APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

Por favor, reportar errores, omisiones y sugerencias a dfridlender@gmail.com
Consultar la página de la materia:
http://cs.famaf.unc.edu.ar/wiki/doku.php?id=algo2:main

Introducción

En las materias **Introducción a los Algoritmos** y **Algoritmos y Estructuras de Datos I** el énfasis estaba puesto en **qué** hace un programa. Se especificaba con detalle las pre- y post- condiciones que el programa debía satisfacer y se derivaba un programa que satisficiera dicha especificación.

En **Algoritmos y Estructuras de Datos II**, el énfasis estará puesto en **cómo** resuelve el programa el problema especificado. Desarrollaremos instrumentos que nos permitan comparar diferentes programas que resuelven un mismo problema.

Uno de los aspectos que es importante considerar al comparar algoritmos es el referido a los recursos que el programa necesita para ejecutarse: tiempo de procesamiento, espacio de memoria, tiempo de utilización de un dispositivo. El estudio de la necesidad de recursos de un programa o algoritmo se llama **análisis del algoritmo** y lo que dicho análisis determina es la **eficiencia** del mismo.

El análisis de **eficiencia en espacio** de un programa determinará cuánta memoria debe tener el equipo para su correcta ejecución. El análisis de **eficiencia en tiempo** de un programa nos permite estimar el tiempo que tardará el mismo en completar su ejecución.

Es habitual poner mayor énfasis en el anális de eficiencia en tiempo que en el de la eficiencia en espacio, pero ambos son relevantes para entender el comportamiento de un algoritmo.

Considere el siguiente **problema del pintor**

> Un pintor tarda una hora y media en pintar una pared de 3 metros de largo. ¿Cuánto tardará en pintar una de 5 metros de largo?

Sabemos cómo resolver este tipo de problemas: si pintar una pared de 3 metros de largo requiere una hora y media, pintar cada metro está insumiendo 30 minutos. Por lo tanto, pintar 5 metros requerirá dos horas y media. La corrección de este sencillo razonamiento se debe a que sabemos que el tiempo que lleva pintar una pared (de altura fija) es **proporcional a su longitud**. Esto lo podemos deducir de la manera habitual, conocida, de pintar una pared.

Podemos hacernos preguntas similares al abordar otros problemas. Por ejemplo, considere el siguiente **problema del bibliotecario**

> Un bibliotecario tarda un día en ordenar alfabéticamente una biblioteca con 1000 expedientes. ¿Cuánto tardará en ordenar una con 2000 expedientes?

Con un razonamiento similar al utilizado para el problema del pintor, podríamos rápidamente concluir que el bibliotecario tardará dos días. Pero ¿es correcto razonar así? El tiempo que lleva ordenar una biblioteca, ¿es proporcional a la cantidad de expedientes? ¿Cuánto más trabajo es ordenar 2000 expedientes que 1000? ¿Es el doble? Para contestar estas preguntas tenemos que analizar la manera de ordenar. No contamos -como en el caso de pintar- con una manera única, habitual o universal de ordenar expedientes.

Consideremos nuevamente problemas más familiares, como los siguientes

(1) Un pintor demora una hora y media en pintar una pared cuadrada de 3 metros de lado. ¿Cuánto demorará en pintar una de 5 metros de lado?
(2) Si lleva cinco horas inflar un globo aerostático esférico de 2 metros de diámetro, ¿cuánto llevará inflar uno de 4 metros de diámetro?

El primero de ellos se parece al problema del pintor, salvo que ahora la pared es cuadrada. Al pensarlo con detenimiento, descubrimos que el tiempo que lleva pintar la pared cuadrada, no es **proporcional** a la longitud del lado, sino **a la superficie** de la pared. La magnitud de la tarea de pintar la pared, está determinada por su superficie. Como la superficie de un cuadrado de 3 metros de lado es 9 metros cuadrados y se pinta en una hora y media, pintar cada metro cuadrado lleva 10 minutos. La superficie de un cuadrado de 5 metros de lado es 25 metros cuadrados, pintarlos lleva 250 minutos, o sea 4 horas y 10 minutos.

