Ficha 10

Análisis de Algoritmos – Ordenamiento

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.

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. 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*).

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.

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*.

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*").

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))*.

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:

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).

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*).

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*). 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 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):

- ***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…

3.] 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:

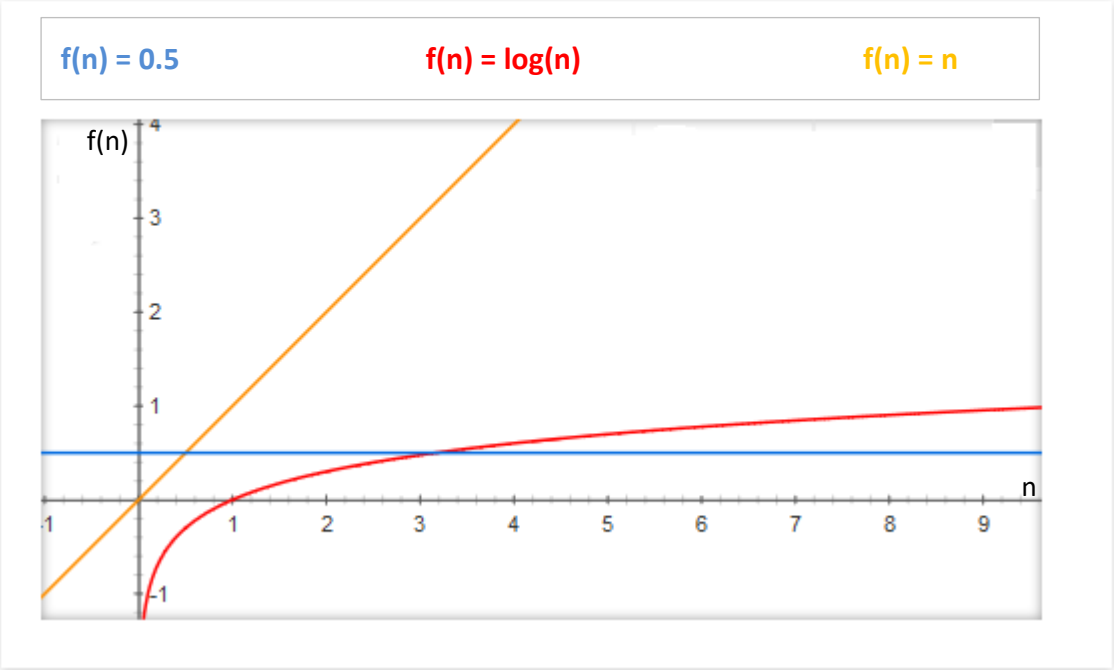
$$O(1) < O(\log(n)) < O(n) < O(n*\log(n)) < O(n^2) < O(n^3) < O(2^n)$$

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

[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.

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

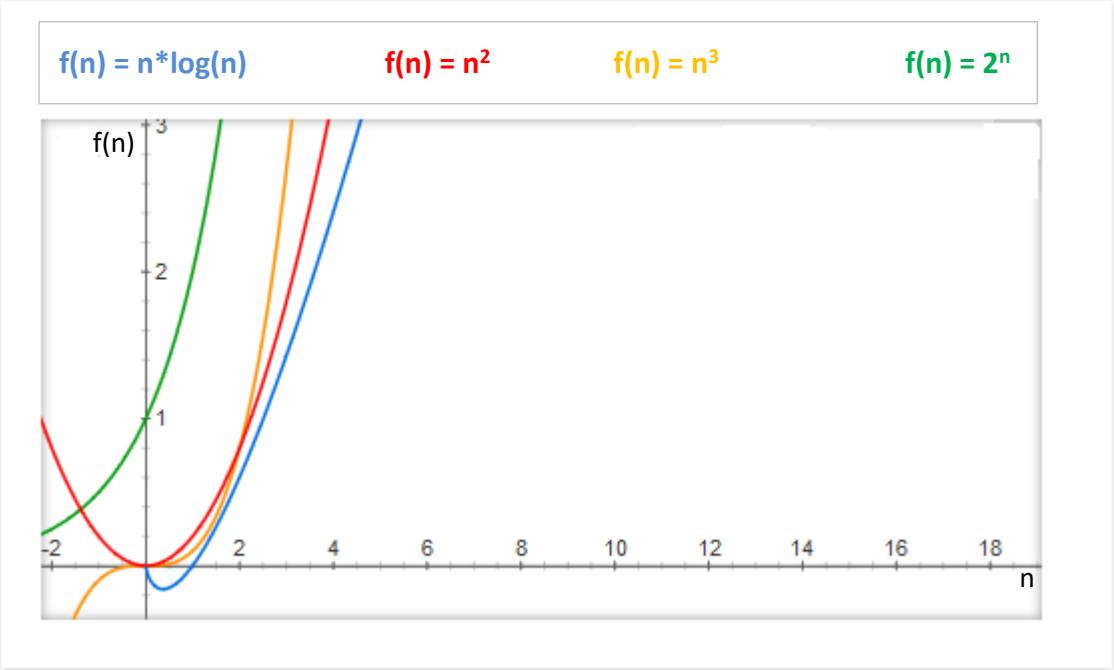


La gráfica anterior muestra que para valores bajos de *n*, las tres curvas se comportan en forma aceptable (si se están representando tiempos de ejecución) y que incluso la *curva del logaritmo* (en color rojo) parece mejor que las otras dos. Pero lo realmente importante es el comportamiento de las tres 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 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]:

[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.

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

4.] Formalización de la notación *Big O*.

La notación *Big O* se usa para indicar un *límite superior* para el comportamiento esperado de un algoritmo en cuanto al tiempo de ejecución o el espacio ocupado o algún otro parámetro[3]. Si se dice que un algoritmo de ordenamiento tiene un tiempo de ejecución en el peor caso de $O(n^2)$, de alguna forma se está diciendo que ese algoritmo *no se comportará*

[3] Parte del desarrollo de esta sección se basa en: Weiss, M. A. (2000). "*Estructuras de Datos en Java*". Madrid: Addison Wesley. ISBN: 84-7829-035-4 (página 103 y siguientes). También se han seguido ideas (sobre todo en el planteo de las gráficas) del material del curso "*Design and Analysis of Algorithms I*" – Stanford University (a cargo de *Tim Roughgarden*, Associate Professor): https://www.coursera.org/courses.

peor que n^2 en cuanto al tiempo de ejecución. Puede decirse que la función *f* que calcula el tiempo de acuerdo al valor de *n*, siempre se mantendrá menor o igual que n^2 multiplicada por alguna constante *c*.

Formalmente, decir que una función *f* está en el orden de otra función *g*, implica afirmar que eventualmente, para cualquier valor suficientemente grande de *n*, la función *f* siempre será *menor o igual* que la función *g* multiplicada por alguna constante *c* mayor a cero. En símbolos:

$$\text{Si} \quad f(n) \text{ es } O(g(n)) \quad \Rightarrow \quad f(n) \leq c*g(n)$$

(para todo valor *n* suficientemente grande y algún *c* > 0)

Para verlo mejor, analicemos la gráfica de la *Figura 3* (en *página 10*) En ella se supone que la función *f* es orden de *g*. A los efectos del ejemplo, no tiene importancia cuáles sean estrictamente las funciones *f* y *g*, sino sólo analizar lo que implica la relación *f(n) = O(g(n))*.

Si *f* es orden *g*, entonces podremos encontrar al menos una constante *c > 0* (que en la gráfica vale 2) tal que a partir de cierto número n_0 los valores de *g* multiplicados por *c* serán siempre mayores o iguales a los valores de *f*. Eso equivale a decir que la nueva función *c*g(n)* será mayor a ***f(n)*** para todo $n > n_0$. En el esquema gráfico, puede verse que a partir del valor n_0 la curva *2 * g(n)* se vuelve siempre mayor que la curva *f(n)*, con lo cual *f(n) = O(g(n))*.

Note que la relación *f(n) = O(g(n))* implica que *f* será menor o igual a *c * g* a partir de cierto valor n_0, y no necesariamente para valores pequeños de *n* (o sea, para valores de $n < n_0$). Es decir, lo que importa es lo que pasará para valores grandes o muy grandes de *n*, que es lo que se conoce como el *comportamiento asintótico de la función*.

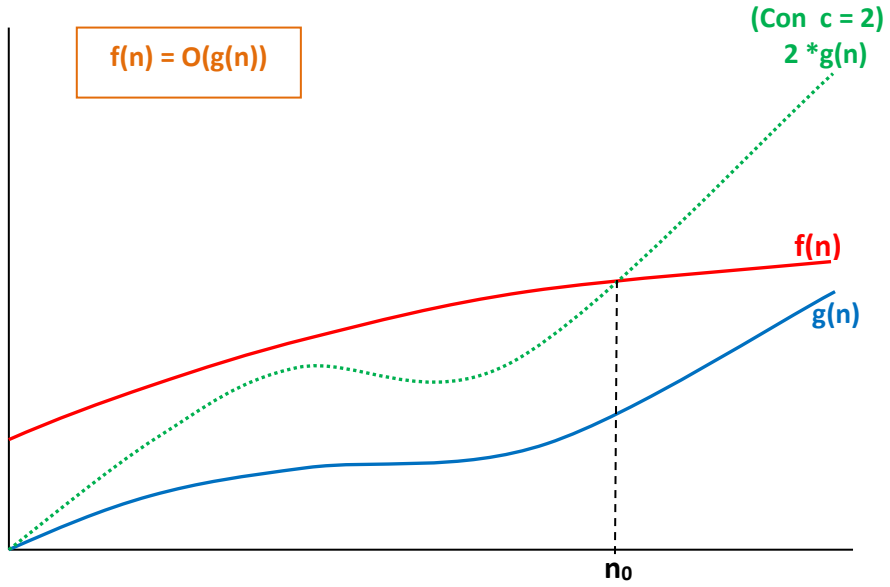
También note que la constante *c > 0* puede ser cualquiera y solo basta con encontrar una. Si puede encontrarse al *menos una constante c > 0* para la cual se pueda probar que *f(n) <= c *g(n)* desde cierto punto n_0 en adelante, entonces *f* es orden de *g*. En la gráfica anterior, también hubiese sido válido suponer *c = 3* o *c = 4*, o cualquier valor de *c > 0* que cumpla la relación.

Si se observa con atención, lo que se está haciendo es intentar determinar la forma de variación de la función *f* de acuerdo a la forma en que varía otra función ***g*** ya conocida. Pero como en rigor no se usa necesariamente la propia ***g***, sino alguna función múltiplo de ***g*** (porque se multiplica a ***g*** por la constante *c*), entonces la expresión *O(g(n))* en realidad está implicando *una familia de funciones* cuyo comportamiento es caracterizado por el comportamiento de ***g***.

