2n^2 - n - 10 \leq C\, n^2$$

sustituimos n por el valor del umbral, y nos queda que

$$C \geq 5/9 = 0{,}556,$$

Luego podríamos decir que, para cualquier constante C ≥ 5/9, f(n) cumple con la definición, pero esto no es realmente cierto, ya que, recordemos, estamos analizando para valores de n suficientemente grandes.

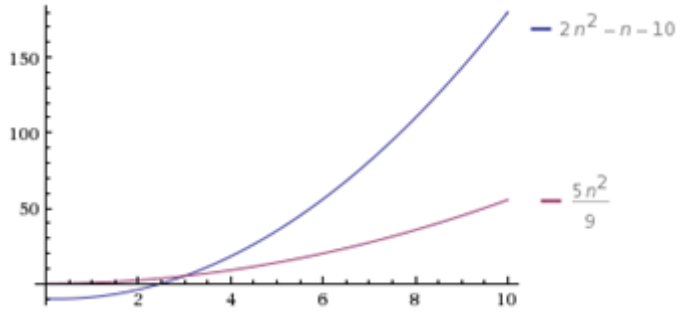


Figura 3.1. Comparación no asintótica de T(n) y su orden de crecimiento

En este polinomio sería válido quitarle los términos de menor grado (son insignificantes para tamaños muy grandes), ya que restan, por lo que T(n) siempre será un poco más pequeña (si los términos de menor orden fueran positivos sería distinto); por ello sería válido decir

$$\exists C \in \mathbb{R}^+, \forall n \geq n_0 \in \mathbb{N} \mid 0 \leq 2n^2 \leq C\, n^2$$

resolvemos la última ecuación y nos queda que $C \geq 2$, y luego, si calculamos el umbral con T(n)

$$2n^2 - n - 10 \leq 2n^2 \rightarrow n \geq -10$$

Por lo que podemos decir que, a partir de n_o=5/2 y para cualquier constante C ≥ 2, f(n) cumple con la definición, es decir, $2\,n^2$ acota asintóticamente a T(n).

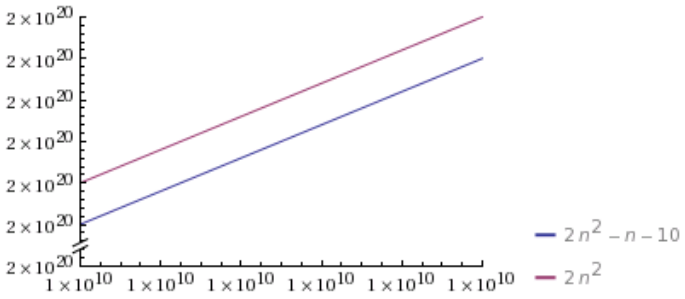


Figura 3.2. Comparación asintótica de T(n) y su orden de crecimiento para n grande

--

Otra forma de ver esta definición es diciendo que cualquier función g(n) que pertenezca al orden O(f(n)) ha de ser positiva y asintótica con Cf(n) más allá de n_0.

Además, al estar acotando superiormente a T(n), podría darse el caso de que f(n) fuese de un orden superior. En este caso también se cumple la definición y es coherente con la inclusión de órdenes de complejidad antes mencionado; sin embargo la acotación no es asintótica. Para esta situación tenemos la notación o(f(n)), leído *"o pequeña de f de n"*, que podemos definir como :

$$\forall C \in \mathbb{R}^+, \forall n \geq n_0 \in \mathbb{N} \mid 0 \leq T(n) \leq C\, f(n) \rightarrow T(n) = o(f(n))$$

cuya única diferencia es que es para todas las constantes, mientras que O lo es solo para alguna constante.

Como ya se ha dicho anteriormente, a los efectos del análisis computacional, lo necesario es asociar el orden de complejidad con la menor f(n) que caracterice el crecimiento de T(n) y esta será O(f(n)).

Por último, el hecho de definir la función representativa de un orden de complejidad sin constante multiplicativa, lo es debido a que se calcula a partir de la definición.

3.2.- REGLA DEL UMBRAL

Sean las funciones $T: \mathbb{N} \rightarrow \mathbb{R}^+$ y $f: \mathbb{N} \rightarrow \mathbb{R}^+$, estrictamente positivas, y sea el umbral $n_0 = 1$ (es decir, están definidas para todo ℕ), se puede enunciar la **regla del umbral,** que dice que

$$T(n) \in O\left(f(n)\right) \leftrightarrow \exists\, C \in \mathbb{R}^+, \forall\, n \in \mathbb{N} \mid 0 \leq T(n) \leq C\, f(n)$$

O en términos de conjuntos

$$O\left(f(n)\right) = \{T: \mathbb{N} \rightarrow \mathbb{R}^{+} \mid \exists\, C \in \mathbb{R}^{+}, \forall\, n \in \mathbb{N} \mid 0 \leq T(n) \leq C\, f(n)\ \}$$

Es decir, para toda T(n) estrictamente positiva, si existe una constante C en la que para todo valor de $n \in \mathbb{N}$, se cumple que T(n) $\leq C$ f(n), entonces T(n) está en el orden de f(n).

Ejemplo 3.2.

Supongamos T(n) = $2n^2$ + n + 10, estrictamente positiva, $f_1(n)= n^3$ y $f_2(n)= n^2$, también son estrictamente positivas.

Si aplicamos la regla del umbral para $f_1(n)$ y $f_2(n)$, es fácil demostrar que, para C≥13, T(n) es asintótica con C $f_1(n)$ con o pequeña

$$T(n) \in o\,(n^3) \quad o \quad T(n) = o\,(n^3)$$

mientras que para cualquier C se cumple que T(n) $\leq C$ $f_2(n)$, es decir

$$T(n) \in O\,(n^2) \quad o \quad T(n) = O\,(n^2)$$

--

En este ejemplo se ha demostrado que es válido decir que una función T(n) está en un orden superior al mínimo suyo. Sin embargo, esto ocultaría realmente la caracterización del comportamiento que buscamos; así, lo correcto es determinar siempre el menor de los órdenes a los que pertenece la función T(n). En este ejemplo, lo necesario es conocer que T(n) ∈ O (n^2).

Se podría decir que la definición de orden de complejidad dado en el capítulo anterior es una regla del umbral generalizada.

3.3.- REGLA DE LA SUMA

Como hemos dicho, la forma de iniciar el análisis es contando operaciones elementales, es decir, sumando los tiempos unitarios de todas ellas. De esta forma, un algoritmo se puede dividir en dos partes independientes, de manera que

$$T(n) = T_a(n) + T_b(n), \text{ donde } T, T_a, T_b : \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}, siendo\ \mathbb{R}^+ \cup \{0\} \equiv \mathbb{R}^{\geq 0}$$

donde $T_a(n)$ y $T_b(n)$ son las funciones representativas del coste computacional de cada una de las partes en que hemos dividido el algoritmo. A su vez, cada una está en el orden de complejidad

$$T(n) = O(f(n)), \quad T_a(n) = O(f_a(n)) \text{ y } T_b(n) = O(f_b(n))$$

La regla de la suma dice que la suma, $O(f_a(n)) + O(f_b(n))$, de los órdenes de complejidad de las funciones representativas, $T_a(n)$ y $T_b(n)$, de los órdenes de complejidad a las que pertenece cada división del algoritmo, es del orden de complejidad, $O(f(n))$, al que pertenece $T(n)$. Formalmente

$$T(n) = f(n) + g(n) \rightarrow O(f(n)) = O(f_a(n)) + O(f_b(n)) \rightarrow O(f(n)) = O(f_a(n) + f_b(n))$$

En termino de conjuntos $O\ f(n) = O(f_a(n)) \cup O(f_b(n))$; al ser la unión una operación cerrada para los conjuntos y la suma para las operaciones algebraicas, tradicionalmente se hace la equivalencia (la suma algebraica no se puede sustituir por la unión de conjuntos). Así las operaciones a realizar con los conjuntos orden de complejidad son más fáciles.

La generalización de la regla para varias particiones del algoritmo es inmediata por aplicación de la propiedad asociativa de la suma.