Para el segundo problema, como el tiempo necesario para inflar el globo, es decir, la magnitud de la tarea de inflarlo, es **proporcional a su volumen**, que a su vez -por ser el globo una esfera- es **proporcional al cubo del diámetro** (su fórmula es $V = \frac{\pi d^3}{6}$), al duplicarse el diámetro de 2 a 4 metros, el volumen se multiplica por $2^3 = 8$. Inflar el globo de 4 metros de diámetro llevará entonces 40 horas.[1]

Estos dos problemas sirven para mostrar que para resolverlos es fundamental determinar a qué es **proporcional** la magnitud de la tarea. También resulta destacable, que no es imprescindible conocer la fórmula exacta. Por ejemplo, el problema del globo aerostático se pudo resolver ignorando el factor $\frac{\pi}{6}$ en la fórmula del volumen de la esfera. Sólo se usó que el volumen es proporcional al cubo del diámetro.

Concluimos entonces que para resolver el problema del bibliotecario es necesario determinar **a qué es proporcional la magnitud de la tarea de ordenar expedientes**. Para ello es imprescindible estudiar métodos de ordenación.

En lo que sigue, asumiremos la existencia de ciertos elementos o items a ordenar, que existe una relación de orden total entre ellos, y que deben ordenarse de menor a mayor. No se asume que los items sean diferentes, pueden contener repeticiones.

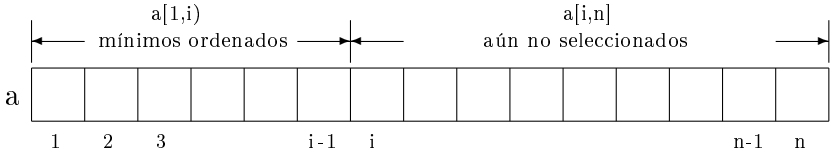
Ordenación por selección. El método de ordenación más sencillo -aunque no el más rápido- es el de **ordenación por selección**. Como su nombre lo indica, ordena seleccionando elementos. En un primer paso selecciona el mínimo elemento y lo coloca en el primer lugar, separándolo de este modo del resto de los elementos. En un segundo paso vuelve a seleccionar el mínimo elemento (de los restantes) y lo coloca en el segundo lugar, separándolo del resto. En el paso i, habiendo ordenado ya los $i-1$ elementos

[1]Ejercicio: comprobar realizando las cuentas detalladamente, la corrección de este resultado.

mínimos, vuelve a seleccionar el mínimo y lo coloca en el lugar i, separándolo del resto. Y continúa de esta manera hasta colocar el último de los elementos en su lugar correcto.

Para escribir este algoritmo con precisión, utilizaremos pseudo-código: un lenguaje algorítmico de fácil traducción a distintos lenguajes de programación. Asumimos que los elementos a ordenar vienen dados en un arreglo (llamado a) que el algoritmo de ordenación debe modificar para que al finalizar la ejecución del algoritmo, dicho arreglo contenga los mismos elementos que el original, pero ordenados de menor a mayor.

Durante la ejecución de la ordenación por selección, podemos pensar que el arreglo está dividido en dos partes: una primera parte a[1,i) (inicialmente vacía pero que crece a cada paso del algoritmo) que contiene los elementos que ya han sido seleccionados, es decir, los $i-1$ menores elementos del arreglo en orden creciente; y una segunda parte a[i,n] (inicialmente es la totalidad del arreglo pero decrece a cada paso) con el resto de los elementos aún no seleccionados.



De acuerdo a lo explicado, el algoritmo de ordenación por selección consiste de un ciclo en el que se cumplen las siguientes condiciones como invariante:

- el arreglo a es una permutación del original,
- un segmento inicial a[1,i) del arreglo está ordenado, y
- dicho segmento contiene los elementos mínimos del arreglo.