En ese sentido, la expresión *O(g(n))* es lo que se conoce como un *orden de complejidad*: un conjunto o familia de funciones que se comportan asintóticamente de la misma forma. La función más característica de ese conjunto (la más simple, sin constantes ni términos independientes adicionales) es la que normalmente se usa para expresar la relación de orden, y se suele designar como *función representante* del conjunto (o *función característica* o también *función dominante*). Así, cuando decimos que la *búsqueda secuencial* tiene un tiempo de ejecución *t(n) = O(n)*, estamos diciendo que el tiempo *t* tiene la misma forma de variación asintótica que el conjunto de funciones representadas por *g(n) = n* (que es en este caso es la *función característica* de ese orden de complejidad). Como todas en *O(n)* se comportan de la misma forma que *g(n) = n*, no tiene relevancia entonces incluir constantes y

términos independientes en la expresión de orden. No es necesario que el analista agregue detalles: decir (por ejemplo) $t(n) = O(2n + 5)$ es asintóticamente lo mismo que $t(n) = O(n)$.

Figura 3: Idea general del significado de f(n) = O(g(n)).



Y otro detalle a considerar: al expresar una relación de orden en notación *Big O*, se está indicando una *cota superior* para el comportamiento de una función *f* en términos del comportamiento de la familia de funciones *O(g)*, pero en general el analista buscará la *menor cota superior para f*. Está claro que si el tiempo de ejecución *t* de la búsqueda secuencial es $t(n) = O(n)$, es también cierto que $t(n) = O(n^2)$ (ya que como vimos $O(n) < O(n^2)$) y si se prosigue con ese argumento, resulta que prácticamente todos los algoritmos conocidos son $O(2^n)$ (ya que la función exponencial o algún múltiplo de ella siempre será mayor que las demás para *n* grande). Pero al expresar una función *f* en notación *Big O*, lo que se quiere obtener es la familia de funciones que sirva como *mejor cota superior* para el comportamiento de *f*, y no cualquier familia que sea mayor o igual que *f* (ya que en este caso el análisis sería demasiado amplio y vago).

Todo lo anterior ayuda en el análisis de algoritmos: muchas veces un programador tiene un conocimiento estimado (o incluso intuitivo) del comportamiento de la función *f* que predice (por ejemplo) el tiempo de ejecución de un algoritmo en el peor caso, pero desconoce la forma analítica precisa de esa función (o a los efectos del análisis del peor caso no requiere de esa forma precisa). Si el programador puede probar que su función desconocida *f* es del orden de otra función conocida (y posiblemente más simple) *g*, entonces tendrá una *cota superior* para su función *f*: Sabrá que su algoritmo *nunca será peor* que *g* multiplicada por una constante dada. Esto puede parecer muy vago, pero en el análisis asintótico tiene mucha importancia: al fin y al cabo, saber que nuestro algoritmo nunca será peor que $n*log(n)$ (por ejemplo) nos garantiza que ese algoritmo será subcuadrático incluso en el peor caso, aún si ignoramos los coeficientes precisos de la *verdadera* función *f*.

A partir de estos elementos, surgen algunas relaciones de orden básicas que debemos tener presentes, y que de alguna manera ayudan a justificar que en notación *Big O* se puede prescindir de las constantes y quedarse solo con el término o función dominante. Por otra parte, estas relaciones ayudan a poder estimar en forma rápida el comportamiento

asintótico de un algoritmo sin entrar permanentemente en la necesidad de demostrar la relación de orden. No es necesario que estudie ni recuerde las demostraciones, pero viene bien tenerlas a mano:

---

a.) Cualquier función polinómica en *n* de grado *k*, es orden n^k. Simbólicamente:

$$f(n) = a_k n^k + a_{k-1} n^{k-1} + \ldots + a_2 n^2 + a_1 n + a_0 = O(n^k)$$

Demostración: si tomamos $c = |a_k| + |a_{k-1}| + |a_{k-2}| + \ldots + |a_2| + |a_1| + |a_0|$ y $n_0 = 1$, entonces debemos probar que para todo $n > n_0 = 1$, se cumple que $f(n) <= c * n^k$. O sea:

$f(n) = a_k n^k + a_{k-1} n^{k-1} + a_2 n^2 + a_1 n + a_0$
$\Rightarrow$ $f(n) <= |a_k| * n^k + |a_{k-1}| * n^{k-1} + |a_{k-2}| * n^{k-2} + \ldots + |a_2| * n^2 + |a_1| * n + |a_0|$
$\Rightarrow$ $f(n) <= |a_k| * n^k + |a_{k-1}| * n^k + |a_{k-2}| * n^k + \ldots + |a_2| * n^k + |a_1| * n^k + |a_0| * n^k$
$\Rightarrow$ $f(n) <= (|a_k| + |a_{k-1}| + |a_{k-2}| + \ldots + |a_2| * + |a_1| * + |a_0|) * n^k$
$\Rightarrow$ **$f(n) <= c * n^k$** que es lo que se quería probar.

Este resultado muestra que si sabemos que nuestra función *f* en *n* (desconocida) es polinómica de grado *k*, entonces *podemos prescindir de todas las constantes y de todos los términos que no sean de grado k*, y quedarnos sólo con el término dominante n^k, simplificando el análisis.

---

b.) Sea $f(n) = 2^{n+10}$ y $g(n) = 2^n$. Entonces $2^{n+10} = O(2^n)$

Demostración: Debemos tomar dos constantes *c* y n_0 para las que se cumpla que $2^{n+10} <= c * 2^n$. Sabemos que:

$$2^{n+10} = 2^n * 2^{10} = 2^n * 1024$$

Por lo que si tomamos $c = 1024$ sabemos que para $n0 = 1$, la relación postulada se cumple.

Procediendo en forma similar, se puede probar que $2^{n+k} = O(2^n)$ para cualquier $k >= 1$, lo cual también simplifica el análisis.

---

c.) En notación *Big O*, la base de los logaritmos puede obviarse, ya que la función $\log_b(n)$ es $O(\log_2(n))$ para cualquier base $b > 1$. Simbólicamente:

Sea $f(n) = \log_b(n)$ y $g(n) = \log_2(n) = \log(n)$. Entonces $\log_b(n) = O(\log(n))$ [con $b > 1$]

Demostración: Por defecto, asuma que la base 2 es implícita (o sea: asuma que $\log_2(n) = \log(n)$). Intentamos entonces probar que $\log_b(n) <= c * \log(n)$ para alguna constante cualquiera $c > 0$.

Sea $\log_b(n) = k$. Entonces (por definición de logaritmo): $b^k = n$. Tomemos la constante $p = \log(b)$. Entonces $2^p = b$.

Entonces, de los dos hechos anteriores resulta:

$n = b^k = (2^p)^k$
$n = 2^{pk}$
$\log(n) = p*k$
$\log(n) = p * \log_b(n)$
$\log_b(n) = 1/p * \log(n)$

Por lo tanto, si tomamos $c >= 1/p$, entonces $\log_b(n) <= c * \log(n)$ lo que prueba la relación 3. Hemos visto en fichas anteriores que este hecho resulta sumamente práctico cuando la relación de orden analizada incluye logaritmos: otra vez, se simplifica el análisis.

5.] Algunas consideraciones prácticas.

De ahora en adelante, se espera que el alumno sea capaz de al menos "intuir" una relación de orden para un algoritmo en el peor caso (es decir, dado un algoritmo, ser capaz de dar en notación *Big O* un orden para el peor caso de ese algoritmo en tiempo o espacio). Para ganar destreza en esa tarea, van algunos consejos prácticos:

i. Dado el algoritmo que debe analizar (expresado en forma de diagrama de flujo, o de pseudocódigo, o de programa fuente o incluso expresado en forma coloquial o intuitiva), tenga en claro *cuál será el factor de eficiencia* que quiere analizar: tiempo de ejecución o memoria empleada, por ejemplo.

ii. En ese algoritmo determine con exactitud el tamaño del problema (o volumen de datos a procesar). En muchos casos esto es simple de hacer (el tamaño *n* de un arreglo si se quiere ordenar ese arreglo o buscar en él) pero en algunos casos no es tan simple ni tan obvio (aunque no enfrentaremos casos extremos por ahora…) En algunos casos, el tamaño puede venir expresado con dos o más variables (por caso, si se pide procesar dos arreglos de tamaños *m* y *n* respectivamente), y la expresión de orden final podría estar a su vez basada en dos o más variables (no es extraño encontrar expresiones de la forma $t(v, w) = O(v*log(w))$ siendo *v* y *w* dos variables, o casos como $t(n, m) = O(m * n^2)$, por ejemplo).

iii. En el algoritmo analizado, determine claramente cuál es la *instrucción crítica* que ese algoritmo lleva a cabo. La instrucción crítica es la instrucción o bloque de instrucciones que se ejecuta más veces a lo largo de toda la corrida, y si está analizando el tiempo de ejecución del algoritmo, entonces la acumulación de los tiempos de ejecución de la operación crítica es lo que lleva al algoritmo a su tiempo final. En un ordenamiento, por lo general, la instrucción que más se ejecuta es la comparación (y lo mismo en una búsqueda) por lo que esa será la instrucción crítica.

iv. En este momento, tenga en cuenta que no es lo mismo realizar un conteo exhaustivo y riguroso de operaciones críticas, que hacer un análisis asintótico. Si lo que necesita es un *conteo exhaustivo*, entonces deberá extremar el análisis y tratar de expresar una fórmula con toda rigurosidad y detalle de constantes, términos no dominantes y términos independientes que indique cuántas operaciones críticas se ejecutan exactamente para un valor dado de *n*. Pero si lo que necesita es un *análisis asintótico*, entonces siga leyendo los puntos que vienen a continuación…

v. Para el *análisis asintótico*, si ya tiene una expresión de *conteo exhaustiva* identifique el *término dominante* en ella, y limítese a ese término. No diga que un algoritmo es $O(n^3 + n^2)$: por la *relación a.)* vista en *página 11*, en este caso bastará con decir $O(n^3)$. Entienda que para *n* suficientemente grande, el término n^2 será técnicamente despreciable frente a n^3. Si no tiene una expresión o fórmula exhaustiva, identifique en forma general la estructura del algoritmo que contiene a la operación crítica y deduzca en forma intuitiva. Valen los siguientes consejos para situaciones comunes, independientemente de otras situaciones que deberá resolver con otros criterios:

- Si la operación crítica es (por ejemplo) una comparación y está contenida dentro de dos ciclos anidados de *n* iteraciones cada uno, entonces su algoritmo es $O(n^2)$.
- Una operación crítica de tiempo de ejecución constante ($O(1)$) incluida dentro de *k* ciclos anidados de *n* iteraciones cada uno, tendrá tiempo de ejecución $O(n^k)$.
- Si su algoritmo consta de varios bloques de instrucciones independientes entre sí, entonces por la misma *relación a.)* de página 11 su algoritmo tendrá un tiempo de ejecución en el orden del que corresponda al bloque con mayor tiempo. Así, si su algoritmo tiene un primer bloque que ejecuta en tiempo $O(n)$ y luego dos bloques más

que ejecutan en tiempos $O(n^2)$ y $O(log(n))$, entonces el tiempo completo sería $t(n) = O(n) + O(n^2) + O(log(n))$ pero esto es asintóticamente igual a $O(n^2)$, por lo que $t(n) = O(n^2)$.