3.4.- REGLA DEL MAXIMO

Vamos a analizar de la regla de la suma. Se nos pueden dar tres casos:

- Si $f_a(n)$ y $f_b(n)$ están en el mismo orden de complejidad, ambas estarán acotadas por una misma función *C* h(n). Por la misma definición h(n) = O (f(n)), y por tanto se puede ver fácilmente que $f_a(n)$ y $f_b(n)$ son equivalentes y a su vez equivalentes a f(n) en cuanto a su orden de complejidad.

- Si $f_a(n)$ está en un orden inferior a $f_b(n)$, el orden de $f_b(n)$ ya contiene a $f_a(n)$ (por la inclusión de órdenes de complejidad), por lo que $T_a(n)$ y $T_b(n)$ estarán acotadas por una misma función *C* $f_b(n)$.

- Si $f_b(n)$ está en un orden inferior a $f_a(n)$, el orden de $f_a(n)$ ya contiene a $f_b(n)$ (por la inclusión de órdenes de complejidad), por lo que $T_a(n)$ y $T_b(n)$ estarán acotadas por una misma función *C* $f_a(n)$

Basándonos en esto, podemos enunciar **la regla del máximo**, que dice que el orden de complejidad de la suma de tiempos de varios algoritmos está en el mayor orden al que pertenezcan las funciones de los algoritmos. Formalmente, completamos la regla de la suma diciendo:

$$T(n) = f(n) + g(n) \rightarrow O(f(n)) = O(f_a(n)) + O(f_b(n)) \rightarrow O(f(n)) = O(f_a(n) + f_b(n))$$

luego

$$O(f(n)) = \max(O(f_a(n), O\, f_b(n)) = O(\max(f_a(n), f_b(n))$$

Esta es la regla por la que podemos despreciar los términos de orden inferior en los polinomios: porque están acotados por el término de orden superior y hace que sean irrelevantes para nuestra definición asintótica, es decir, para valores de n suficientemente grandes. Un detalle que hay que tener en cuenta al aplicar esta regla es que cada función ha de ser positiva (asintóticamente hablando); en una función podemos tener términos negativos, pero ¿tiene sentido que una parte de un algoritmo vaya hacia atrás en el tiempo? No. En este caso hay que operar la función para conseguir que cada término sea positivo, para valores de n suficientemente grandes (recordar el ejemplo 1). Lo ilustraremos con otro ejemplo:

Ejemplo 3.3

Supongamos que, para su análisis, hemos dividido un algoritmo en 3 partes, obteniendo las siguientes funciones:

$$P_1(n)= 5n^2 \quad P_2(n)= 2n \quad P_3(n)=5$$

aplicando la regla de la suma sabemos que

$$T(n) = 5n^2 + 2n +5$$

Está claro que $P_1(n)$, $P_2(n)$ y $P_3(n)$ son estrictamente positivas para todo n; por tanto podemos aplicar la regla del máximo. Si los órdenes de complejidad de cada $P_i(n)$ son

$$P_1(n)= O (n^2) \quad P_2(n) = O(n) \quad P_3(n)= O (1)$$

$$T(n) = O (f(n)) = \max (O (n^2), O(n),O (1))$$

$$T(n) = O(n^2)$$

--

Ejemplo 3.4

Supongamos que en el análisis de un algoritmo hemos obtenido la siguiente función

$$T(n) = n^5 - 2n^2 +n$$

tal como está no podemos aplicar la regla de la suma y por tanto tampoco la del máximo. Si sumamos y restamos n^5 nos queda la siguiente función equivalente

$$T(n) = 2n^5 + (n^5- 2n^2) + n$$

Ahora el segundo término es positivo para algún n suficientemente grande (en concreto, $n > 2^{1/3}= 2$), luego ahora sí podemos aplicar la regla del máximo

$$2n^5 = O(n^5),\ (n^5 - 2n^2) = O(n^5)\ \text{y}\ n = O(n)$$

$$T(n) = O(f(n)) = \max(O(n^5), O(n^5), O(n))$$

$$T(n) = O(n^5)$$

--

El ejemplo 3.4 plantea una solución trivial al problema, pero podríamos tener funciones mucho más complicadas que requerirían mucho más cálculo.

3.5.-PERTENENCIA A UN ORDEN DE COMPLEJIDAD. REGLA DEL LÍMITE

Hemos dicho antes que lo normal es caracterizar T(n) con el orden de complejidad menor de los que pertenece. También hemos dicho que T(n) está acotado superiormente por una función que puede ser de un orden superior. Pero hasta ahora no tenemos un medio fácil para determinar que si T(n) = O (f(n)), T(n) no pertenece al orden superior al representado por f(n).

Una manera de hacerlo es demostrarlo por contradicción sobre la definición de orden de complejidad. Si g(n) es estrictamente positiva en $\mathbb{N}$, podemos aplicar la regla del umbral. Si no podemos encontrar ninguna constante que cumpla con T(n) = O (g(n)) es que tenemos una contradicción.

Para probar formalmente que T(n) = O (f(n)), tendríamos que ir demostrando que se cumple la definición, empezando por el orden más bajo, para todos y cada uno de los órdenes de complejidad inferiores hasta llegar a g(n).

Este método puede ser un poco largo. Tenemos otro método más rápido, aunque no aplicable a todas funciones, que es aplicar **la regla del límite.**

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}^+$ Significa que f(n) = O (g_r(n)) y g(n) = O (f_r(n)), es decir, las dos funciones pertenecen al mismo orden de complejidad (son equivalentes en complejidad).

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ Significa que f(n) tiene un orden de crecimiento mayor que g(n), luego g(n) ∈ O (f_r(n)) (en concreto g(n) ∈ o(f_r(n)))y f(n) ∉ O (g_r(n))

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ Significa que g(n) tiene un orden de crecimiento mayor que f(n) luego g(n) ∉ O (f_r(n) y f(n) ∈ O (g_r (n)) (en concreto f(n) ∈ o(g_r(n))).

Donde f_r(n) es la función representativa del orden de f(n) y g_r(n) es la función representativa del orden de g(n).

Hemos dicho que este método no es válido para todas las funciones. Esto es así porque existen funciones de la forma h(n) = f(n) / g(n), para las que no existe el límite (puede ser una función oscilante por ejemplo) o es negativo, aunque f(n) y g(n) estén en el mismo orden de complejidad.

La demostración de que la regla del límite es inmediata por aplicación de la definición de límite en ℕ.

La regla del límite $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$, también se puede utilizar para demostrar que f(n) = o(g(n)).

3.6.- REGLA DEL PRODUCTO

Esta regla es menos importante que las anteriores, pero no menos útil. Dadas dos funciones $T_1(n) \in O$ (f(n) y $T_2(n) \in O$ (g(n)), podemos decir que el producto de las dos funciones está en el mismo orden que el producto de las funciones representativas; en términos de conjuntos:

$$O\left(h(n)\right) = \{h' : \mathbb{N} \to \mathbb{R}^+ \cup \{0\} \mid \exists\, T_1(n) \in O\big(f(n)\big), T_2(n) \in O\big(g(n)\big),$$

$$\forall\, n \geq n_0 \in \mathbb{N} \rightarrow h'(n) = T_1(n)T_2(n)\ \}$$

o lo que es lo mismo

$$h'(n) = O\,(h(n)) = [O\,(f(n))][O\,(g(n))] = O\,[f(n)\ g(n)]$$

Al igual que la regla de la suma, es fácil generalizar la del producto para varias funciones por la propiedad asociativa.

3.7.- ORDEN DE COMPLEJIDAD LOGARITMICO

Podemos tener funciones cuyo orden de complejidad sea logarítmico. Un problema que se nos plantea es ¿en qué base lo indicamos?

