Ficha 20

Análisis de Algoritmos – Introducción

1.] Introducción y conceptos básicos.

A lo largo de estas fichas de clase hemos visto diversos algoritmos para resolver numerosos problemas. Se hizo evidente (y lo será cada vez en mayor medida) que para el mismo problema pueden plantearse diferentes algoritmos. Por ejemplo, el problema de buscar un valor en un arreglo podía resolverse buscando en *forma secuencial* o en *forma binaria*, y también existen muchos algoritmos diferentes para ordenar un arreglo. La cuestión entonces es la siguiente: si se dispone de varios algoritmos para resolver el mismo problema, ¿cómo comparar el rendimiento de cada uno para decidir cuál aplicar en una situación concreta? Dedicaremos el último apartado de este capítulo justamente a presentar conceptos esenciales de *análisis de algoritmos*, que nos permitan poder hacer esas comparaciones [1].

En general, dado un problema y varios algoritmos para resolverlo, se busca comparar el rendimiento de esos algoritmos en cuanto a algún *parámetro de eficiencia*. Normalmente, los dos parámetros más usados son el *tiempo de ejecución esperado* (o sea, qué tan veloz es de esperar que sea cada algoritmo), y el *espacio de memoria empleado* por cada uno. Lo ideal sería lograr algoritmos que usando la menor cantidad posible de espacio de memoria sean a su vez muy veloces, pero en la práctica ambos parámetros suelen estar en relación inversa: si se desea mucha velocidad, el precio suele ser un mayor uso de memoria, y viceversa [2]. Sin embargo esto no es una regla terminante: hemos visto que el algoritmo de búsqueda binaria es mucho más veloz que el de búsqueda secuencial, y sin embargo ambos algoritmos usan casi la misma cantidad de memoria (el vector en el cual se procede a buscar, y un pequeño número de variables locales auxiliares).

En la práctica, el parámetro de eficiencia más analizado es el del *tiempo de ejecución*, simplemente porque se supone que el espacio de memoria será aproximadamente el mismo en todos los algoritmos diseñados para ese problema, o que las diferencias serán apenas relevantes. Por otra parte, el análisis comparativo suele centrarse en dos situaciones: el rendimiento del algoritmo en el *caso promedio*, y el rendimiento del mismo en el *peor caso*. Ambas situaciones se refieren respectivamente al comportamiento del algoritmo evaluado cuando se presenta *la configuración de datos más común* (o caso *promedio*, que suele ser aquella que surge de tomar los datos en orden estrictamente aleatorio) o bien cuando se presenta *la configuración de datos más desfavorable* (o *peor caso*) [2].

Por ejemplo, es claro que para la búsqueda secuencial el *peor caso* se da cuando el valor a buscar no está en el arreglo, o se encuentra al final o muy cerca del final del mismo, porque en ese caso se deben comprobar los *n* elementos (o casi todos ellos) para terminar la búsqueda. En muchas ocasiones el análisis de algoritmos se centra sólo en el *peor caso*, simplemente porque el análisis en el *caso promedio* suele ser muy complejo, o bien porque se adopta un criterio "pesimista" (o sea, suponer que el peor caso se presentará con mucha

frecuencia, y en todo caso, el algoritmo debe plantearse para funcionar incluso en ese *peor caso*...)

En todas las situaciones, lo que se busca es comparar los rendimientos relativos entre los diferentes algoritmos y no tanto poder medir en forma numérica y minuciosa el rendimiento de cada uno. Es decir, nos interesa poder *demostrar* que el algoritmo de búsqueda secuencial será menos veloz que el de búsqueda binaria en ciertas condiciones, sin tener que tomar un cronómetro (o usar funciones de medición de tiempo del lenguaje usado) y medir los tiempos de ejecución cada vez que queramos estar seguros de lo mismo. Por ese motivo, se busca poder deducir alguna *fórmula* o *expresión matemática* que permita modelar el comportamiento del algoritmo en cuanto al tiempo o el espacio empleado, y luego, sabiendo qué formulas describen mejor a cada algoritmo, se comparan los comportamientos de la funciones o relaciones representadas por esas fórmulas.