- Si todo su algoritmo está compuesto sólo por un bloque de instrucciones de tiempo constante (asignaciones o condiciones, por ejemplo) sin ciclos ni procesos ocultos dentro de una función, entonces todo el algoritmo ejecuta en tiempo constante $t(n) = O(1)$ (ya que sería $t(n) = O(1) + O(1) + \ldots + O(1)$ que es lo mismo que $t(n) = O(1)$ (por la misma relación a.) de *página 11*).
- Si su algoritmo se basa en tomar un bloque de *n* datos, dividirlo por 2, aplicar una operación de tiempo constante y luego procesar sólo una de las mitades de forma de volver a dividirla y continuar así hasta no poder hacer otra división, entonces su algoritmo ejecuta en tiempo $t(n) = O(log_2(n))$ que por la *relación c.* de *página 11*, es lo mismo que $t(n) = O(log(n))$ sin importar la base del logaritmo.

vi. No incluya *constantes* dentro de la expresión de orden, salvo aquellas que indiquen el exponente específico del término dominante. En general no dirá $O(2n)$ sino simplemente $O(n)$. La notación O le permite rescatar de un solo golpe de vista la forma de variación del término dominante, y en esa variación son en general despreciables las constantes.

vii. No se preocupe por la base del logaritmo si su expresión de orden incluye logaritmos. La relación 3 prueba que la base no es relevante en análisis asintótico. Y en todo caso, la base del logaritmo es ella misma una constante.

En el *punto iv* de estas consideraciones prácticas, hemos indicado que no es lo mismo un *conteo exhaustivo* que un análisis de *comportamiento asintótico*. El primero es mucho más riguroso. El segundo es más amplio. Tomemos por ejemplo el ya conocido *Ordenamiento de Selección Directa* para un arreglo *v* de *n* componentes, que en general se plantea así:

```
n = len(v)
for i in range(n-1):
    for j in range(i+1, n):
        if v[i] > v[j]:
            v[i], v[j] = v[j], v[i]
```

Si queremos hacer un *conteo exhaustivo* del número de operaciones críticas (en este caso, las *comparaciones*) que este algoritmo ejecuta, debemos ver que el proceso consiste en tomar la primera casilla del arreglo (designada como *pivot*), comparar su contenido con los *(n-1)* casilleros restantes y dejar el menor valor en la casilla *pivot*. Luego se toma la segunda casilla como *pivot*, se compara contra las *(n-2)* restantes, y se vuelve a dejar el menor de lo que quedaba del arreglo en la casilla *pivot*. Así, se hacen *(n-1)* pasadas, y cuando el *pivot* sea la casilla *(n-2)* se hará una única y última comparación más contra la casilla *(n-1)* y el arreglo quedará ordenado. Pero esto significa que la *cantidad de comparaciones* a realizar sale de:

```
Pasada 1:      n-1 comparaciones
Pasada 2:      n-2 comparaciones
Pasada 3:      n-3 comparaciones
...            ...
Pasada n-2:    2 comparaciones
Pasada n-1:    1 comparación
```

Lo anterior implica que el total de comparaciones $t(n)$ a ejecutar para el total *n* de casilleros en el arreglo, será igual a la suma:

$$t(n) = 1 + 2 + \ldots + (n-3) + (n-2) + (n-1)$$

que se puede demostrar que es igual a:

$$t(n) = (n-1) * n / 2$$

y entonces:

$$t(n) = (n^2 - n) / 2$$

que tiene forma claramente cuadrática... Esto quiere decir que el algoritmo de *Selección Directa* hará una cantidad de comparaciones que será *exactamente* función de *n al cuadrado*, y por lo tanto el tiempo *t* demorado por ese algoritmo en ordenar el arreglo será *proporcional a n al cuadrado*. Hasta aquí, hemos desarrollado un *conteo exhaustivo* de operaciones críticas y la función obtenida expresa con todo detalle ese conteo. Si ahora se nos pide un análisis de *comportamiento asintótico*, el camino está allanado porque ya tenemos la fórmula precisa del *conteo exhaustivo*. El quinto punto de la lista de recomendaciones que hemos mostrado antes, nos lleva directamente a la expresión de orden: el tiempo de ejecución de este algoritmo, en notación *Big O*, es $t(n) = O(n^2)$, prescindiendo de constantes y términos no dominantes. Pero aún sin tener la fórmula rigurosa del *conteo exhaustivo*, podríamos haber llegado a la misma conclusión simplemente analizando el código fuente (como también se indica en la quinta recomendación práctica): dos ciclos anidados de aproximadamente *n* repeticiones cada uno, y una instrucción condicional contenida en el ciclo más interno: $O(n^2)$.

Un análisis similar (tanto *exhaustivo* como *asintótico*) permite deducir que los otros algoritmos de *ordenamiento simples* o *directos* (como la ordenación por *Intercambio Directo* y la ordenación por *Inserción Directa*) también tienen un *comportamiento cuadrático* en el *peor caso*. El hecho es que el algoritmo de *Selección Directa* siempre hará la cantidad de comparaciones calculada más arriba, pero los otros dos podrían realizar una cantidad algo menor en *casos más favorables.* En otras palabras, *si sólo se consideran comparaciones* (y no por ejemplo la cantidad de veces que se hacen efectivamente intercambios), el método de *Selección* siempre cae en su peor caso, pero los otros dos podrían tener casos muy favorables dependiendo del estado inicial del arreglo. Los tres son de comportamiento cuadrático, y *asintóticamente pertenecen al mismo orden de complejidad*, pero las constantes que describen en forma analítica la función de *conteo exhaustivo* pueden ser diferentes.

En la siguiente tabla se muestran los tiempos de ejecución en notación *Big O* (comportamiento asintótico) para el peor caso de los algoritmos de ordenamiento vistos (o que veremos más adelante), más algunas consideraciones respecto de casos promedio o incluso mejores casos si esas situaciones son relevantes:

Tabla 2: Comportamiento asintótico de los principales algoritmos de ordenamiento.

Algoritmo	Tiempo	Observaciones
Burbuja (Intercambio Directo)	$O(n^2)$ (peor caso)	Mejor caso: O(n) si el arreglo está ya ordenado.
Selección Directa	$O(n^2)$ (peor caso)	Siempre...
Inserción Directa	$O(n^2)$ (peor caso)	
Quicksort	**O(n*log(n)) (caso medio)**	Peor caso (depende de la implementación): **$O(n^2)$**.
Heapsort	O(n*log(n))	Caso medio: también O(n*log(n)).
Shellsort	$O(n^{1.5})$	Para la serie de incrementos mostrada en clase.

Los siguientes son problemas y/o algoritmos muy comunes para los que hacemos un breve análisis asintótico del *peor caso* en notación *Big O*. Intente rápidamente (en lo posible, sin mirar el análisis que hacemos para cada uno) hacer su propia estimación, a modo de ejercicio:

- *Búsqueda secuencial de un valor x en un arreglo desordenado de n componentes*: La operación crítica es la comparación, y en el peor caso debe hacerse n veces (si x no está en el arreglo o está muy al final). Entonces para el peor caso resulta *t(n) = O(n)*.
- *Búsqueda secuencial de un valor x en un arreglo ordenado de n componentes*: La operación crítica es la comparación, y en el peor caso (otra vez) debe hacerse *n* veces (si *x* no está en el arreglo y es mayor a todos los elementos del mismo). Entonces para el peor caso también resulta *t(n) = O(n)*.
- *Búsqueda binaria de un valor x en un arreglo de n elementos (por supuesto, ordenado)*: La operación crítica es la comparación, y en el peor caso debe hacerse $log_2(n)$ veces (si *x* no está en el arreglo), ya que en ese caso se hacen $log_2(n)$ divisiones por 2, y una comparación por cada división. Entonces para el peor caso *t(n) = O(log(n))*.
- *Conteo de la cantidad de valores de un arreglo de tamaño n que son mayores al promedio de todos los valores del arreglo*: Hay un primer proceso en el cual deben hacerse n acumulaciones para calcular el promedio (hasta aquí, O(n)). Y luego un segundo proceso en el cual se hace n comparaciones para determinar cuántos valores son mayores que el promedio (con lo que se tiene otra vez O(n)). El tiempo total será *t(n) = O(n) + O(n) = O(2n) = O(n)* con lo que *t(n) = O(n)*.
- *Multiplicación de matrices (suponga, para simplificar, que las matrices son cuadradas y del mismo orden n)*: Sin entrar en demasiados detalles, el algoritmo clásico emplea tres ciclos for de n iteraciones cada uno, y en cada giro realiza la acumulación de un producto. Por lo tanto, resulta $t(n) = O(n^3)$.
- *Conteo de las frecuencias de aparición de los n números de un conjunto, que pueden venir repetidos, sin conocer el rango en el que vienen esos números y usando un arreglo de objetos de conteo*: Cada uno de los *n* números debe buscarse secuencialmente en un arreglo que en el peor caso también llegará a tener *n* casilleros si todos los números fuesen diferentes. Como una búsqueda secuencia ejecuta en tiempo *t(n) = O(n)*, y debemos hacer *n* búsquedas, tenemos un tiempo final *t(n) = n * O(n)* que es lo mismo que $t(n) = O(n * n) = O(n^2)$.
- *Conteo de las frecuencias de aparición de los n números de un conjunto, que pueden venir repetidos, conociendo el rango en el que vienen esos números y usando un arreglo de conteo directo*: Cada uno de los *n* números debe contarse en un arreglo de acceso directo (cada número se cuenta en la casilla cuyo índice coincide con el mismo número). Como un conteo de ese tipo ejecuta en tiempo constante *t(n) = O(1)*, y debemos hacer *n* conteos, tenemos un tiempo final *t(n) = n * O(1)* que es lo mismo que *t(n) = O(n)*.
- *Fusión de dos arreglos ordenados (tamaños m y n) en un tercer arreglo ordenado*: En este caso, hay dos arreglos de entrada con tamaños *m* y *n* y hay que procesar todo el conjunto, por lo cual el tamaño del problema es *m + n*. El algoritmo esencial para hacer esta fusión, genera un arreglo de tamaño *m + n* ordenado. Se recorren los dos arreglos originales, cada uno con su índice, se comparan dos elementos (uno de cada vector) y el menor se lleva al arreglo de salida. Como se hace una comparación por cada uno de los *m + n* casilleros del arreglo de salida, el tiempo total resultante es *t(n) = O(m + n)*. Note que nada impide que la expresión de orden esté basada en dos o más variables.
- *Inserción y eliminación de un elemento en una pila o en una cola*: Sin importar cuántos elementos tenga la pila o la cola, la inserción y la eliminación proceden en tiempo constante (en nuestro caso, ambas estructuras han sido implementadas sobre un arreglo en Python, y en ambos casos, insertar o eliminar el elemento del frente o del fondo se hace en tiempo constante). Por lo tanto, para todas las operaciones pedidas, el tiempo de ejecución resulta ser *t(n) = O(1)*.

6.] Otras notaciones típicas del análisis de algoritmos.