Si cogemos la definición de logaritmo y la propiedad por la que se cambia de base, resulta que lo que estamos haciendo es multiplicar por una constante, que, como ya sabemos, la podemos eliminar.

$$\log_a n = \frac{\log_b n}{\log_b a}$$

donde a es una base conocida, luego $\log_b a$ es una constante; así podemos sustituir

$$K = \frac{1}{\log_b a}$$

$$\log_a b = K \ln b$$

Para cada base, K es distinto, pero siempre es una constante, luego por la definición de orden de complejidad, la base en la que demos la función representativa es irrelevante. No obstante, es bueno establecer un convenio sobre la base en la que se representan los logaritmos en notación asintótica (una buena opción es el neperiano, ya que simplifica la notación).

3.8.- AJUSTANDO EL ANALISIS. NOTACION OMEGA (Ω)

Bien, hasta ahora se han planteado el O(f(n) como una curva asintótica que acota superiormente a nuestra T(n). Esto significa que ninguna implementación va a consumir recursos en un orden superior a O (f(n)), y que vamos a elegir como representativa la curva asintótica más ajustada, es decir, ninguna implementación va a consumir más de *C*f(n) con tamaños del problema grandes.

Ahora se trata de encontrar una curva asintótica que acote T(n) por debajo (siempre en el caso peor, claro). Esta tarea es bastante más difícil que la anterior. Al igual que en la notación O, vamos a tener un conjunto de funciones que pertenecen a un orden de complejidad dado, representadas por una función característica y para la que vamos a utilizar la notación omega (Ω).

Dada una T(n) y dada una función f(n),

$$f: \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\}$$

representativa de un orden de complejidad, diremos que **T(n) está en "omega de f de n"** Ω(f(n)), **el orden de complejidad representado por f(n)** y acotado por debajo si, para un n suficientemente grande, existe una constante Real positiva *d* que cumpla con la definición:

$$\exists d \in \mathbb{R}^+, \forall n \geq n_0 \in \mathbb{N} \mid T(n) \geq d\, f(n) \geq 0 \rightarrow T(n) = \Omega(f(n))$$

O en términos de conjuntos,

$$\Omega\left(f(n)\right) = \{t: \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} \mid \exists d \in \mathbb{R}^+, \forall n \geq n_0 \in \mathbb{N} \rightarrow T(n) \geq d\, f(n)\}$$

$$T(n) \in \Omega(f(n))$$

Al igual que las otras medidas, la notación tradicional es utilizando expresiones algebraicas en lugar de expresiones de conjuntos.

Así, podemos decir que, superado el umbral n_0, la función representativa multiplicada por una constante se convierte en una curva asintótica que acota por debajo la a la función T(n). De esta definición también se asume que cualquier función g(n) $\in$ Ω(f(n)) es positiva y asintótica con df(n) para tamaños suficientemente grandes.

A partir de la definición de la cota superior y la inferior a T(n), se puede enunciar la **regla de dualidad.** Como podemos ver si T(n) = Ω(f(n)), estamos diciendo que T(n) está acotada por debajo por *d*f(n). Si aplicamos la definición de la cota superior, podemos ver que f(n) está acotada superiormente por *C*T(n), es decir f(n) = O (T(n)); es más se cumple la razón *C* = 1/d. Formalmente

$$T(n) = \Omega(f(n)) \text{ si y solo si } f(n) = O(T(n))$$

Gracias a esta conclusión, podemos aplicar para esta notación las mismas reglas enunciadas para la notación O: máximo, umbral y del límite.

3.9. UTILIDAD DE LA NOTACION OMEGA (Ω)

Observemos la definición: hemos dicho que T(n) está el conjunto Ω(f(n)) si hay una constante d en la que T(n) ≥ *d*f(n), es decir acotada inferiormente. Se podría pensar que sería correcto decir (según la definición lo es), para la notación omega, que

$$T(n) = \Omega(1)$$

Sin embargo, al igual que con la notación O, estamos buscando la curva asintótica más cercana a T(n) que acote inferiormente a esta. Sin embargo, aquí radica la dificultad de esta notación.

Un error de concepto en el que se puede caer es decir que Ω(f(n)) es la cota inferior del algoritmo, que Ω(f(n)) es la cota para el caso mejor. Ciertamente la definición se podría utilizar así, en cuyo caso estaríamos acotando el comportamiento del algoritmo para cualquier tamaño de n. Sin embargo esta notación se utiliza para **denotar la cota inferior al**

caso peor, con lo que al decir que T(n) = Ω(f(n)) se está indicando que existe un ejemplar de tamaño n que se resuelve con un consumo de recursos de al menos *d*f(n) , es decir, que hay un ejemplar del caso peor que al menos tarda ese *d*f(n) (es la cota inferior a la dificultad del problema en el caso peor).

Así definida, podemos tener infinitos ejemplares del problema que consumen menos de *d*f(n), cosa que no ocurre si se utiliza la notación para denotar el caso mejor.

Además, al estar acotando por debajo a T(n), podría darse el caso de que f(n) fuese de un orden todavía inferior (como hemos dicho antes T(n) = Ω(1)). En este caso también se cumple la definición y es coherente con la inclusión de órdenes de complejidad antes mencionado; sin embargo la acotación no es asintótica. Para esta situación tenemos la notación ω(f(n)), leído *"omega pequeña de f de n"*, que podemos definir como :

$$\forall d \in \mathbb{R}^+, \forall n \geq n_0 \in \mathbb{N} \mid T(n) \geq d\, f(n) \geq 0 \rightarrow T(n) = \omega(f(n))$$

cuya única diferencia es que es para todas las constantes, mientras que Ω lo es para alguna constante.

Una definición alternativa, más intuitiva quizás, es:

$$T(n) \in \omega(f(n)\ si\ y\ solo\ si\ f(n) \in o(T(n))$$

Por último, también se puede utilizar la regla del límite (si existe) para demostrar la pertenencia a omega pequeña f(n) = ω (g(n)):

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

Como ya se ha dicho anteriormente, a los efectos del análisis computacional, lo necesario es asociar el orden de complejidad con la mayor f(n) que caracterice el crecimiento de T(n) y esta será Ω (f(n)).

3.10. NOTACION ORDEN EXACTO

Existen algoritmos para los que no se puede determinar una cota inferior para el caso peor, y para otros resulta que las dos notaciones están en el mismo orden de complejidad. Para el segundo escenario existe la **notación orden exacto**, denotada por le letra griega theta Θ. Decimos que T(n) está en el orden exacto de f(n) si los órdenes cota superior e inferior están en el mismo orden de complejidad.

$$T(n) = \Theta(f(n)) \text{ si y solo si } T(n) = O(f(n)) \text{ y } T(n) = \Omega(f(n))$$

o el conjunto de funciones acotantes superior e inferiormente del mismo orden, igualmente positivas y asintóticas con T(n):

$$\Theta(f(n)) = \{O(f(n)) \cap \Omega(f(n))\}$$

y que, en términos formales, se define como

$$\Theta\left(f(n)\right) = \{t: \mathbb{N} \rightarrow \mathbb{R}^+ \cup \{0\} \mid \exists C, d \in \mathbb{R}^+, \forall\, n \geq n_0 \in \mathbb{N} \rightarrow 0 \leq df(n) \leq T(n) \leq C\, f(n)\}$$

Dado que el orden exacto es una intersección de las dos notaciones anteriores, también le es de aplicación todas las reglas del umbral, suma, máximo, producto. La regla del límite también se puede aplicar, reformulándose de la siguiente forma:

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = C \in \mathbb{R}^+$ Significa que f(n) = Θ(g_r(n)) y g(n) = Θ(f_r(n)), es decir, las dos funciones pertenecen al mismo orden de complejidad (son equivalentes en complejidad).

- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ Significa que f(n) tiene un orden de crecimiento mayor que g(n). Así, f(n) ∈ Ω (g_r(n)) (en concreto f(n) ∈ ω (g_r(n))), sin embargo f(n) ∉ Θ(g_r(n)).