Para la deducción de esas fórmulas, se parte del hecho que cada algoritmo tiene lo que podría llamarse un *tamaño* o *volumen* natural. Ese tamaño suele venir dado por el número de datos que debe procesar. Por ejemplo, si pretendemos ordenar un arreglo, el tamaño de ese problema es obviamente el valor *n* que indica cuántos elementos tiene el arreglo. Lo mismo vale para el problema de la búsqueda en un arreglo. En ciertos problemas, pueden usarse otros elementos para dar el tamaño del problema pero en lo que sigue supondremos que viene dado por la cantidad *n* de datos a procesar [2].

Sabiendo el tamaño *n* del lote de datos, se intenta deducir qué fórmula se adapta mejor a las variaciones del tiempo (o el espacio usado) cuando varía *n*, suponiendo los datos configurados en el *caso promedio* o en el *peor caso*. Limitaremos todo nuestro estudio siguiente al análisis del *peor caso*, por ser más simple de plantear.

Veámoslo a partir de un ejemplo: El algoritmo de *búsqueda secuencial* en un arreglo de tamaño *n*, tiene su peor caso cuando el valor a buscar está muy al final (o no está). Podemos ver que el *tiempo total* que insumirá el algoritmo en ese caso, es aproximadamente el que corresponda a efectuar exactamente ***todas*** las comparaciones posibles, o sea, *n comparaciones*. Se dice entonces que el tiempo esperado para el algoritmo de búsqueda secuencial en el *peor caso*, *está en el orden de n* (o sea, nunca demorará más de lo que demore en hacer *n* comparaciones). Esto suele denotarse simbólicamente, expresando que el tiempo esperado para el algoritmo es *O(n)* (léase: *orden n* u *orden de n*). El *O(n)* se conoce también como *orden lineal* [2].

En definitiva: no esperamos que nos den el número medido en milésimas de segundo, sino una expresión que nos muestre una cota superior para el tiempo de ejecución (en realidad, la *mejor* cota superior). Esta forma de expresar el rendimiento de un algoritmo (ya sea para el tiempo, para el espacio, o para el parámetro que se use), se designa como *notación O* (se lee como "*notación O mayúscula*" o también como "*notación Big O*") [2] [3].

Un análisis similar puede hacerse para la búsqueda binaria. Este algoritmo partirá en dos al arreglo tantas veces como sea necesario, quedándose con un segmento e ignorando al otro, hasta dar con el valor buscado. En el *peor caso*, ese valor no estará en el arreglo y deberán hacerse todas las particiones posibles, *efectuando una comparación en cada partición* que no sea desechada. Entonces, la cantidad de comparaciones en el peor caso es aproximadamente igual al número de veces que se divide por dos al vector.

Puede probarse que dado un número *n > 1*, la cantidad de veces que podemos dividir por dos hasta obtener un cociente de 1 es igual al logaritmo de *n* en base dos (o sea, $log_2(n)$ o bien, *log(n)* asumiendo que cuando la base del logaritmo no se escribe es entonces igual a 2). De allí que la búsqueda binaria a lo sumo realizará *log(n)* particiones, y por lo tanto el tiempo de ejecución de ese algoritmo en el peor caso es *O(log(n))*. También se dice que ese algoritmo tiene tiempo de ejecución de *orden logarítmico*. Un análisis más detallado de la búsqueda binaria revela que la misma en realidad hace *log(n) + 1* comparaciones, por lo que también podría decirse que el tiempo de ejecución es realmente *O(log(n) + 1)*. Sin embargo, es común en notación *O* que las constantes se supriman (en un volumen muy grande de datos las constantes pueden despreciarse), se termina diciendo que la búsqueda binaria es *O(log(n))* [2] [3].

En general, todo algoritmo que aplique el criterio de dividir sucesivamente por dos al lote de datos, quedándose con la mitad y desechando la otra en cada partición, tendrá *orden logarítmico*. En informática cada vez que un logaritmo aparece se supone que la base del mismo es dos, pero también puede probarse que en notación *O* la base del logaritmo carece de importancia.