En ocasiones, se desea poder indicar no ya un *límite o cota superior*, sino (por ejemplo) un *límite o cota inferior* para una función *f* dada. Para ese y otros casos, existen otras notaciones de orden (que solo citaremos a modo documentativo, aunque no usaremos frecuentemente):

Si se quiere expresar que el tiempo de ejecución o el espacio de memoria ocupado por un algoritmo *no se comportará mejor* que cierta función *g*, se usa la notación *Omega (Ω)*. Si estamos analizando el tiempo de ejecución de un algoritmo, entonces al decir que el tiempo de ese algoritmo no se comportará mejor que cierta función *g*, queremos decir que los valores de tiempo de ejecución estarán siempre *por arriba* (o a lo sumo serán iguales) de los calculados para esa función límite. Así, por ejemplo, puede verse que si se considera la cantidad de comparaciones, el algoritmo de *Selección Directa* no sólo es $O(n^2)$, sino también $\Omega(n^2)$ (léase: *Omega n cuadrado*): simplemente, no lo hará ni mejor ni peor que algún múltiplos o submúltiplo de n^2.

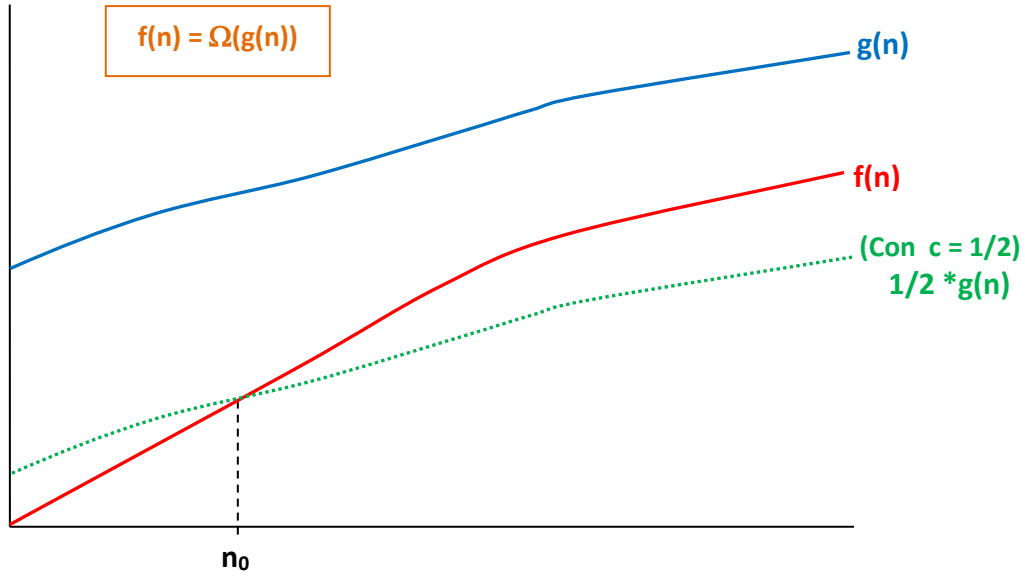
Formalmente, la función *f* que mide el tiempo o el espacio (o cualquier otro factor) de un algoritmo de acuerdo a los valores de *n*, es $\Omega(g(n))$ si se ajusta a la siguiente descripción:

$$\text{si } f(n) \text{ es } \Omega(g(n)) \Rightarrow f(n) \geq c*g(n)$$

(para n suficientemente grande y algún $c > 0$)

Gráficamente, si *f* es *omega g*, entonces podremos encontrar al menos una constante *c* mayor a cero para la cual los valores *f(n)* serán siempre mayores o iguales a *c * g(n)*, a partir de cierto número n_0:

Figura 4: Idea general del significado de f(n) = Ω(g(n)).



Por otra parte, si se quiere indicar que una función *f* se comporta tanto por arriba como abajo *de la misma forma* que múltiplos de otra función *g*, se puede usar también la notación *Theta (Θ)*: si *f(n)* es *Θ(g(n))*, significa que *f* puede acotarse tanto superior como inferiormente por múltiplos de *g*. En otras palabras:

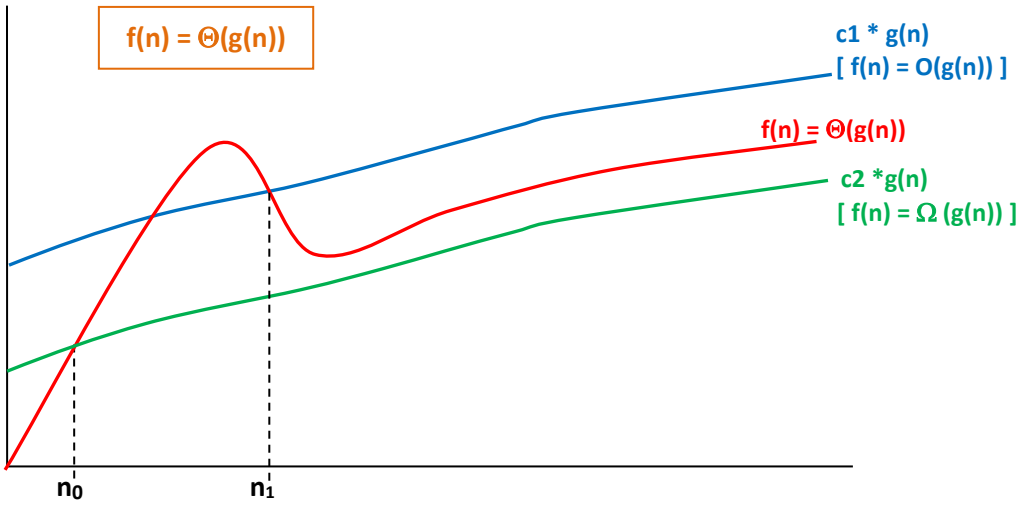
$$\text{Si } f(n) \text{ es } \Theta(g(n)) \Rightarrow f(n) \text{ es } O(g(n)) \text{ y } f(n) \text{ es } \Omega(g(n))$$

(para n suficientemente grande)

La notación *Theta* es muy precisa: no solo nos proporciona una cota superior para el análisis asintótico de una función, sino también *que también nos garantiza que esa cota superior es la mejor posible*. Note que este es el caso del algoritmo de Selección Directa respecto a $g(n) = n^2$ al considerar comparaciones: el algoritmo no será mejor que una función cuadrática y una función cuadrática es la mejor aproximación que puede darse tanto superior como inferior. Se dice entonces que ese algoritmo es $\Theta(n^2)$ (y se lee: *Theta n cuadrado*).

Gráficamente, si *f* es *Theta g*, entonces la gráfica de *f* podrá "encerrarse" entre dos curvas proporcionales a *g*, para dos constantes *c1* y *c2* diferentes:

Figura 5: Idea general del significado de f(n) = Θ(g(n)).



Finalmente, otra notación que también suele usarse, es la notación *o()* (léase: *o minúscula* o *little o*). Esta notación es similar a *O()*, pero considerando *sólo relación de menor estricto* (sin el signo igual). Es útil cuando se quiere indicar una función *estrictamente superior* para el comportamiento de un algoritmo. Así, si podemos decir que un algoritmo dado es (por ejemplo) $O(n^{1.5})$ en cuanto a cierta cantidad de operaciones críticas, entonces se puede indicar también a ese algoritmo como $o(n^2)$: es una forma elegante (aunque menos precisa) de decir que el algoritmo analizado es *subcuadrático*... En símbolos:

$$\text{Si } f(n) \text{ es } o(g(n)) \quad \Rightarrow \quad f(n) < c*g(n)$$

(para n suficientemente grande y algún $c > 0$)

Con el siguiente cuadro mostramos un resumen de las cuatro notaciones (tomado y adaptado del libro *Estructuras de Datos en Java*, de *Mark Allen Weiss – página 116*):

Tabla 3: Resumen de notaciones típicas del análisis de algoritmos.

Expresión	*Significado*
f(n) es O(g(n))	El crecimiento de f(n) es ≤ que el crecimiento de g(n)
f(n) es Ω(g(n))	El crecimiento de f(n) es ≥ que el crecimiento de g(n)
f(n) es Θ(g(n))	El crecimiento de f(n) es = que el crecimiento de g(n)
f(n) es o(g(n))	El crecimiento de f(n) es < que el crecimiento de g(n)

7.] Métodos de ordenamiento. Clasificación tradicional.

Es claro que pueden existir muchos algoritmos diferentes para resolver el mismo problema, y el problema de la ordenación de un arreglo es quizás el caso emblemático de esa situación. Podemos afirmar que existen varias docenas de algoritmos diferentes para lograr que el

contenido de un arreglo se modifique para dejarlo ordenado de menor a mayor, o de mayor a menor. Muchos de esos algoritmos están basados en ideas intuitivas muy simples, y otros se fundamentan en estrategias lógicas muy sutiles y no tan obvias a primera vista. En general, se suele llamar métodos simples o métodos directos a los del primer grupo, y métodos compuestos o métodos mejorados a los del segundo, aunque la diferencia real entre ambos no es sólo conceptual, sino que efectivamente existe una diferencia de rendimiento muy marcada en cuanto al tiempo que los métodos de cada grupo demoran en terminar de ordenar el arreglo. Más adelante nos introduciremos en la cuestión del análisis comparativo del rendimiento de algoritmos, y justificaremos formalmente la diferencia entre los dos grupos, pero por ahora nos concentraremos sólo en la clasificación de los algoritmos y el funcionamiento de cada uno de ellos.