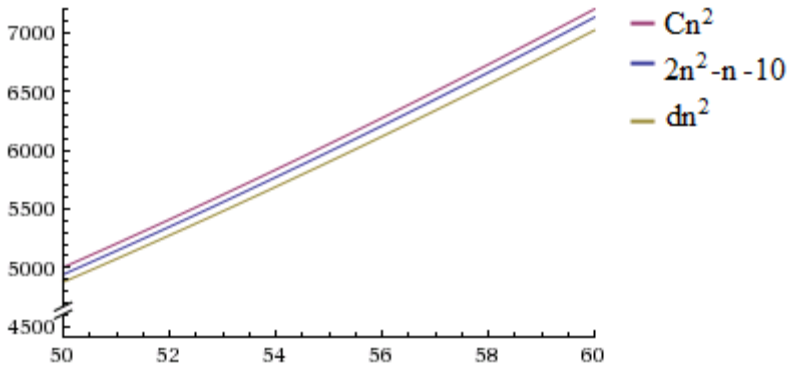
- $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$ Significa que g(n) tiene un orden de crecimiento mayor que f(n). Así f(n) ∈ O (g_r(n)) (en concreto f(n) ∈ o(g_r(n))), sin embargo f(n) $\notin$ Θ(g_r(n)).

Donde f_r(n) es la función representativa del orden de f(n) y g_r(n) es la función representativa del orden de g(n).

Un detalle a tener en cuenta con el orden exacto es que, a partir del valor de umbral, las constantes C y d pueden ser distintas (de hecho lo son).

Ejemplo 3.5

Sea T(n)= $2n^2 - n - 10$. La representación gráfica de la figura siguiente muestra como T(n) = Θ (n^2)



3.4. Comparación asintótica de T(n) y su orden de crecimiento exacto para n grande

--

Dos resultados que se dan con frecuencia en el análisis de eficiencia, y que son de orden exacto son

- Polinomios
- Sumatorios de las formas

$$\sum_{i=1}^{n} i^k = \Theta\,(n^{k+1}) \text{ o } \sum_{i=1}^{n} k^i = \Theta\,(k^{n+1})$$

3.11.- PROPIEDADES DE LA NOTACIÓN ASINTÓTICA

Es fácil demostrar que estas notaciones tienen las propiedades relacionales

- Notación O.- Reflexiva, antisimétrica, transitiva y dualidad con Ω
- Notación Ω.- Reflexiva, transitiva y dualidad con O
- Notación Θ.- Reflexiva, simétrica y transitiva.
- Notación o.- No reflexiva, transitiva y dualidad con ω
- Notación ω.- No reflexiva, transitiva y dualidad con o

Dadas f(n), g(n) y h(n), asintóticas no negativas:

- Reflexiva
  - f(n) ∈ O(f(n))
  - f(n) ∈ Ω(f(n))
  - f(n) ∈ Θ (f(n))
- Transitiva
  - si f(n) ∈ O(g(n)) y g(n) ∈ O(h(n)) entonces f(n) ∈ O(h(n))
  - si f(n) ∈ Ω (g(n)) y g(n) Ω(h(n)) entonces f(n) ∈ Ω (h(n))
  - si f(n) ∈ Θ (g(n)) y g(n) ∈ Θ (h(n)) entonces f(n) ∈ Θ (h(n))
  - si f(n) ∈ o(g(n)) y g(n) ∈ o(h(n)) entonces f(n) ∈ o(h(n))
  - si f(n) ∈ ω (g(n)) y g(n) ∈ ω (h(n)) entonces f(n) ω(h(n))
  -
- Simétrica (solo para orden exacto)
  - si f(n) ∈ Θ (g(n)) entonces g(n) ∈ Θ (f(n))
- Antisimétrica
  - si f(n) ∈ O(g(n)), g(n) ∈O(f(n)) solo es cierta si son del mismo órden
- Dualidad:
  - f(n) ∈ O(g(n)) si y solo si g(n) ∈ Ω(f(n))
  - f(n) ∈ o(g(n)) si y solo si g(n) ∈ ω(f(n))

Al ser conjuntos, para la notación O, podemos definir una relación de orden parcial, ya que cumple las propiedades reflexiva, antisimétrica y transitiva. No es un orden completo,

pues no cumple la propiedad de completud: si f,g son funciones reales positivas en n, es fácil comprobar que existen $f(n) \notin O(g(n))$ y que a su vez $g(n) \notin O(f(n))$.

Por tanto, aun siendo un orden parcial, está justificado que estableciéramos una clasificación de órdenes de crecimiento de las funciones T(n): así decir que $f(n) \in O(g(n))$ significa que, como venimos demostrando, f(n) tiene un *grado de complejidad* menor o igual que g(n). Así mismo es lo que justifica que se pueda utilizar una notación algebraica para el análisis en lugar de una notación de conjuntos, que es lo que realmente se está haciendo.

CAPÍTULO 4.- ANÁLISIS DE EFICIENCIA

OBJETIVOS

- Comprender los órdenes de complejidad
- Comprender como se realiza el análisis de un algoritmo
- Comprender como se demuestra formalmente el coste computacional de un algoritmo no recursivo
- Comprender el análisis de un algoritmo de naturaleza recursiva.

PREREQUISITOS

- Programación en algún lenguaje imperativo que soporte recursividad
- Conocer el uso de la recursividad.
- Inducción matemática

CAPÍTULO 4.- ANÁLISIS DE EFICIENCIA

4.1.- SIGNIFICADO DE LOS ÓRDENES DE COMPLEJIDAD

¿Qué significa cada uno de los órdenes de complejidad a la hora de sacar conclusiones del análisis de eficiencia? Teniendo en cuenta que lo que hacemos es analizar el comportamiento del algoritmo según crece el tamaño, podemos decir que:

- Tiempo constante O(1).- Es independiente del tamaño del problema. Solo es aplicable a operaciones elementales.

  Ejemplo 4.1

  Operación suma: x+y

  Suponiendo la suma de dos números considerados elementales para cualquier valor de x e y, la sentencia está en el orden O(1) (exactamente en Θ(1)).
  --

- Tiempo logarítmico O(log(n)).- Este es el mejor comportamiento real de un logaritmo, ya que el aumento del tamaño del problema afecta muy poco al aumento del tiempo; para un aumento del tamaño k, el tiempo aumenta ln k.

- Tiempo lineal O(n).- Está considerado también como un comportamiento muy bueno. Conforme aumenta el tamaño del problema, aumenta linealmente (proporcionalmente) el tiempo de ejecución.

- Tiempo polinomial $O(n^a)$, para a ≥ 2 .- Aquí ya empiezan los problemas, ya que un pequeño aumento en el tamaño del problema hace aumentar muy rápido el tiempo necesario para ejecutarlo. Se considera que todavía son tratables, aunque de forma

limitada. Multiplicar el tiempo de ejecución por k, solo va a permitir aumentar el tamaño del problema en $n\ \sqrt[a]{k}$. Algunos de estos órdenes tienen nombre propio:

- O(n^2) orden cuadrático
- O (n^3) orden cúbico

- Tiempo exponencial.- Son problemas intratables, incluso para tamaños pequeños del problema. Estos son:
  - Exponencial O (a^n) con a>1.
  - Factorial O (n!)
  - Polinómico exponencial O(n^n)

En conclusión, cuando se crea un algoritmo, lo que interesa es obtener el orden de complejidad, en el caso peor, lo más bajo posible y además con constantes multiplicativas pequeñas. Cuando se comparan algoritmos, generalmente se busca igualmente el menor orden de complejidad. Puede haber excepciones a esta regla, por ejemplo cuando se sabe que el tamaño real del problema no será muy grande y se conoce un algoritmo alternativo que, si bien pertenece a un orden peor a priori, en esos valores intermedios el coste es equivalente, probablemente será mejor utilizar este último (cuando más eficiente, más complicado suele ser un algoritmo, y eso, como sabemos, tiene unos costes de creación y mantenimiento que también hay que valorar).

4.2.- OPERACIONES ELEMENTALES

En un apartado anterior se han clasificado las operaciones elementales y los detalles a tener en cuenta para calificarlas como tal. También hemos dicho que el análisis de eficiencia consiste en 'contar' el número de operaciones elementales con el ejemplar que represente el peor caso, de manera que este será el tiempo máximo que tardará el algoritmo para cualquier ejemplar del problema. Por comodidad las reproducimos aquí

Una operación elemental tomará un tiempo procesarla, el cual será constante en cada implementación. Ya hemos dicho que para el análisis asintótico, esta constante, que varía de una implementación a otra, no es relevante y que por ello se pueden eliminar; es por ello las operaciones elementales se consideran con coste unitario.