Y bien: la búsqueda secuencial es *O(n)* en el peor caso, mientras que la búsqueda binaria es *O(log(n))* también en el peor caso. ¿Qué significa esto? Es simple: si el arreglo tiene *n = 1000* elementos, la búsqueda secuencial insumirá *1000* comparaciones en el peor caso (con el tiempo que sea que eso implique en la máquina donde corra), mientras que la búsqueda binaria no hará más de *9* o *10* comparaciones en el mismo peor caso. Y se pone mejor: a medida que *n* crece, el logaritmo también crece, pero lo hace a un ritmo de crecimiento muy suave... Si *n = 100000* (cien mil), esa misma cantidad de comparaciones insumirá la búsqueda secuencial, pero la binaria hará a lo sumo *16*... Quiere decir que si pueden diseñarse algoritmos cuyo comportamiento sea logarítmico, se podrá estar seguro que esos algoritmos serán básicamente eficientes en cuanto al tiempo, y muy estables a medida que el número de datos crece.

Por supuesto, existen muchas funciones de orden posibles aunque algunas son muy típicas y frecuentes. Por ejemplo, analicemos intuitivamente el caso del ordenamiento por *Selección Directa* visto en una ficha anterior. Prescindiendo de constantes, básicamente se trata de dos ciclos *for* anidados, de forma que el primero de ellos hace aproximadamente *n* vueltas, y el segundo hace aproximadamente otras *n* por cada una que da el primero. Claramente, esto lleva a un esquema de *n * n* repeticiones, de forma que en cada una de ellas se hace una comparación. Esto sugiere que el total de comparaciones *estará en el orden n al cuadrado*, con lo que el tiempo de ejecución también será $O(n^2)$. En general, cada vez que se presenten dos ciclos anidados con aproximadamente *n* repeticiones cada uno, tendremos *orden cuadrático*.

Evidentemente, lo ideal sería que un algoritmo o acción demore siempre lo mismo para procesar un lote de datos, sin importar si aumenta el valor del tamaño *n* de ese lote. Los algoritmos que hemos visto en las primera fichas (antes de llegar a estudiar el uso de ciclos) son de este tipo: se cargaba siempre la misma cantidad de datos (sin posibilidad de alterar esa cantidad), y por lo tanto la demora era siempre la misma.

Sin embargo, esos ejemplos no son muy significativos pues la cantidad de datos era constante. ¿Qué algoritmos o procesos admitirán que *n* crezca de una corrida a la otra sin alterar su tiempo de ejecución? Por ahora, el único caso que conocemos fue analizado

también fichas anteriores: el acceso a un componente individual de un arreglo. Como sabemos, si queremos acceder al valor en la componente *i* de un arreglo *v*, sólo debemos escribir *v[i]* y con esa expresión se accederá al componente en el mismo orden de tiempo cada vez, sin importar sin el arreglo tiene 2, 3 o 1000 elementos. Cuando un algoritmo o proceso se comporta de esta forma, denotamos su tiempo como *O(1)* (léase: *orden uno* u *orden proporcional a uno*). También se dice que dicho algoritmo tiene tiempo de ejecución de *orden constante*.

Para terminar esta somera introducción al tema del análisis de algoritmos, exponemos una clasificación de las principales y más elementales funciones de orden que suelen aparecer, sin que esto signifique que sean las únicas (más adelante, en otra sección de esta misma ficha, analizaremos con más detalle estas mismas funciones típicas). La última columna de la tabla indica algunos algoritmos o casos que responden a cada función de orden citada [2]:

Tabla 1: Funciones típicas en el análisis de algoritmos.