Tradicionalmente, como dijimos, los métodos de ordenamiento se suelen clasificar en los dos grupos ya citados, y en cada grupo se encuentran los siguientes:

Métodos Simples o Directos	**Métodos Compuestos o Mejorados**
Intercambio Directo (Burbuja)	Método de Ordenamiento Rápido (*Quicksort*) [C. Hoare - 1960]
Selección Directa	Ordenamiento de Montículo (*Heapsort*) [J. Williams – 1964]
Inserción Directa	Ordenamiento por Incrementos Decrecientes (*Shellsort*) [D. Shell – 1959]

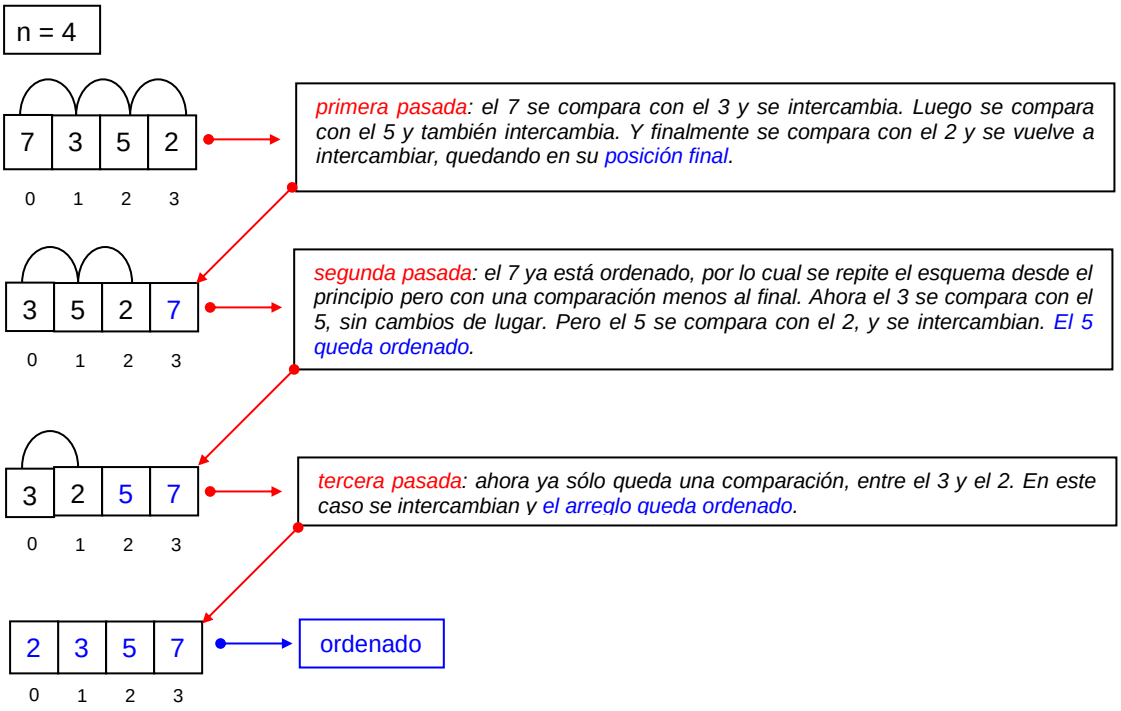
En principio, los métodos clasificados como Simples o Directos son algoritmos sencillos e intuitivos, pero de mal rendimiento si la cantidad n de elementos del arreglo es grande o muy grande, o incluso si lo que se desea es ordenar un archivo en disco. Aquí, mal rendimiento por ahora significa "demasiada demora esperada". En cambio, los métodos presentados como Compuestos o Mejorados tienen un muy buen rendimiento en comparación con los simples, y se aconsejan toda vez que el arrreglo sea muy grande o se quiera ordenar un conjunto en memoria externa. Si el tamaño n del arreglo es pequeño (no más de 100 o 150 elementos), los métodos simples siguen siendo una buena elección por cuanto su escasa eficiencia no es notable con conjuntos pequeños. Note que la idea final, es que cada algoritmo compuesto mostrado en esta tabla representa en realidad una mejora en el planteo del algoritmo simple de la misma fila, y por ello se los llama "métodos mejorados". Así, el algoritmo Quicksort es un replanteo del algoritmo de intercambio directo (más conocido como ordenamiento de burbuja) para eliminar los elementos que lo hace ineficiente. Paradójicamente, el ordenamiento de burbuja o bubblesort es en general el de peor rendimiento para ordenar un arreglo, pero ha dado lugar al Quicksort, que en general es el de mejor rendimiento conocido.

8.] Funcionamiento de los métodos de ordenamiento simples o directos.

Haremos aquí una breve descripción de las estrategias de funcionamiento de los tres métodos directos. Más adelante, mostraremos un análisis de rendimiento basado en calcular la cantidad de comparaciones que estos métodos realizan. En todos los casos, suponemos un arreglo v de n elementos, y también suponemos que se pretende ordenar de menor a mayor. Hemos incluido un modelo llamado *TSB-Ordenamiento*, con una clase *Arreglo* que contiene un atributo de tipo arreglo de valores *int*. En los métodos de esa clase se implementan todos los algoritmos que aquí se presentan.

- ***Ordenamiento por Intercambio Directo (bubblesort):*** la idea esencial del método de es que cada elemento en cada casilla v[i] se compara con el elemento en v[i+1]. Si este último es mayor, se intercambian los contenidos. Se usan dos ciclos anidados

para conseguir que incluso los elementos pequeños ubicados muy atrás, puedan en algún momento llegar a sus posiciones al frente del arreglo. Gráficamente, supongamos que el tamaño del arreglo es n = 4. El método procede así:

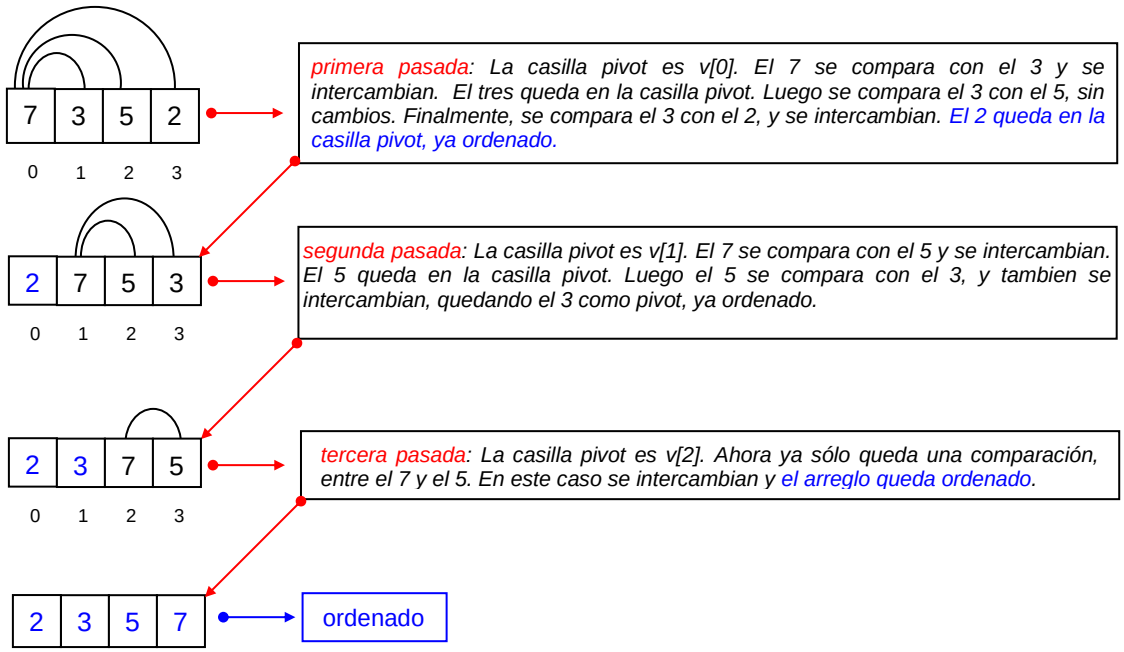


Puede verse que si el arreglo tiene n elementos, serán necesarias a lo sumo n-1 pasadas para terminar de ordenarlo en el peor caso, y que en cada pasada se hace cada vez una comparación menos que en la anterior Se puede probar que esto lleva a un tiempo de ejecución $O(n^2)$ en el peor caso. Otro hecho notable es que el arreglo podría quedar ordenado antes de la última pasada. Por ejemplo, si el arreglo original hubiese sido [2 – 3 – 7 – 5], entonces en la primera pasada el 7 cambiaría con el 5 y dejaría al arreglo ordenado. Para detectar ese tipo de situaciones, el algoritmo conocido como Burbuja Mejorado agrega una variable a modo de bandera de corte: si se detecta que una pasada no hubo ningún intercambio, el ciclo que controla la cantidad de pasadas se interrumpe antes de llegar a la pasada n-1 y el ordenamiento se da por concluido. Se ha denominado ordenamiento de burbuja porque metafóricamente los elementos parecen burbujear en el arreglo a medida que se ordena... (☹) El siguiente método de la clase *Arreglo* (proyecto *TSB-Ordenamiento*) lo implementa:

```
public void intercambio ()
{
  boolean ordenado = false;
  int i , j , aux , n = v.length;
  for (i=0;  i < n - 1  &&  !ordenado;  i++)
  {
        ordenado = true;
        for ( j=0;  j < n - i - 1;  j++ )
        {
             if ( v[ j ] > v[ j + 1 ] )
             {
                  ordenado = false;
                  aux = v[ j ];
                  v[ j ] = v[ j+1 ];
```

```
                    v[ j+1 ] = aux;
                }
            }
        }
    }
```

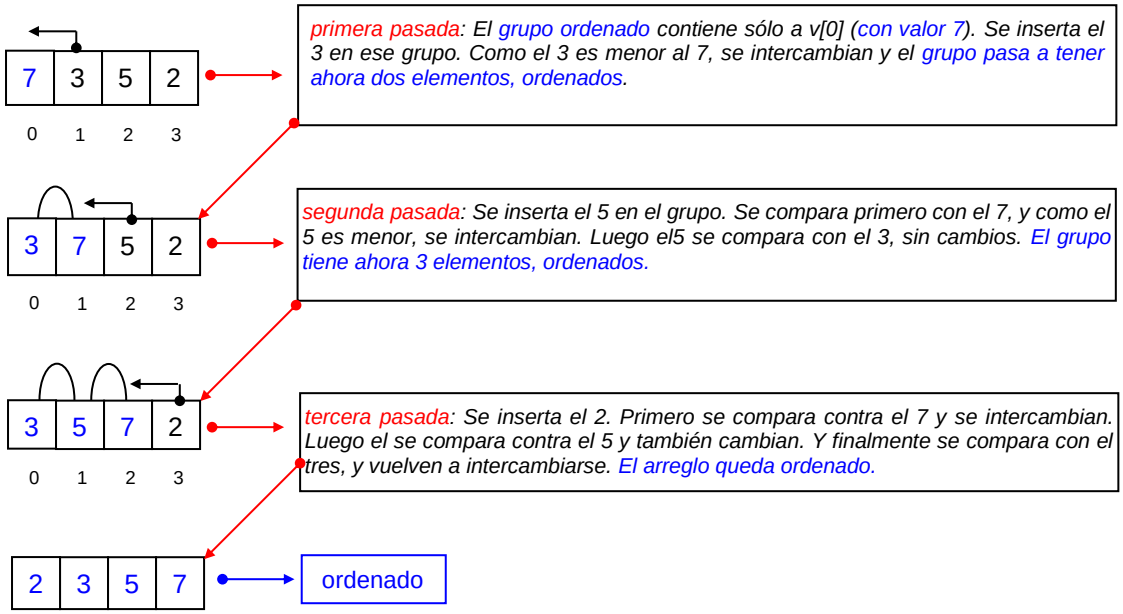
- ***Ordenamiento por Selección Directa:*** Igual que antes, se requieren n-1 pasadas para terminar de ordenar el arreglo, pero ahora se toma la casilla v[i] y se compara su contenido contra todo el resto del vector. La idea es que cada vez que aparezca un valor menor que el contenido en v[i], se reemplace v[i] por ese menor. La casilla v[i] se designa como casilla pivot. Cuando una pasada termina, la casilla pivot contiene al menor valor encontrado, que ya queda ordenado. En la pasada siguiente, se cambia la casilla pivot a la que sigue; y se procede igual, buscando el menor de los que quedan. Note que ahora no hay forma de usar una bandera de corte. El método procede así:



Y en la clase *Arreglo*, el siguiente método lo implementa:

```
public void seleccion ()
{
      int n = v.length;
      for (int i = 0; i < n - 1; i++)
      {
            for (int j = i + 1; j < n; j++)
            {
                  if ( v[i] > v[j] )
                  {
                     int aux = v[ i ];
                     v[ i ] = v[ j ];
                     v[ j ] = aux;
                  }
            }
      }
}
```

- ***Ordenamiento por Inserción Directa:*** La idea es ahora distinta y más original. Se comienza suponiendo que el valor en la casilla v[0] conforma un subconjunto. Y como tiene un solo elemento, está ordenado. A ese subconjunto lo llamamos un grupo ordenado dentro del arreglo. Se toma v[1] y trata de insertar su valor en el grupo ordenado. Si es mayor a v[0] se intercambian, y si no, se dejan como estaban. En ambos casos, el grupo tiene ahora dos elementos y sigue ordenado. Se toma v[2] y se procede igual, comenzando la comparación contra v[1] (que es el mayor del grupo). Así, también hacen falta n-1 pasadas, de forma que en cada pasada se inserta un nuevo valor al grupo. El método procede así:



Y en la clase *Arreglo*, el siguiente método lo implementa:

```
public void insercion ()
{
      int n = v.length;
      for (int j = 1; j < n; j++)
      {
            int k, y = v[j];
            for (k = j-1; k >= 0 && y < v[k]; k--)
            {
                v[k+1]= v[k];
            }
            v[k+1]= y;
      }
}
```

9.] Funcionamiento de los métodos de ordenamiento compuestos o mejorados.