Función de operación elemental = $T(n) \leq C(1)$;
tiempo unitario por una constante.

Por comodidad reproducimos aquí las operaciones elementales:

- Instrucciones de asignación y comparación (De TDA[7] no estructurado)
- Instrucciones de entrada/salida, paso de parámetros
- Operaciones aritméticas y lógicas.- sumas restas, multiplicación y división. De todas estas, sus operandos no pueden depender del tamaño del problema. Por ejemplo, en operaciones aritméticas con operadores grandes, son estos los que fijan el tamaño del problema y por lo tanto ya no son elementales.

4.3.-COMPOSICIÓN SECUENCIAL

Esta regla ya la expusimos de forma implícita para demostrar la regla de la suma y del máximo. Si tenemos I_1 e I_2, dos instrucciones elementales con tiempos de ejecución t_1 y t_2 , entonces se cumple que $t = t_1 + t_2$. Si le añadimos una tercera instrucción elemental I_3 con tiempo de ejecución t_3, entonces $t' = t + t_3$.

De forma general, si tenemos dos conjuntos de instrucciones A y B, con tiempos de ejecución t_a y t_b, y con órdenes de complejidad $t_a \in O(t'_a(n))$ y $t_b \in O(t'_b(n)))$, entonces podemos sumar sus tiempos $t= t_a + t_b$, y por la regla del máximo

$$t \in \max(O(t'_a(n)), O(t'_b(n))).$$

Para la aplicación de este procedimiento hay que tener en cuenta que los tiempos de ambos conjuntos no deben ser interdependientes, esto es, que una dependa de algún parámetro de la otra; en ese caso habrá que operarlas para sacar la función correspondiente a la intersección de los conjuntos de instrucciones.

[7] Tipo de Datos Abstracto que no sea un vector o registro

Hay que tener en cuenta que podemos encontrarnos con esquemas de bifurcación y de iteración, llamadas a procedimientos….todos ellos interrelacionados. Por ello, cualquier conjunto de instrucciones se evalua de la parte más interna a la más externa. Dado un conjunto A de instrucciones, puede darse el caso de que, para poder evaluar su tiempo, necesitemos saber el tiempo de un subconjunto B de instrucciones de las que depende (este subconjunto puede ser un bucle interno o la llamada a una función o procedimiento); en ese caso es necesario evaluar primero el subconjunto B para poder evaluar el conjunto más externo A que lo contiene.

4.4- COMPOSICIÓN DE BIFURCACIÓN

Supongamos el siguiente fragmento de código

```
SI (condición)
  ENTONCES {ejecutar conjunto A de instrucciones}
  SINO {ejecutar conjunto B de instrucciones}
```

En este ejemplo tenemos tres tiempos de ejecución: t_c el tiempo de evaluar la condición que está en $O(t'_c(n))$, t_a y t_b respectivamente los tiempos de los conjuntos A y B y con órdenes de complejidad $t_a \in O(t'_a(n))$ y $t_b \in O(t'_b(n))$,

- Si se cumple la condición, entonces se ejecuta el conjunto A. En este caso el orden de complejidad del tiempo consumido dependerá de la evaluación de la condición y del conjunto A

$$T_1(n) = \max [(O(t'_c(n)), O(t'_a(n))]$$

- Si no se cumple la condición se ejecuta el conjunto B. En este caso el orden de complejidad del tiempo consumido dependerá de la evaluación de la condición y del conjunto B

$$T_2(n) = \max [(O(t'_c(n)), O(t'_b(n))]$$

Por tanto, el caso peor de este esquema será la rama que peor orden de complejidad tenga, es decir

$$T(n) \in \max(\max[(O(t'_c(n)), O(t'_a(n))], \max[(O(t'_c(n)), O(t'_b(n))]) \equiv [\text{como es unión, se}$$

$$\text{eliminan los repetidos}] \equiv T(n) \in \max[(O(t'_c(n)), O(t'_a(n)), O(t'_b(n))]$$

4.5.- COMPOSICIÓN DE ITERACIÓN

Son secuencias de operaciones elementales, las cuales son necesarias repetir según una condición. Para esta situación, los algoritmos pueden utilizar dos enfoques distintos: iteración o recursión.

4.5.1. Enfoque iterativo

Esta estructura corresponde con los esquemas de control: MIENTRAS, REPETIR y PARA. Debido a que REPETIR y PARA son particularizaciones del primero, el procedimiento que se mostrará será para el esquema MIENTRAS (mientras se cumpla la condición, realizar la secuencia de operaciones):

```
Inicializar condición
MIENTRAS (condición) hacer
  Secuencia de operaciones
  Cambiar condición
  FIN MIENTRAS
```

- Dada la "condición" hay que encontrar una función que la represente, de forma que llegue un momento en que esta no se cumpla para salir del bucle. Sin perder la generalidad, consideraremos una función elemental en la que una variable de condición se debe decrementar de alguna manera en cada iteración, y que inicialmente será mayor que uno. El valor final de esta función será menor o igual que 1, que corresponde con la condición de salida del bucle.

- El valor inicial de esta función va a indicar el número máximo de veces que se puede repetir el bucle.

- La función hallada ha de tener en cuenta que la comprobación de iteraciones se va a realizar (interaciones+1) veces (la última es la comprobación de salida del bucle).

- Un vez determinada esta función, tenemos que establecer cómo disminuye la función. Esta disminución podrá ser por restas sucesivas o por divisiones sucesivas. Para ello se utilizará la inducción generalizada.

- Siendo k cualquier de las iteraciones, se determinan los valores de las variables para las iteraciones k y k+1

- Una vez que se ha demostrado la validez de la función para k y k+1, ya podemos demostrarla para la salida del bucle.

Si llamamos f a la función representativa de la condición y O(B(n)) al orden de complejidad de la misma y a O(P(n)) el orden de las operaciones del bloque mientras, el coste de cada iteración será:

$$O\,(B(n)) + O\,(P(n)) = \max\,(O\,(B(n))\,,\, O\,(P(n)))$$

y el coste del bucle, donde f es el valor de la función, será:

$$\sum_{i=1}^{f} \max((O\,(B(n))\,,O\,(P(n)))$$

Si el coste P(n) depende de alguna manera de la variación del número de iteraciones, entonces la última ecuación se convierte en una suma de una progresión aritmética o geométrica, según si la disminución es por restas o por divisiones sucesivas, respectivamente. En caso contrario se convierte en:

$$O\,(\max\,(B(n),\, fP(n))$$

La aplicación del método a un bucle PARA es inmediata, en las que no hace falta calcular como varían las variables, pues está establecida en la misma definición de bucle.

Ejemplo 4.2

Supongamos el siguiente algoritmo (aunque está en pseudocódigo no es un esquema, y por lo tanto es analizable). Para facilitar el trabajo de análisis, es bueno conocer qué hace el algoritmo.

```
//Dado un intervalo numérico [1,m], el bucle vsitará el mismo
// alternativamente de a en a por la izquierda y de b en b
// por la derecha, hasta que no queden valores (reflejado por i<j)

//inicialización de variables de condición
i=1
j=m  //m >1
par = true
//bucle con j >i, a>0, b>0 y a>b
MIENTRAS (i<j) HACER
      SI (par=true) ENTONCES  i=i+a
                              par = false
                  SINO        j=j-b
                              par = true
      FIN SI
FIN MIENTRAS
```

Las variables de la condición son i, j y valor. Las dos primeras se van a modificar de forma que

- i de i_0 a i_i con $i_i \geq 1$
- j de m a j_j con $j_j \leq m$

y la tercera simplemente es para que cambie de extremo a la hora de reducir la distancia.