Función	Significado (cuando mide tiempo de ejecución)	Casos típicos
O(1)	Orden constante. El tiempo de ejecución es constante, sin importar si crece el volumen de datos.	Acceso directo a un componente de un arreglo.
O(log(n))	Orden logarítmico. Surge típicamente en algoritmos que dividen sucesivamente por dos un lote de datos, desechando una parte y procesando la otra.	Búsqueda binaria.
O(n)	Orden lineal. Se da cuando cada uno de los datos debe ser procesado una vez.	Búsqueda secuencial. Recorrido completo de un arreglo.
O(n*log(n))	Surge típicamente en algoritmos que dividen el lote de datos, procesando cada partición sin desechar ninguna, y combinando los resultados al final. No hemos analizado aún algoritmos que respondan a este orden.	Ordenamiento Rápido (Quick Sort).
O(n^2)	Orden cuadrático. Típico de algoritmos que combinan dos ciclos de *n* vueltas cada uno.	Ordenamiento por Selección Directa.
O(n^3)	Orden cúbico. Típico de algoritmos que combinan tres ciclos de *n* repeticiones cada uno. No hemos analizado aún algoritmos que respondan a ese orden	Multiplicación de matrices.
O(2^n)	Orden exponencial. Algoritmos que deben explorar una por una todas las posibles combinaciones de soluciones cuando el número de soluciones crece en forma exponencial.	Problema del viajante. Solución recursiva de la Sucesión de Fibonacci.

2.] Noción intuitiva de la notación *Big O*.

Hemos dicho que una de las motivaciones del *análisis de algoritmos* es la posibilidad realizar *comparaciones de rendimiento* entre distintos algoritmos planteados para resolver el mismo problema. Pero incluso si no se busca en forma inmediata esa comparación, el hecho es que contar con un elemento formal de medición que indique qué tan eficiente es un algoritmo respecto de cierto factor (como el tiempo de ejecución o el consumo de memoria) prepara al programador para tomar decisiones a futuro, cuando efectivamente deba seleccionar el mejor algoritmo para el problema que enfrente, o para estimar en forma correcta los parámetros de uso de ese algoritmo (por caso, para saber si podrá ejecutar ese algoritmo en

una computadora con determinada cantidad de memoria, sabiendo el consumo de memoria que el algoritmo reclama).

Como sabemos, en general los factores de medición de eficiencia empleados para el análisis de un algoritmo son el *tiempo de ejecución* esperado y el *consumo de memoria*, aunque en ocasiones también influye un tercer factor, como es la *complejidad aparente del código fuente* (intuitivamente, si dos algoritmos para resolver un problema tienen tiempos de ejecución y consumo de memoria similares, entonces posiblemente se elegirá el más compacto, claro y sencillo en cuanto a código fuente) [2].

También dijimos que en la práctica, el parámetro de eficiencia más analizado es el del *tiempo de ejecución*, y que el análisis comparativo suele centrarse en dos situaciones: el rendimiento del algoritmo en el *caso promedio*, y su rendimiento en el *peor caso*. El primero se refiere al comportamiento del algoritmo cuando se presenta *la configuración de datos más común* (o *caso promedio*, que suele ser aquella que surge de tomar los datos en orden estrictamente aleatorio) y el segundo cuando se presenta *la configuración de datos más desfavorable* (o *peor caso*) [2].

El análisis procede (en la medida de lo posible) intentando deducir alguna *fórmula* o *expresión matemática* que permita modelar formalmente el comportamiento del algoritmo en cuanto al tiempo o el espacio empleado. Para la deducción de esas fórmulas, se parte del *tamaño* o *volumen* natural del problema, que suele venir dado por el número de datos que se deben procesar (por ejemplo, el tamaño *n* de un arreglo), y se intenta deducir qué fórmula se adapta mejor a las variaciones del tiempo (o el espacio usado) cuando varía *n*, suponiendo el *caso promedio* o el *peor caso*. En general, limitaremos todo nuestro estudio siguiente al análisis del *peor caso*, por ser más simple de plantear.