Los métodos de este grupo están basados en la idea de mejorar algunos aspectos que hacen que los métodos directos no tengan buen rendimiento cuando el tamaño del arreglo es grande o muy grande. De nuevo, suponemos un arreglo v de n elementos, y ordenamiento de menor a mayor. La clase Arreglo del modelo TSB-Ordenamiento implementa los métodos que mostrarán aquí.

- ***Ordenamiento Rápido (Quicksort):*** Si se observa el método de Intercambio Directo o Bubblesort, se verá que algunos elementos (los mayores o "más pesados") tienden a viajar más rápidamente que otros (los menores o "más livianos") hacia su posición

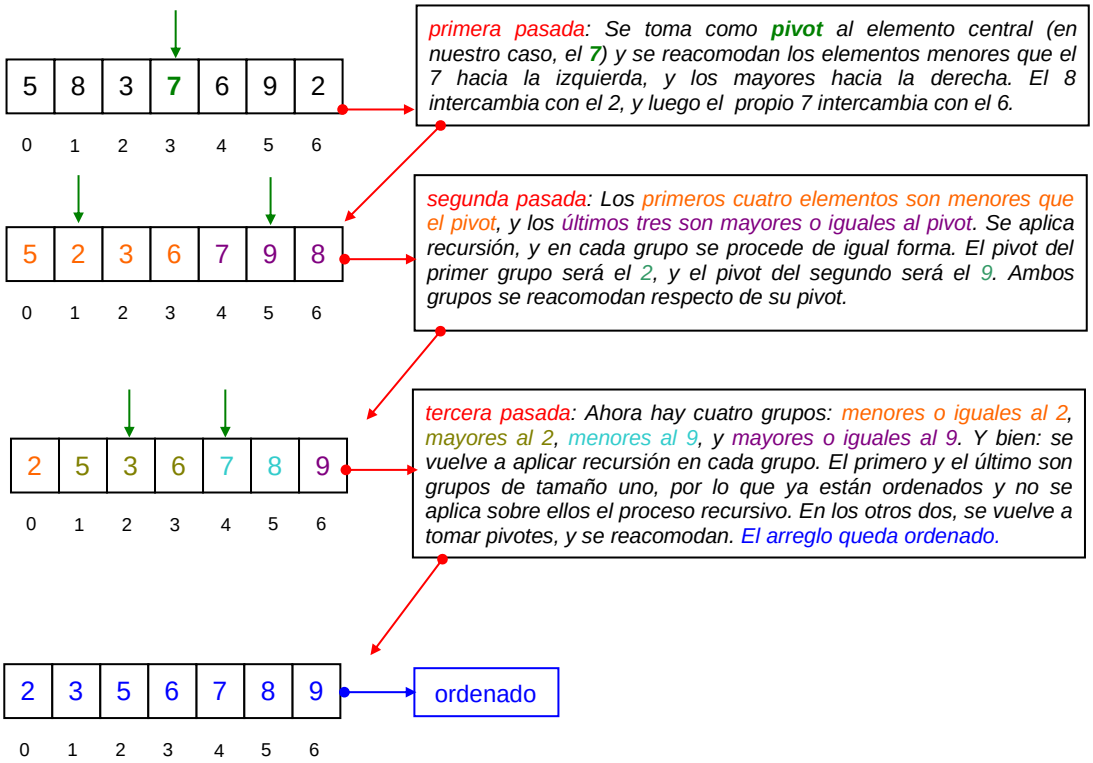
final en el arreglo. El proceso de "burbujeo" es evidentemente asimétrico, pues los elementos más grandes se comparan más veces que los más chicos en la misma pasada. También se llama liebres y tortugas a estos elementos... en obvia alusión a su velocidad de traslado por el vector. El resultado es que el método necesita muchas más pasadas hasta que se acomode el último de los menores que viaja desde la derecha...

En 1960, un estudiante de ciencias de la computación llamado C. Hoare se haría famoso al presentar en Communicatons of the ACM una versión mejorada del bubble sort, al cual llamó simplemente Ordenamiento Rápido o Quicksort... y desde entonces ese método se ha convertido en el más estudiado de la historia de la programación, entre otras cosas por ser hasta ahora en general el método más rápido (aunque hay situaciones en que se comporta bastante mal, esas situaciones son raras y poco probables). La idea es recorrer el arreglo desde los dos extremos. Se toma un elemento pivot (que suele ser el valor ubicado al medio del arreglo pero puede ser cualquier otro). Luego, se recorre el vector desde la izquierda buscando algún valor que sea mayor que el pivot. Al encontrarlo, se comienza una búsqueda similar desde la derecha, pero ahora buscando un valor menor al pivot. Cuando ambos hayan sido encontrados, se intercambian entre ellos, y se sigue buscando otro par de valores en forma similar, hasta que ambas secuencias de búsquda se crucen entre ellas. De esta forma, se favorece que tanto los valores mayores ubicados muy a la izquierda como los menores ubicados muy a la derecha, viajen rápido hacia el otro extremo... y todos se vuelven liebres.

Al terminar esta pasada, se puede ver que el arreglo no queda necesariamente ordenado, pero queda claramente dividido en dos subarreglos: el de la izquierda contiene elementos que son todos menores o iguales al pivot, y el de la derecha contiene elementos mayores o iguales al pivot. Pero ahora se puede aplicar exactamente el mismo proceso a cada subarreglo, usando recursividad. El mismo método que particionó en dos el arreglo original, se invoca a sí mismo dos veces más, para partir en otros subarreglos a los dos que se obtuvieron recién. Con esto se generan cuatro subarreglos, y con más recursión se procede igual con ellos, hasta que sólo se obtengan particiones de tamaño uno... Y en ese momento, el arreglo quedará ordenado. El método procede como se ve en la gráfica de la página siguiente; y se implementa con los siguientes métodos en la clase *Arreglo* (note que el segundo método es privado, y es que el que realmente ordena el arreglo, aplicando recursión):

```
public void quickSort ()
{
    ordenar ( 0, v.length - 1 );
}

private void ordenar (int izq, int der)

{
   int i, j, x, y;
   i = izq;
   j = der;
   x = v[(izq + der) / 2];
   do
   {
        while (v[ i ] < x && i < der) i++;
```

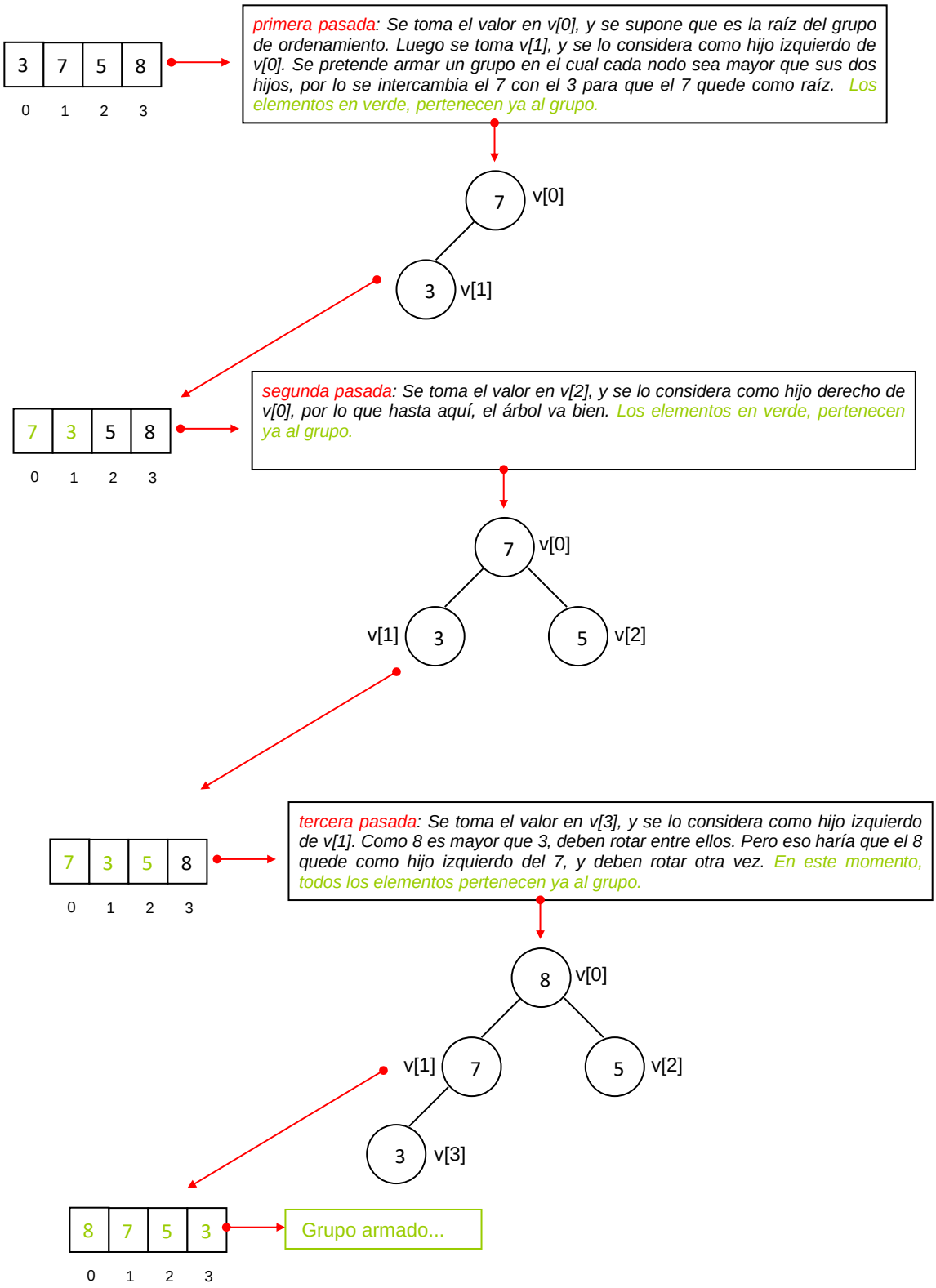
```
            while (x < v[ j ] && j > izq ) j--;
            if (i  <=  j)
            {
                  y = v[ i ];
                  v[ i ] = v[ j ];
                  v[ j ] = y;
                  i++;
                  j--;
            }
      }
      while (i  <=  j );
      if ( izq  <  j ) ordenar(izq, j);
      if ( i  < der ) ordenar(i, der);
}
```



- ***Ordenamiento de Montículo (Heapsort):*** El método de ordenamiento por Selección Directa realiza n-1 pasadas, y el objetivo de cada una es seleccionar el menor de los elementos que sigan sin ordenar en el arreglo, y trasladarlo a la casilla pivot. El problema es que la búsqueda del menor en cada pasada se basa en un proceso secuencial, y exige demasiadas comparaciones.