Sea f la función condición. La distancia de las variables i y j va disminuyendo de alguna manera, en este caso (j-i)+1, donde el 1 representa la comprobación de la salida del bucle. Cuando se sale del bucle (cuando $i \geq j$), $f \leq 1$ (que es la condición de salida). Por tanto

$$f = (j-i)+1$$

En cada rama del condicional SI vamos a tener una variación de las variables de condición. Ahora vamos a aplicar inducción. Sea la condición inicial $f_0 = j_0 - i_0 + 1$ (1) y sea k una iteración cualquiera, $f_k = j_k - i_k + 1$ (2), y sea su sucesora la función $f_{k+1} = i_{k+1} - j_{k+1} + 1$ (3)

- Rama ENTONCES, la siguiente iteración vendrá determinada por

$$i_{k+1} = i_k + a \text{ (4) y } j_{k+1} = j_k \text{ (5)}$$

En f sucesora (3) sustituimos por las anteriores(4) y (5) y nos queda

$$f_{k+1} = j_k - i_k - a + 1 \rightarrow f_{k+1} = (j_k - i_k + 1) - a \rightarrow$$

$$\text{[sustituimos por (2)] } f_{k+1} = f_k - a \text{ (6)}$$

- Rama SINO, la siguiente iteración vendrá determinada por

$$i_{k+1} = i_k \text{ (7) y } j_{k+1} = j_k - b \text{ (8)}$$

En f sucesora (3) sustituimos las anteriores (7) y (8) y nos queda

$$f_{k+1} = j_k - b - i_k + 1 \rightarrow f_{k+1} = (j_k - i_k + 1) - b \rightarrow$$

$$\text{[sustituimos por (2)] } f_{k+1} = f_k - b \text{ (9)}$$

Luego, por las ecuaciones (6) y (8), hemos demostrado que i varía de a en a y j de b en b, es decir, por restas sucesivas.

Ahora, se presentan dos casos:

- Que a y b sean iguales, en cuyo caso la función varía de a en a (o b en b, da igual). Hagamos a = C, luego $f_{k+1} = f_k - C$.

- Que a y b sean distintos[8]. En este caso, cada dos iteraciones del bucle, se habrá reducido la distancia entre i y j en a+b. Por lo tanto, de media, en cada iteración se reducirá la distancia en $\frac{a+b}{2}$. Como en el caso anterior, hagamos $\frac{a+b}{2} = C$, luego

$$f_{k+1} = f_k - C$$

De esta forma queda demostrado que para cada iteración $f_k = f_{k-1} - C$

Ahora necesitamos demostrar la salida del bucle. Si nos damos cuenta

$f_k = f_{k-1} - C = f_{k-2} - 2C = f_{k-3} - 3C = \ldots = f_0 - kC$ que será la condición de salida.

Hagamos $f_k = 1$ para la salida del bucle

$$f_0 - kC \leq 1 \rightarrow f_0 - kC = 1 \rightarrow \frac{f_0 - 1}{C} = k \rightarrow [\text{sustituyendo}]\ k = \frac{m - 1}{C}$$

y el resulta de k será el número de iteraciones del bucle.

Una vez identificado el número de iteraciones del bucle, se puede proceder a analizar el algoritmo. Se puede decir que el tamaño del problema viene fijado por la constante *m* y el caso peor viene fijado por el caso de que a=b=1=C

- El SI interior: todas las instrucciones de las dos ramas están en el orden O(1) (son todo asignaciones, sumas y restas)

- Cada iteración del bucle tendrá un coste en el orden O(1), ya que tanto la instrucción de bifurcación como la comparación están en ese orden.

[8] Aunque se especifica que deben ser positivos los dos, no viene mal echar un vistazo al caso de que sean negativos o de signo distinto. Si ambos son negativos, en todo caso el bucle será infinito. Si a es positivo y b negativo habría que comprobar que |a| > |b| para que el bucle no sea infinito (como lo sería al revés). Si a es negativo y b positivo habría que comprobar que |a| < |b| para que el bucle no fuera infinito. Estos dos casos lo que se está aumentando es los límites del bucle. Si no se hubiese restringido, todo esto debería ser tenido en cuenta en nuestro análisis (estos detalles son más de diseño, pero se indican por claridad).

- Como la ejecución de la instrucción de bifurcación no depende de número de iteraciones del bucle (este algoritmo realmente solo tiene modificaciones en las variables de condición), su orden de complejidad será

$$\max (O (\max (O(1), kO(1))$$

  Como $f_{k\,=}$ 1 en la salida y C= 1, entonces k= m-1 y por lo tanto la iteración tendrá un coste, eliminando términos de orden inferior, en O(m).

- Las instrucciones de inicialización también están en el orden O(1), por lo que el coste de todo el algoritmo está en O(m), que es un coste lineal.

Ejemplo 4.3

Supongamos el siguiente algoritmo (aunque está en pseudocódigo no es un esquema, y por lo tanto es analizable).

```
//inicialización de variables de condición
i=1
j=m  //m >1
par=true
//bucle con j >i,
MIENTRAS (i<j) HACER
      SI (par=true) ENTONCES  i=(i+j)/2)+1  //división entera
                              par=false
                  SINO        j=(i+j/2)-1  //división entera
                              par=true
      FIN SI
FIN MIENTRAS
```

Las variables de la condición son i, j y valor. Las dos primeras se van a modificar de forma que

- i de i_0 a i_i con $i_i \geq 1$
- j de m a j_j con $j_j \leq$ m

y la tercera simplemente es para que cambie de extremo a la hora de reducir la distancia.

Sea f la función condición. La distancia de las variables i y j va disminuyendo de alguna manera, en este caso (j-i)+1, donde el 1 representa la comprobación de la salida del bucle. Cuando se sale del bucle (cuando i ≥ j) f ≤1 (que es la condición de salida). Por tanto

$$f = (j-i)+1$$

En cada rama del SI vamos a tener una variación de las variables de condición. Ahora vamos a aplicar inducción. Sea la condición inicial $f_0 = j_0 - i_0 + 1$ (1) y sea k una iteración cualquiera, $f_k = j_k - i_k + 1$ (2), su sucesora será $f_{k+1} = i_{k+1} - j_{k+1} + 1$ (3); la división entera se define como $[(i+j)DIV\ 2] \leq \frac{i+j}{a} < [(i+j)DIV\ 2) + 1]$

- Rama ENTONCES, la siguiente iteración vendrá determinada por

$$i_{k+1} = \frac{i_k + j_k}{2} + 1\ (4)\quad y\quad j_{k+1} = j_k\ (5)$$

En f sucesora (3) sustituimos por las anteriores(4) y (5) y nos queda

$$f_{k+1} = j_k - \left[\frac{i_k + j_k}{2} + 1\right] + 1 \quad \rightarrow \quad f_{k+1} = \frac{j_k - i_k}{2} \quad \rightarrow$$

$$[sustituyendo\ por\ (2)] \rightarrow$$

$$f_{k+1} = \frac{j_k - i_k + 1}{2} - \frac{1}{2} \rightarrow \quad f_{k+1} = \frac{f_k}{2} - \frac{1}{2}\ (6)$$

- Rama SINO, la siguiente iteración vendrá determinada por

$$i_{k+1} = i_k\ (7)\ y\ j_{k+1} = \frac{i_k + j_k}{2} - 1\ (8)$$

En f sucesora (3) sustituimos las anteriores (7) y (8) y nos queda

$$f_{k+1} = \left[\frac{i_k + j_k}{2} - 1\right] - i_k + 1 \rightarrow \quad f_{k+1} = \frac{j_k - i_k}{2} \quad \rightarrow$$

$$[sustituyendo\ por\ (2)] \rightarrow$$

$$f_{k+1} = \frac{j_k - i_k + 1}{2} - \frac{1}{2} \rightarrow \quad f_{k+1} = \frac{f_k}{2} - \frac{1}{2} \quad (6)$$

Luego, por la ecuación (6), hemos demostrado que i se reduce a la mitad cada vez, es decir, por divisiones sucesivas.

Como estamos haciendo un estudio para tamaños del problema grandes, se puede hacer la aproximación

$$f_{k+1} = \frac{f_k}{2}$$

ya que en promedio, las inexactitudes que se produzcan por la división entera, se compensa con la eliminación del ½.