Vimos el ejemplo del algoritmo de *búsqueda secuencial* en un arreglo de tamaño *n*: su peor caso se da cuando el valor a buscar está muy al final (o no está), ya que entonces el *tiempo total* que insumirá el algoritmo es aproximadamente el que corresponda a efectuar exactamente ***todas*** las *n* comparaciones posibles con lo que entonces el tiempo esperado para el algoritmo de búsqueda secuencial en el *peor caso*, *está en el orden de n*. Y vimos que esto suele denotarse simbólicamente, expresando que el tiempo esperado para el algoritmo es *O(n)* (*orden n, orden de n*, u *orden lineal*).

Esta forma genérica de expresar el rendimiento de un algoritmo (ya sea para el tiempo o para el espacio), se designa como *notación O* (se lee como *notación O mayúscula* o también como *notación Big O*) [2] [3]. Vimos en esta misma ficha que el algoritmo de búsqueda binaria tiene un tiempo de ejecución del orden del *logaritmo de n* (*O(log(n))*) para el peor caso o que el algoritmo de ordenamiento por selección directa ejecuta en un tiempo $O(n^2)$, y que un algoritmo o proceso cuyo tiempo de ejecución es siempre el mismo, sin importar el tamaño del problema (por ejemplo, el acceso a una casilla individual de un arreglo o la ejecución de una asignación simple), tiene tiempo de ejecución constante [1] y se denota como *O(1)*.

Muchos de los algoritmos que aparecen con más frecuencia en el estudio de las estructuras de datos tienen tiempos de ejecución que para el peor caso se comportan en *el orden de funciones* muy comunes y conocidas. Como vimos, los más típicos de esos órdenes son los siguientes (aunque no los únicos: cualquier otra función podría aparecer) [2]:

- ***O(1)****:* Un algoritmo con tiempo de ejecución *en el orden de uno* (o *proporcional a 1*), tiene un tiempo de ejecución *constante*, sin importar el número de datos *n*. Esto es claramente lo ideal al diseñar un algoritmo, pues no importa lo que crezca *n*, el tiempo de ejecución será siempre el mismo. Un ejemplo claro de una operación que es *O(1)* en la práctica, es el acceso directo a un componente de un arreglo: no importa cuántos elementos tenga el arreglo (o sea, no importa el valor de *n*), el tiempo para acceder en forma directa a un componente es siempre el mismo.
- ***O(log(n))****:* Un algoritmo cuyo tiempo de ejecución sea del *orden del logaritmo de n*, será ligeramente más lento a medida que *n* sea mayor. Es un orden muy satisfactorio en la práctica, si puede lograrse. Los algoritmos que generalmente tienen tiempos de *orden logarítmico* son los que resuelven un problema de gran tamaño procediendo a transformarlo en uno más pequeño, dividiéndolo por alguna fracción constante. Por ejemplo, el algoritmo de *búsqueda binaria* en un arreglo ordenado tiene *O(log(n))* en el peor caso.
- ***O(n)****:* Un algoritmo cuyo tiempo de ejecución es del *orden de n* (también se dice que su tiempo de ejecución es *lineal*), es aquél que para cada elemento de entrada realiza una pequeña cantidad de procesos iguales. Un caso típico de un algoritmo cuyo orden es *lineal*, es el de la *búsqueda secuencial en un arreglo*.
- ***O(n*log(n))****:* Se da en algoritmos que resuelven un problema dividiéndolo en pequeños subproblemas, resolviéndolos en forma independiente y combinando después las soluciones. Esta estrategia se conoce como *divide y vencerás*, y será analizada con detalle en lo que resta del curso. El algoritmo de ordenamiento *Quicksort* (en el caso promedio) tiene este rendimiento y está basado en la estrategia *divide y vencerás*.
- ***O(n^2)****:* Un algoritmo con tiempo de ejecución de *orden cuadrático*, sólo tiene utilidad práctica en problemas relativamente pequeños (o sea, con *n* pequeño). El tiempo de ejecución del orden de n^2 suele aparecer en algoritmos que procesan pares de elementos de datos, y la forma típica de estos algoritmos incluye un par de ciclos anidados. Se mencionó ya que todos los métodos de ordenamiento directos tienen ese tiempo de ejecución en el peor caso.
- ***O(n^3)****:* Un algoritmo de tiempo de ejecución de *orden cúbico* tampoco es muy práctico, salvo en casos de problemas muy pequeños. La forma típica de estos algoritmos incluye *tres ciclos anidados*. A modo de ejemplo, es de orden cúbico el tiempo de ejecución del popular algoritmo que permite multiplicar dos matrices entre sí.
- ***O(2^n)****:* Los algoritmos con *orden de ejecución exponencial* son muy poco útiles en la práctica. Sin embargo, aparecen con frecuencia en casos de algoritmos del tipo de *fuerza bruta*, en los que todas las soluciones posibles son investigadas una por una. Suelen aparecer tiempos de este orden en problemas de optimización de soluciones, y en esos casos se considera todo un éxito el poder replantear un algoritmo de modo de lograr tiempos cuadráticos o cúbicos…