  En 1964, otro estudiante de ciencias de la computación, llamado J. Williams, publicó un algoritmo mejorado en Communicatons of the ACM, y llamó al mismo Ordenamiento de Montículos o Heapsort (en realidad, debería traducirse como Ordenamiento de Grupos de Ordenamiento, pero queda redundante...) Ese algoritmo reduce de manera drástica el tiempo de búsqueda o selección del menor en cada pasada, usando una estructura de datos conocida como grupo de ordenamiento (que no es otra cosa que una cola de prioridades, pero optimizada en velocidad: los elementos se insertan rápidamente, ordenados de alguna forma, pero cuando se extrae alguno, se extrae el menor [o el mayor, según las necesidades]) Igual que antes, al seleccionar el menor (o el mayor) se lo lleva a su lugar final del arreglo.

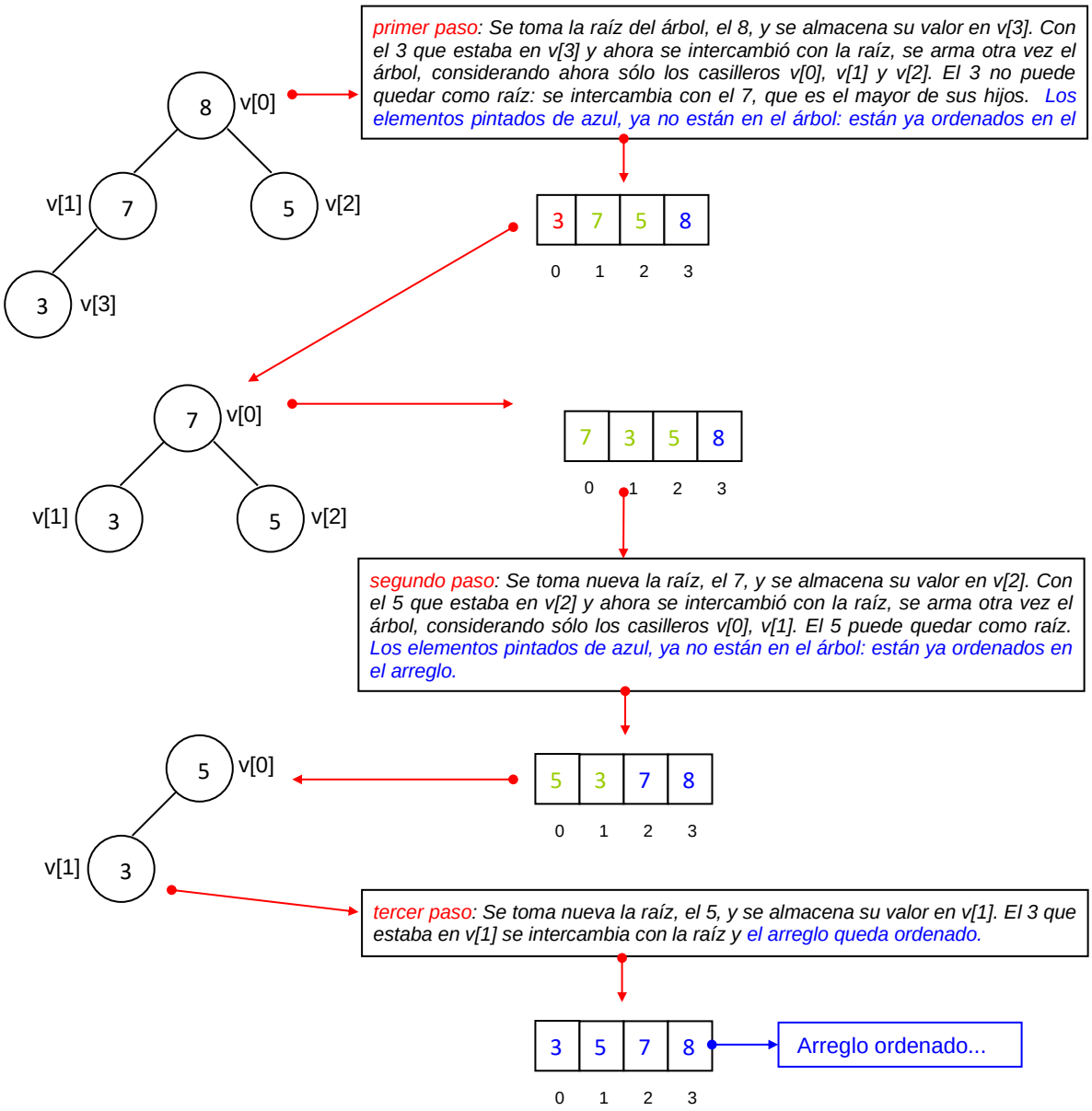
Luego se extrae el siguiente menor (o mayor), se lo reubica, y así se prosigue hasta terminar de ordenar.



En esencia, un grupo de ordenamiento es un árbol binario casi completo (todos los nodos hasta el anteúltimo nivel tienen dos hijos, a excepción de los nodos más a la derecha en ese nivel que podrían no tener los dos hijos) en el cual el valor de cada nodo es mayor que el valor de sus dos hijos. La idea es comenzar con todos los elementos del arreglo, y formar un

árbol de este tipo, pero dentro del mismo arreglo (de otro modo, se usaría demasiado espacio adicional para ordenar...). Es simple implementar un árbol completo o casi completo en un arreglo: se ubica la raiz en el casillero v[0], y luego se hace que para nodo en el casillero v[i], su hijo izquierdo vaya en v[2*i + 1] y su hijo derecho v[2*i + 2]. Así, el algoritmo Heapsort primero lanza una fase de armado del grupo, en la cual los elementos del arreglo se reubican para simular el árbol. Gráficamente, el modelo simple de la página anterior muestra el proceso de armado del grupo para un pequeño arreglo de cuatro elementos.

Una vez armado el grupo, comienza la segunda fase del algoritmo: Se toma la raiz del árbol (que es el mayor del grupo), y se lleva al último casillero del arreglo (donde quedará ya ordenado). El valor que estaba en el último casillero se reinserta en el árbol (que ahora tiene un elemento menos), y se repite este mecanismo, llevando el nuevo mayor al anteúltimo casillero. Así se continúa, hasta que el árbol quede con sólo un elemento, que ya estará ordenado en su posición correcta del arreglo. Gráficamente, la segunda fase sería así:



Y en la clase *Arreglo*, el siguiente método lo implementa:

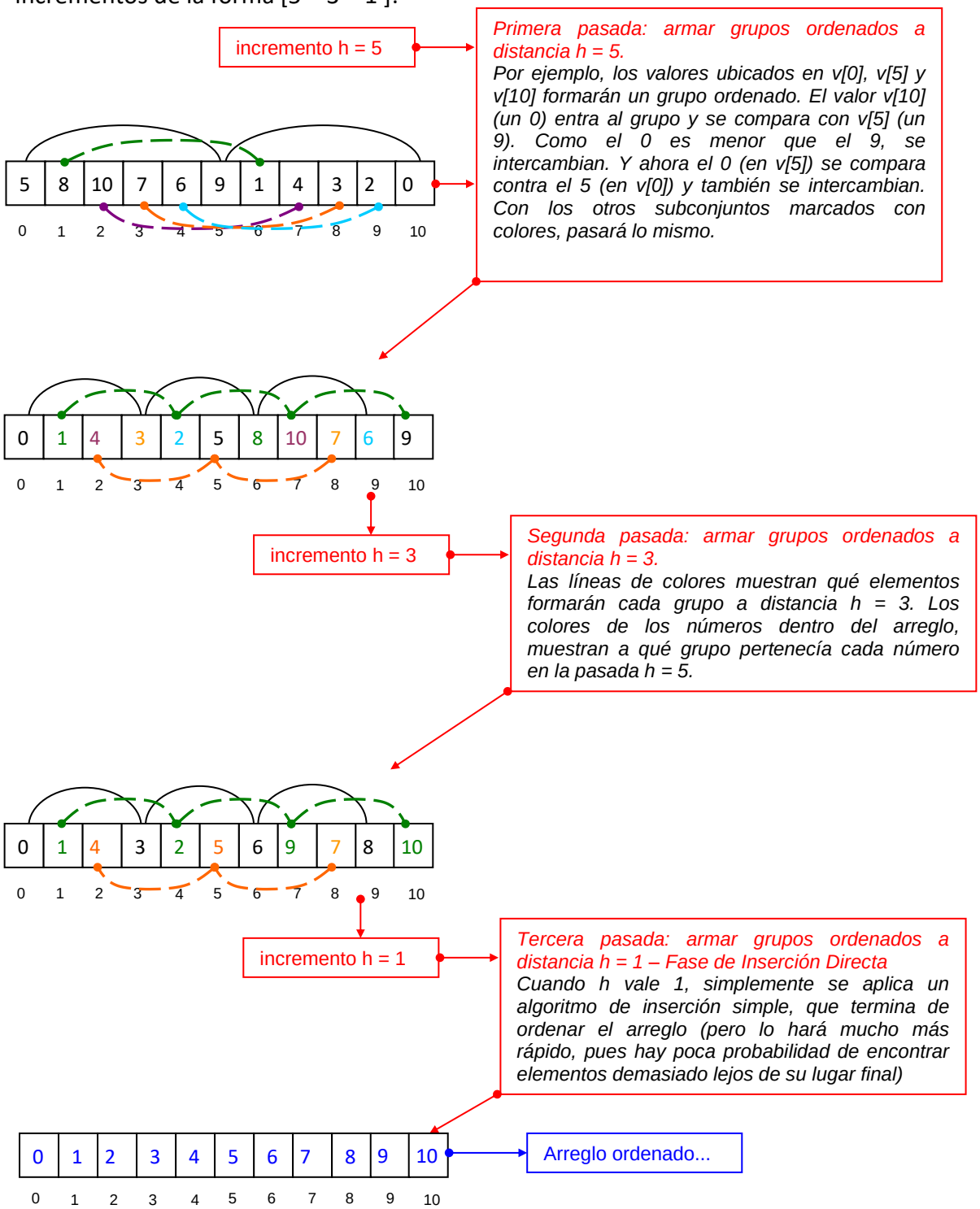
```
public void heapSort()
{
   int i, e, s, f, valori;
   int n = v.length;

   // Primera fase: crear el grupo inicial...
   for (i = 1; i < n; i++)
   {
        e = v[i];
        s = i;
        f = (s-1)/2;
        while (s>0 && v[f] < e)
        {
              v[s] = v[f];
              s = f;
              f = (s-1)/2;
        }
        v[s] = e;
   }

   // Segunda fase: extraer raiz, y reordenar el vector y el grupo...
   for (i = n-1; i>0; i--)
   {
        valori = v[i];
        v[i] = v[0];
        f = 0;
        if (i == 1) s = -1; else s = 1;
        if (i > 2 && v[2] > v[1])  s = 2;
        while ( s >= 0 && valori < v[s])
        {
              v[f] = v[s];
              f = s;
              s = 2*f + 1;
              if (s + 1 <= i - 1 && v[s] < v[s+1]) s++;
              if (s > i - 1) s = -1;
        }
        v[f] = valori;
   }
}
```