Ahora necesitamos demostrar la salida del bucle. Sustituyamos r= ½ Si observamos

$$f_k = \frac{f_{k+1}}{2} \rightarrow \quad f_k = r f_{k+1}$$

entonces de forma recursiva se cumple que $f_k = rf_{k-1} = r^2 f_{k-2} = r^2 f_{k-3} = \ldots = r^k f_0$ y la condición de salida del bucle es $f_k \leq 1$, por lo que sustituyendo

$$f_k = r^k f_0 \leq 1 \text{ [si lo hacemos igual a 1]} \rightarrow 1 = r^k f_0 \rightarrow$$

$$[\text{sustituyendo por (1)}] \rightarrow 1 = r^k (j_0 - i_0 + 1) \rightarrow \frac{1}{r^k} = j_0 \rightarrow$$

$$\left[\text{sustituyendo } r = \frac{1}{2} \text{ y } j_0 = m\right] \rightarrow m = 2^k \rightarrow \log_2 m = k$$

donde k es el número de iteraciones del bucle.

Una vez identificado el bucle, se puede proceder a analizar el algoritmo. Se puede decir que el tamaño del problema viene fijado por la constante *m* y el caso peor viene fijado por él mismo

- El SI interior: todas las instrucciones de las dos ramas están en el orden O(1) (son todo asignaciones, sumas y restas).

- Cada iteración del bucle tendrá un coste en el orden O(1), ya que tanto la instrucción de bifurcación como la comparación están en ese orden.

- Como la ejecución de la instrucción de bifurcación no depende de número de iteraciones del bucle, su orden de complejidad será max (O (max (O(1), kO(1))

- Como $f_{k=}$ 1 en la salida, entonces k=$\log_2 m$ y por lo tanto la iteración tendrá un coste computacional del orden[9] O(log(m)).

- Las instrucciones de inicialización también están en el orden O(1), por lo que el coste de todo el algoritmo está en O(log(m)), que es un coste mejor que el anterior.

--

Estos dos ejemplos nos han adentrado en la demostración formal de la complejidad de un algoritmo con un bucle. En muchas ocasiones no necesitaremos realizar todos estos cálculos tan exactos y será válido un análisis no tan somero. Para ello, una técnica muy utilizada es buscar la instrucción elemental que más veces se va a ejecutar en el algoritmo, ya que su coste será dominante sobre el resto de fragmentos del algoritmo. A esta instrucción elemental se la denomina **instrucción crítica o barómetro.**

En los dos ejemplos anteriores es fácil ver que la instrucción crítica es la bifurcación SI-ENTONCES-SINO, la cual hemos visto que está en el coste constante O(1), y que se ejecutará las veces que determina el bucle, m en el primer caso y log m en el segundo, conclusiones que coinciden con los resultados de la demostración.

[9] Recordar que en el orden de complejidad no es necesario indicar la base.

Ejemplo 4.4:

Suponer que hay que ordenar un array con n enteros, que el primer índice del array es 1 y el algoritmo es de ordenación (creciente) por inserción directa (aunque está en pseudocódigo no es un esquema, y por lo tanto es analizable).

```
//El primer elemento se considera ordenado consigo mismo
 PARA (finOrdenado entre 2 y n)
      temporal = array[finOrdenado] //guarda el elemento a insertar
      nuevaPosicion=finOrdenado
      //hasta que el elemento temporal sea mayor que el siguiente,
      hacia atrás, en la parte
      //ordenada
      MIENTRAS (nuevaPosicion > 1 )
            Y (temporal < array[nuevaPosicion-1])
            //pasar el elemento actual a la derecha preparar el
      siguiente
            Array[nuevaPosicion]=array[nuevaPosicion-1]
            nuevaposicion=nuevaPosicion-1
      FIN MIENTRAS
            //inserta el elemento guardado en la posición calculada
            Array[nuevaPosicion]= temporal
FIN PARA
```

Analicemos un poco el funcionamiento del algoritmo[10]. Como vemos, son dos bucles anidados. El segundo bucle (el MIENTRAS) tiene garantizada la salida del mismo por la primera condición. La segunda condición compara el valor al insertar con el siguiente en la lista de números ordenados. Dentro del bucle se produce la asignación de desplazamiento y el decremento de la posición en la parte ordenada.

¿Cuál es el tamaño del problema? Obviamente será el tamaño n del array; pero ¿Cuál es el caso mejor y cuál el peor? Veamos, si el array está ordenado, resultará que, en cada iteración del bucle exterior, el interior se ejecutará solo una vez. Si está en orden inverso resultará que para cada elemento del array tendrá que compararlo con todos sus predecesores y desplazar toda la lista ordenada a la derecha. Según esto, nuestro caso peor será este último. Además, podemos identificar la instrucción crítica como la segunda condición del 2º bucle (el MIENTRAS), ya que realizará estas comparaciones:

[10] Para un análisis más profundo del funcionamiento del algoritmo, se recomienda consultar [Wirth 1986]

Si finOrdenado es 2, realizará una comparación
Si finOrdenado es 3, realizará dos comparaciones
Si finOrdenado es 4, realizará dos comparaciones
…
Si finOrdenado es n, realizará n-1 comparaciones

Todas las operaciones del algoritmo son elementales, luego, basándonos en la instrucción crítica, el coste del algoritmo será 2+3+4+…+ (n-1). ¿esto a qué suena? A la suma de una progresión aritmética, luego

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

Eliminando la constante, resulta que la función representativa del algoritmo de inserción directa pertenece al orden de complejidad $O(n^2)$. Es más, como T(n) es un polinomio, este algoritmo está en el orden exacto $\Theta(n)$.

En el caso mejor, cuando el array está ordenado, el bucle mientras se realizará exactamente una vez, por lo que el coste del bucle exterior será T'(n)=(n-1), que está en el orden O(n). Observar que esto no es la notación omega, sino simplemente el caso mejor.

--

4.5.2. Enfoque recursivo

Para ponernos en situación con la recursividad, primero recordaremos que una función recursiva es aquella en la que la función puede aparecer en las dos partes de la igualdad de la ecuación. Otra forma de verlo, la que es útil para el análisis de eficiencia, es como una construcción inductiva del problema, en la que tenemos un caso base o caso trivial, y un caso general que es una función con recurrencias; este caso general debe utilizar otra función que determine la descomposición recursiva para que cada subproblema sea más pequeño hasta llegar al caso trivial. Un algoritmo recursivo solo se puede implementar en un lenguaje de programación que admita funciones con llamadas recursivas (a sí mismo).

Al igual que en el caso iterativo, para la demostración formal de T(n) recursiva tenemos los siguientes pasos:

- Dada la "descomposición" hay que encontrar una función que la represente, de manera que esta se debe calcular para que el valor final sea el de la condición trivial, que será la condición de terminación del algoritmo.

- El valor de esta función nos va a decir el número máximo de descomposición del problema que se producirá.

- La función hallada ha de tener en cuenta que la comprobación de subproblemas a resolver se va a realizar en (pasos+1) veces (la última es la comprobación de caso trivial).

- Un vez determinada esta función, tenemos que establecer cómo disminuye la función. Esta disminución podrá ser por restas sucesivas o por divisiones sucesivas. Para ello se utilizará la inducción generalizada.

- Se demostrará que la función es válida para los pasos k y k+1

- Una vez que se ha demostrado para k y k+1, ya podemos demostrar para el caso trivial

¿Qué conclusiones sacamos? Las que ya sabemos: que un algoritmo recursivo tiene su equivalente iterativo (nomenclatura aparte), y por eso el método de análisis es, en esencia, el mismo. Sin embargo, el análisis de eficiencia en un algoritmo con recurrencias es bastante más difícil que en el caso iterativo[11]. Dado que estamos dando una introducción al análisis de la eficiencia, solo esbozaremos la eficiencia de estos algoritmos de una forma aproximada (suficiente para un enfoque asintótico).