Anexo: Temas Avanzados

En general, cada Ficha de Estudios podrá incluir a modo de anexo un capítulo de *Temas Avanzados*, en el cual se tratarán temas nombrados en las secciones de la Ficha, pero que requerirían mucho tiempo para ser desarrollados en el tiempo regular de clase. El capítulo de *Temas Avanzados* podría incluir profundizaciones de elementos referidos a la programación en particular o a las ciencias de la computación en general, como así también explicaciones y demostraciones de fundamentos matemáticos. En general, se espera que el alumno sea capaz de leer, estudiar, dominar y aplicar estos temas aún cuando los mismos no sean específicamente tratados en clase.

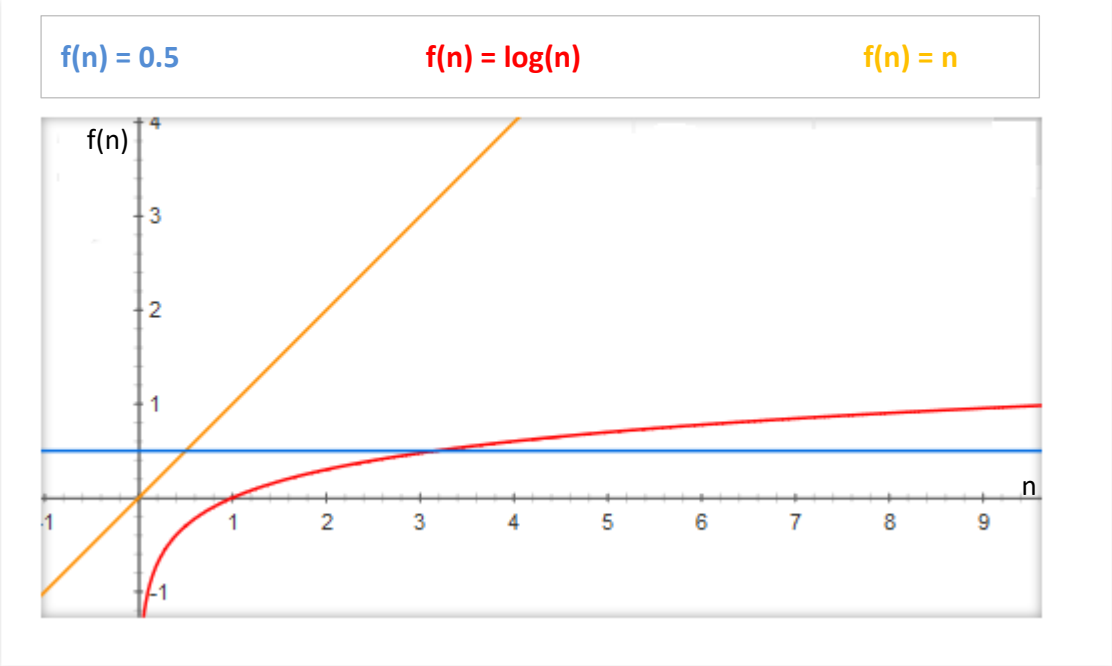
a.) Forma de crecimiento de las funciones clásicas de orden de complejidad.