- ***Ordenamiento por Incrementos Decreciente (Shellsort):*** El método de ordenamiento por Inserción Directa es el más rápido de los métodos simples, pues aprovecha que un subconjunto del arreglo está ya ordenado y simplemente inserta nuevos valores en ese conjunto de forma que el grupo siga ordenado. El problema es que si llega a aparecer un elemento muy pequeño en el extremo derecho del arreglo, cuando el grupo ordenado de la izquierda ya contiene a casi todo el vector, la inserción de ese valor pequeño demorará demasiado, pues tendrá que compararse con casi todo el arreglo para llegar a su lugar final.

  En 1959, un técnico de la General Electric Company llamado Donald Shell, publicó un algoritmo que mejoraba al algoritmo de inserción directa y lo llamó High-Speed Sorting Procedure, aunque en honor a su creador se lo terminó llamando Shellsort.

  La idea es lanzar un proceso de ordenamiento por inserción, pero en lugar de hacer que cada valor que entra al grupo se compare con su vecino inmediato a la izquierda, se comience haciendo primero un reacomodamiento de forma que cada elemento del arreglo se compare contra elementos ubicados más lejos, a distancias mayores

que uno, y se intercambien elementos a esas distancias. Luego, en pasadas sucesivas, las distancias de comparación se van acortando y repitiendo el proceso con elementos cada vez más cercanos unos a otros. De esta forma, se van armando grupos ordenados pero no a distancia uno, sino a distancia h tal que h > 1. Finalmente, se termina tomando una distancia de comparación igual a uno, y en ese momento el algoritmo se convierte lisa y llanamente en una Inserción Directa para terminar de ordenar el arreglo. Las distancias de comparación se denominan en general incrementos decrecientes, y de allí el nombre con que también se conoce al método. Gráficamente, mostramos la idea con un pequeño arreglo, tomando incrementos de la forma [5 – 3 – 1]:



No es simple elegir los valores de los incrementos decrecientes, y de esa elección depende muy fuertemente el rendimiento del método. En general, digamos que no es necesario que sean demasiados: suele bastar con una cantidad de distancias igual o menor al 10% del tamaño del arreglo, pero debe asegurarse siempre que la última sea igual uno, pues de lo contrario no hay garantía que el arreglo quede ordenado. Por otra parte, es de esperar que los valores elegidos como distancias de comparación, no sean todos múliplos entre ellos, pues si así fuera se estarían comparando siempre las mismas subsecuencias de elementos, sin mezclar nunca esas subsecuencias. Sin embargo, no es necesario que los valores de los incrementos decrecientes sean todos necesariamente primos. Es suficiente con garantizar valores que no sean todos múltiplos entre sí.

En la clase *Arreglo*, el siguiente método lo implementa:

```
public void shellSort()
{
   int h, n = v.length;
   for (h = 1; h <= n / 9; h = 3*h + 1);
   for ( ; h > 0; h /= 3)
   {
         for (int j = h; j < n; j++)
         {
              int k, y = v[j];
              for (k = j - h; k >= 0 && y < v[k]; k-=h)
              {
                    v[k+h] = v[k];
              }
              v[k+h] = y;
         }
   }
}
```

10.] Análisis de Eficiencia del Algoritmo Quicksort. Variantes de Selección del Pivot.

Sólo a modo de ejemplo de la clase de análisis que suelen hacerse para medir la eficiencia de un algoritmo, se indica la demostración del hecho ya conocido4: en el caso promedio (es decir, cuando los datos en el arreglo viene dispuestos en una secuencia estrictamente aleatoria), el algoritmo Quicksort es O (n * log(n)).

En principio, supongamos que el tamaño n del arreglo a ordenar es potencia de 2, por ejemplo, n = 2m. De esta forma, resulta que m es entonces igual a $\log_2(n)$ (entiéndase logaritmo en base 2 de n)5. Supongamos también que el elemento elegido como pivot para realizar las comparaciones de intercambio inicial, será el que se encuentra en la parte media del arreglo. Entonces, en el caso promedio (si los elementos están aleatoriamente disribuidos) habrá aproximadamente n comparaciones en la primera pasada, y luego el arreglo se dividirá en dos mitades de tamaño n/2 aproximadamente.

En cada una de esas mitades habrá a su vez otras n/2 comparaciones, y se forman ahora cuatro sub-arreglos de tamaño n/4 aproximadamente. Si se sigue adelante, luego de partir a

[4] Fuente: Langsam, Y., Augenstein, M., y Tenenbaum, A. (1997). "*Estructura de Datos con C y C++ (2da. Edición)*". México: Prentice Hall. ISBN: 968-880-798-2 [disponible en biblioteca central]

[5] Recordemos que el logaritmo en cualquier base de un valor *n*, es el exponente al que hay que elevar la base para obtener *n* como resultado. De allí que si $n = 2^m$, entonces despejando *m* se obtiene que el valor de *m* es $m = log(n)$.

la mitad los sub-arreglos m veces, se tendrán n sub-arreglos de tamaño 1 (uno). De aquí sale que la cantidad total de comparaciones para todo el ordenamiento es aproximadamente:

$$n + 2 * (n/2) + 4 * (n/4) + 8 * (n/8) + ... + n * (n/n)$$

o lo que es lo mismo:

$$n + n + n + n + ... + n \quad \text{(un total de m términos)}$$

Hay m términos porque el arreglo se divide m veces. Por lo tanto, la cantidad total de comparaciones es O (n * m). Pero si recordamos que m = log(n), tenemos finalmente que Quicksort es O (n * log(n)) en el caso promedio. El análisis para el caso en que el número de elementos no es una potencia de 2, es un poco más complicado, pero el resultado es el mismo.

Existen casos de configuraciones de entrada que llevan al Quicksort al peor caso: puede probarse que si los elementos del arreglo están distribuidos de forma que cada vez que se toma el pivot su valor es el menor (o el mayor) en el subarreglo considerado, entonces Quicksort es... $O(n^2)$ (¡cuadrático!) Intutitvamente puede verse que esto es cierto: si recurrentemente se toma el menor o el mayor del subarreglo a ordenar, entonces se producirán dos particiones pero una de ellas tendrá un solo elemento y la otra tendrá n-1 elementos… y al repetirse este patrón, el árbol de llamadas recursivas tendrá n niveles en lugar de log(n) niveles, por lo que surge el orden cuadrático.

Las distribuciones de datos que pueden causar que Quicksort degenere en un rendimiento cuadrático dependen de la forma en que se selecciona el pivot. Una forma simple de hacerlo consiste en elegir como pivot al primer elemento del subarreglo analizado (o elegir el último). Sin embargo, estas dos formas de selección de pivot son justamente las que maximizan la posibilidad de caer en un rendimiento cuadrático: si el arreglo estuviese ya ordenado, el algoritmo demoraría un tiempo de orden cuadrático en no hacer nada más que comparaciones, sin intercambio alguno. Por lo tanto, es una mala idea tomar como pivot a alguno de los elementos de los extremos.

La variante que hemos mostrado implementada en el modelo *TSB-Ordenamiento* (que acompaña a esta ficha de estudio) toma como pivot al elemento central del subarreglo analizado, y esta es una estrategia muy estable incluso si el arreglo ya estuviese ordenado (en este caso, se invierte un tiempo de O(n * log(n)) en recorrer el arreglo, sin hacer intercambios). Sin embargo, aún puede ocurrir que aparezca alguna distribución de datos en la cual el elemento central elegido siempre sea el menor o el mayor. Hemos dicho que esa distribución es altamente improbable, pero aún así muestra que la estrategia de selección del pivot podría mejorarse todavía más.

La estrategia más usada se conoce como *selección del pivot por mediana de tres*, y permite eliminar por completo la posibilidad de caer en el peor caso, incluso para distribuciones de datos muy malas o adversas. Esta variante consiste en seleccionar como pivot al valor mediano entre el primero, el último y el central del subarreglo analizado. De esta forma, se garantiza que nunca se tomará como pivot al mayor o al menor, evitando el peor caso antes mencionado. Idealmente, se podría pensar lo mejor sería tomar como pivot al valor mediano de todo el arreglo, pero eso haría que el cálculo de ese mediano lleve demasiado tiempo.

Se han ensayado otras variantes, tomando como pivot al valor mediano entre cinco o más componentes del subarreglo, pero las pruebas de rendimiento no han mostrado una mejora

significativa en el algoritmo (además de ser más complicadas de programar). Por lo tanto, la estrategia de selección de pivot por mediana de tres se ha consolidado como una de las más usadas en la implementación del Quicksort. Dejamos como tarea para el alumno la modificación del algoritmo presentado en el modelo *TSB-Ordenamiento* para que en lugar de usar selección del pivot por valor central, aplique mediana de tres. Los alumnos pueden encontrar numerosas implementaciones ya desarrolladas en la web y en libros especializados en Estructuras de Datos. Recomendamos –no obstante- el libro "Estructuras de Datos en Java" de Mark Allen Weiss (página 231 y siguientes).

Para concluir esta sección, se muestran los órdenes de peor caso de los algoritmos de ordenamiento vistos (calculando cantidad de comparaciones), más algunas consideraciones respecto de casos promedio o incluso mejores casos si esas situaciones son relevantes:

Algoritmo	**Orden (peor caso)**	**Observaciones**
Burbuja (Intercambio Directo)	$O(n^2)$	Mejor caso: O(n) si el arreglo está ya ordenado.
Selección Directa	$O(n^2)$	Siempre...
Inserción Directa	$O(n^2)$	
Quicksort	$O(n^2)$	Caso medio (y más probable...): O(n * log(n))
Heapsort	O(n * log(n))	Caso medio: O(n * log(n))
Shellsort	$O(n^{1.5})$	Para la serie de incrementos mostrada en clase