[11] En caso de desea profundizar, un buen comienzo puede ser [Brassard et al 1997] y la bibliografía en él recomendada, si bien son necesarios conocimientos de álgebra lineal para sacarle provecho.

Como hemos dicho tenemos dos tipos de decremento: por restas sucesivas y por divisiones sucesivas. Para ambos casos, cualquier problema resuelto recursivamente se podrá representar en forma de sus ecuaciones características generales:

- Por restas sucesivas:

$$T(n) = \begin{cases} T(n_o) = cn^k & caso\ base:\ n = n_o \\ aT(n-b) + cn^k & caso\ general: si\ b > 0 \quad n \geq b \end{cases}$$

Donde

- $T(n_o)$ es el coste de la parte no recursiva
- a es el número de llamadas recursivas dentro de la función
- cn^k es el coste del resto de instrucciones no recursivas del caso general

- Por divisiones sucesivas

$$T(n) = \begin{cases} T(n_o) = cn^k & caso\ base: n = n_o \\ aT(n/b) + cn^k\ caso\ general: si\ b > 0 \quad n \geq n_o\ y\ b\ potencia\ exacta\ de\ \dfrac{n}{n_o} \end{cases}$$

Donde

- $T(n_o)$ es el coste de la parte no recursiva
- a es el número de llamadas recursivas dentro de la función
- cn^k es el coste del resto de instrucciones no recursivas del caso general

Una vez que hemos determinado T(n) y cada una de sus variables, se aplicará la siguiente relación de pertenencia (cuya demostración no se hará aquí):

- Para restas sucesivas

$$T(n) \in \begin{cases} \Theta\,(n^k) & si\ a < 1 \\ \Theta\,(n^{k+1}) & si\ a = 1 \\ \Theta\left(a^{n\ div\ b}\right) & si\ a > 1 \end{cases}$$

- Para divisiones sucesivas

$$T(n) \in \begin{cases} \Theta\,(n^k) & si\ a < 1 \\ \Theta\,(n^k \log n) & si\ a = 1 \\ \Theta\left(n^{\log_b a}\right) & si\ a > 1 \end{cases}$$

Ejemplo 4.5

Supongamos la siguiente función recursiva, que implementa el factorial de un número entero no negativo:

```
FUNCIÓN  Factorial (n) DEVUELVE resultado
     SI (n<2)
                ENTONCES  resultado = 1;
                SINO resultado = n * Factorial(n-1);
     FIN SI
     DEVUELVE resultado;
FIN FUNCIÓN
```

Siendo la primera llamada *resultadoFinal=factorial(n)*

Es un caso claro de restas sucesivas (n-1). La ecuación característica de este algoritmo será:

$$T(n) = \begin{cases} T(n_o) = 1 & caso\ base:\ n < 2 \\ T(n-1) + 2 & caso\ general:\ n \geq 2 \end{cases}$$

En el caso trivial solo hay una asignación y en el caso general hay una multiplicación y una asignación; ambas están en el orden constante O(1), luego

$$\Theta\,(n^k) = O\,(1) = O(n^0) \text{ así k = 0;}$$

por otro lado a=1, ya que solo hay una llamada recursiva, y b=1 ya que se reduce de 1 en uno hasta el caso trivial. De esta forma, aplicando pertenencia determinamos que

$$T(n) \in\ \Theta\,(n^{k+1}) = \Theta\,(n)$$

--

Ejemplo 4.6

Supongamos la siguiente función recursiva, que implementa la búsqueda binaria[12] en un array ordenado con índices entre 1 y n; devuelve el valor del índice que tiene el valor buscado o un valor indicativo de error (aquí mostrado con una constante simbólica).

```
FUNCIÓN busquedaBinaria (valorABuscar, extremoIzquierdo, centro,
extremoDerecho) DEVUELVE resultado
      SI extremoIzquierdo > extremoDerecho
            ENTONCES  resultado=ERROR;
            ELSE SI (valorABusccar=array[centro])
                  ENTONCES resultado= centro
                  ELSE SI (valorABuscar< array[centro])
                        ENTONCES  //mitad izquierda
                         resultado= busquedaBinaria (valorABuscar,
                        extremoIzquierdo,extremoIzquierdo+((centro-
                        1-extremoIzquierdo)/2) , centro-1)
                        SINO      //mitad derecha
                          resultado= busquedaBinaria (valorABuscar,
                        centro+1,centro+1+((extremoDerecho-
                        centro+1)/2), extremoDerecho)
      FIN SI
      DEVOLVER resultado
FIN FUNCIÓN
```

y la llamada inicial será busquedaBinaria(valorABuscar, 1, (n/2), n).

Es un caso claro de divisiones sucesivas (n/2), ya que, en cada paso, divide por la mitad la partición de array que queda por comprobar. Por tanto b=2.

Así, la ecuación característica de este algoritmo será

$$T(n) = \begin{cases} T(n_o) = 1 & caso\ base: n = 1 \\ 1T(n/2) + 1 & caso\ general: n > 1 \end{cases}$$

[12] Para consultar su funcionamiento se recomienda [wirth 1986]

En este caso tenemos dos casos triviales, pero en el caso trivial solo se ejecuta uno de ellos (o error o trivial). Por ello

$$\Theta\,(n^k) = \Theta(1)\,, \mathrm{k} = 0$$

De la misma forma, tenemos dos llamadas recursivas pero solo se ejecuta una en cada paso recursivo. Por ello, a=1, b=2.

Determinamos que $T(n) \in\ \Theta\,((n^k \log n) = \Theta\,(\log n)$.

--

BIBLIOGRAFÍA

[Aho et al 1998] Aho, A.V., Hopcroft, J.,Ullman, J. *Estructuras de datos y algoritmos*. 1ª edición. Méjico: Pearson Educación. Impresión 1998. ISBN 96844443455

[Brassard et al 1997] Brassard, G., Bratley, P. *Fundamentos de Algoritmia.* 1ª Edición. Madrid (España): Pearson Educación S.A, impresión de 2006. ISBN 9788489660007

[Cormen et al 2001]. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C. *Introduction to Algorithms, Second Edition*. International Edition. Editorial McGraw-Hill 2001. ISBN 9780262531962

[Gonzalo et al 1997] Gonzalo, J. Rodríguez, M. *Esquemas algorítmicos: enfoque metodológico y problemas resueltos*. 1ª edición Madrid (España): Editorial UNED impresión de 2006. ISBN 9788436236224

[Hernández et al 2001] Hernández,R. , Lázaro, J., Dormido, R., Ros,S. *Estructuras de datos y algoritmos*. 1ª edición. Madrid (España). Pearson Educación. S.A.2001. ISBN 9788420529806

[Hopcroft 2002] Hopcroft, J., Motwani, R. , Ullman, J.D.. *Introducción a la teoría de Autómatas, Lenguajes y Computación*. 2ª edición. Madrid (España): Pearson Educación S.A, impresión de 2007. ISBN 9788478290567

[Horowitz 1984] . Horowitz, E., *Fundamentals of programming languages*. 2ª edición. Maryland (USA): computer Science Press 1984. ISBN 9780716780016

[Mira et al 1995] Mira, J., Delgado, A.E., Boticario, J.G., Diez, F.J. *Aspectos básicos de la inteligencia artificial.* 1º Edición. Madrid (España): Editorial Sanz y torres, S.L., 2005. ISBN
84-88667-13-2

[Manber, 1989] Manber, U. Introduction to algorithms: a creative approach. 1ª edición. Massachusetts (USA): Addison Wesley. 1989 ISBN 9780201120370

[Peña 1993]. Peña, R. *Diseño de programas. Formalismo y abstracción*. 1ª edición. Madri (España): Editorial Prentice Hall International. Impresión de 1993. ISBN 9780130984500

[RAE 22] *Diccionario de la Real Academia de la Lengua*. 22ª edición [en línea] [consulta 20-3/2015]. Disponible en www.rae.es

[Wirth 1986] Wirth, N. Algoritmos y estructuras de datos. 1ª Edición. Madrid (España). Pretice Hall Hispanoamericana, S.A. Impresión de 1987. ISBN 9789688801130