Las funciones típicas que hemos mostrado en las secciones anteriores se han listado en orden de menor a mayor de acuerdo a la *tasa de variación* de cada una cuando *n* se hace grande o muy grande (que es cuando realmente tiene valor el análisis del comportamiento de un algoritmo). Es decir que para *n* grande o muy grande, se tiene que:

$$O(1) < O(\log(n)) < O(n) < O(n*\log(n)) < O(n^2) < O(n^3) < O(2^n)$$

La gráfica siguiente (desarrollada con el *graficador de funciones de Google*[1]) muestra el comportamiento de las tres primeras. Para la función constante hemos seleccionado graficar **f(n) = 0.5** (en este contexto, tiempo de *ejecución constante* u *O(1)* significa que el algoritmo siempre tendrá el mismo tiempo de ejecución, y no es relevante cuál sea el valor real de esa constante siempre y cuando se trate de un valor razonable para un computador):

Figura 1: Gráfica general de las funciones f(n) = 0.5 - f(n) = log(n) - f(n) = n



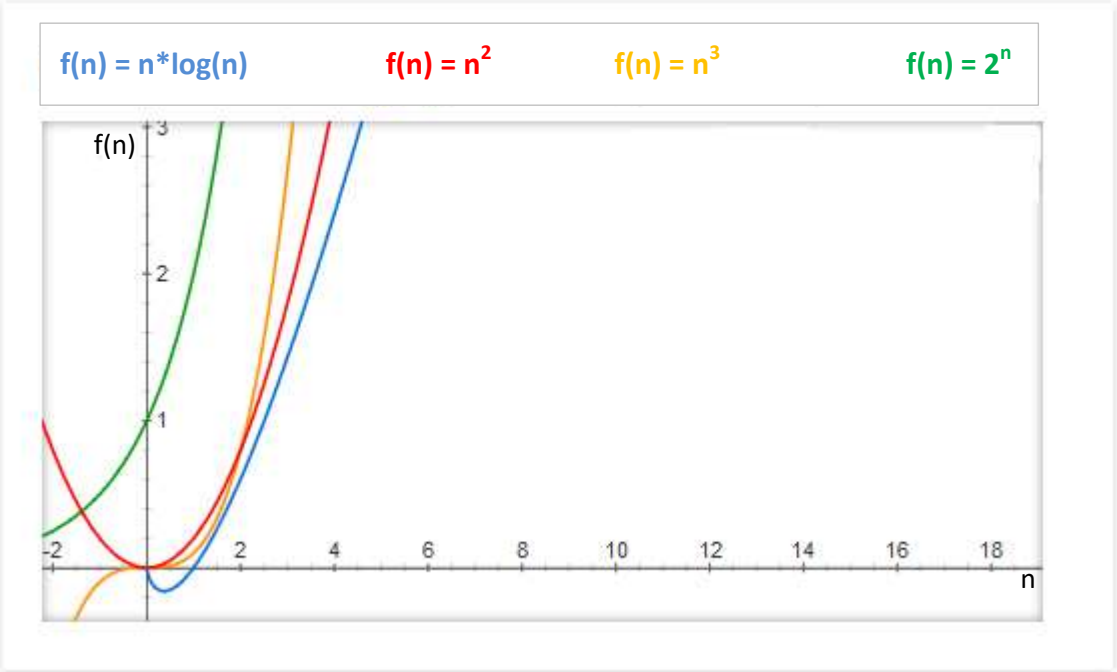
La gráfica anterior muestra que para valores bajos de *n*, las tres curvas se comportan en forma aceptable (si se están representando tiempos de ejecución) y que incluso la *curva del logaritmo* (en color rojo) parece mejor que las otras dos. Pero lo realmente importante es el comportamiento de las tres cuando *n* es grande (en la gráfica es suficiente con mirar lo que ocurre para *n > 3*): en ese caso, la curva de ***f(n) = n*** (en naranja) muestra claramente tiempos mucho mayores a los de las otras dos; y la función f(n) = log(n) se vuelve mayor que **f(n) = 0.5** (en celeste) y permanecerá mayor (ya que la función f(n) = log(n) es monótona creciente).

A su vez, la gráfica que sigue (otra vez: desarrollada con el *graficador de funciones de Google* ya citado) muestra el comportamiento de las últimas cuatro funciones de nuestra tabla. De

[1] Google incluye una serie aplicaciones online para este tipo de cálculos que puede accederse desde la dirección url: https://support.google.com/websearch/answer/3284611?hl=es-US#plotting.

nuevo, para valores pequeños de *n* podrían parecer aceptables los tiempos de respuesta, e incluso pudiera parecer mejor la función *cúbica* (en color naranja) que la *cuadrática* (en color rojo). Pero cuando *n* crece las cosas se ponen en su lugar: claramente la función *exponencial* (en color verde) se hace ridículamente grande para valores muy pequeños de *n* (*n <= 35*, por ejemplo) y se vuelve la mayor de todas, mientras que la cúbica supera a la cuadrática y todas ellas son mayores que **f(n) = n*log(n)** (en celeste)[2]:

Figura 2: Gráfica general de las funciones f(n) = n*log(n) - $f(n) = n^2$ - $f(n) = n^3$ - $f(n) = 2^n$



Lo anterior está mostrando un hecho muy importante de destacar: la *función exponencial* tiene un ritmo o tasa de crecimiento muy violento: para muy pequeños aumentos en el valor de *n* (en el dominio de la función), se obtienen valores de respuesta (en su imagen) muy pero muy grandes. Por lo tanto, si se sabe que un algoritmo tiene rendimiento exponencial en cuanto al tiempo de ejecución, entonces ese algoritmo en la práctica es muy poco aplicable: para valores muy pequeños de *n*, se obtienen tiempos de respuesta asombrosamente altos y ni siquiera una computadora moderna y potente podría llegar a un resultado en un tiempo aceptable (sin exagerar, incluso una computadora muy potente necesitaria *miles de años* para terminar de ejecutar un programa que incluya un número exponencial de pasos…)

Sobre el final de este curso, veremos algunos elementos de la *Teoría de la Complejidad* dentro de la cual los algoritmos con tiempo de ejecución exponencial tienen una importancia fundamental. Por ahora, baste con saber que si para un problema dado *sólo se conocen algoritmos de tiempo de ejecución exponencial*, entonces esos problemas se

[2] Confesamos que hemos hecho una pequeña trampa: la gráfica de la **función cúbica** que se muestra, corresponde en realidad a la función **$f(n) = 0.1*n^3$**, mientras que la gráfica de la curva **cuadrática** corresponde a **$f(n) = 0.2*n^2$**. El motivo fue permitir que la gráfica conjunta de todas las curvas muestre rápidamente la relación de orden para *n* grande, que de otro modo exigiría un gráfico mucho mayor.

designan como *problemas intratables* y son objeto de profundos estudios en el campo de las ciencias de la computación[3].

Créditos

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

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

Bibliografía

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

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

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

[4] Coursera, "Take the world's best courses, online, for free.," 2012. [Online]. Available: https://www.coursera.org/. [Accessed 27 March 2013].

[3] Los usuarios de computadoras poco experimentados tienden a sobre-estimar la capacidad de una computadora moderna para levar a cabo cualquier proceso en forma veloz. Sin embargo, son muy numerosas las situaciones reales y concretas de problemas que necesitan muchísimo tiempo de procesamiento incluso para una super computadora. Una muy buena película de suspenso y espionaje de 1987 llamada *No Way Out* (o *Sin Salida*) [dirigida por *Roger Donaldson* y protagonizada por *Kevin Costner*] utiliza en forma convincente este hecho: un oficial naval estadounidense debe encontrar al asesino de una mujer, pero una fotografía instantánea arruinada (una "polaroid") que ella tenía de este investigador, incriminaría falsamente al oficial. La fotografía es entregada a un experto en computación que aplica sobre ella un *lentísimo proceso* de reconstrucción de imagen, que *llevará muchas horas*, mientras el oficial busca al asesino. Y debe encontrarlo antes que la computadora reconstruya su propia foto…