El algoritmo de ordenación por selección, puede escribirse en pseudo-código de la siguiente manera:

```
{Pre: n ≥ 0 ∧ a = A}
proc selection_sort (in/out a: array[1..n] of T)
    var i, minp: nat
    i:= 1                          {Inv: a es permutación de A ∧ a[1,i) está ordenado ∧}
                                   {∧ los elementos de a[1,i) son menores o iguales a los de a[i,n]}
    do i < n → minp:= min_pos_from(a,i)
               swap(a,i,minp)
               i:= i+1
    od
end proc
{Post: a está ordenado y es permutación de A}
```

La precondición denota por A el valor inicial del arreglo a. La postcondición dice que al finalizar la ejecución, el arreglo a es una permutación de su valor inicial, y que está ordenado. Estas dos condiciones especifican el problema de ordenación: un algoritmo de ordenación es uno que devuelve una permutación ordenada de su entrada.

El invariante es el mismo que se explicó más arriba.

Cuando un algoritmo realiza modificaciones (como en este caso al arreglo a) lo llamamos procedimiento (**proc**). Un procedimiento puede recibir parámetros con datos

(indicadas con el prefijo **in**), con variables para inicializar (prefijo **out**), o con variables para modificar (prefijo **in**/**out**).

Cuando un algoritmo no realiza modificaciones sino que simplemente calcula fórmulas o resultados, lo llamamos función (**fun**). Como la función no realiza modificaciones, todos sus parámetros son meramente datos, por consiguiente el prefijo es innecesario.

El procedimiento selection_sort utiliza dos variables: la variable i, cuyo rol se advierte ya en las explicaciones del algoritmo, y la variable minp que tiene como propósito señalar la posición donde se encuentra el nuevo elemento seleccionado. Para calcularlo se utiliza la función min_pos_from que al aplicarse al arreglo a y a la variable i devuelve la posición del mínimo del segmento de arreglo a[i,n]. Luego de calcularlo se invoca al procedimiento swap para intercambiarlo con el que se encuentra en la posición i. Este intercambio tiene dos propósitos: por un lado, coloca al elemento seleccionado en su posición correspondiente como elemento ya ordenado (la i), y por el otro, evita que el elemento que ocupaba la celda i se pierda, consiguiéndole una nueva celda provisoria: la que justamente se libera por haber estado ocupada por el elemento seleccionado.

```
{Pre: a = A ∧ 1 ≤ i,j ≤ n }
proc swap (in/out a: array[1..n] of T, in i,j: nat)
    var tmp: T
    tmp:= a[i]
    a[i]:= a[j]
    a[j]:= tmp
end proc
{Post: a[i] = A[j] ∧ a[j] = A[i] ∧ ∀ k. k ∉{i,j} ⇒ a[k]=A[k]}
```

La postcondición del procedimiento swap implica que el mismo produce una permutación del arreglo que recibe como parámetro. Como el algoritmo selection_sort sólo modifica el arreglo a invocando al procedimiento swap reiteradamente, la condición que establece que el arreglo ordenado debe ser una permutación del inicial queda garantizada. Ésta es una observación interesante: **si uno se limita a modificar el arreglo sólo invocando al procedimiento swap, necesariamente se tendrá siempre una permutación del arreglo inicial**.

Como ya se explicó, la función min_pos_from aplicada a los parámetros a e i debe devolver la posición del mínimo del segmento de arreglo a[i,n]. Como el objetivo es intercambiarlo con el que se encuentra en la posición i del arreglo, es necesario devolver la **posición** donde ese mínimo se encuentra.

```
{Pre: 0 < i ≤ n}
fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
    var j: nat
    minp:= i
    j:= i+1                                   {Inv: a[minp] es el mínimo de a[i,j)}
    do j ≤ n → if a[j] < a[minp] then minp:= j fi
               j:= j+1
    od
end fun
{Post: a[minp] es el mínimo de a[i,n]}
```

La variable minp que se utiliza para devolver el resultado se declara en el encabezado de la función por medio de la cláusula **ret**.

¿Por qué la función min_pos_from no devuelve el mínimo sino su posición?

¿Por qué dice i $<$ n en vez de i $\leq$ n en la condición del **do** del procedimiento selection_sort?

¿Por qué se llama selection_sort?

El comando for. En el algoritmo presentado los dos ciclos **do** se utilizan para repetir ciertas acciones para cada valor de las variables i (o j) desde un valor (cota inferior) a otro (cota superior). La variable i (o j) recibe el nombre de índice. Se observa que los índices sólo son modificados al final de cada ciclo, cuando se los incrementa. En estas situaciones, utilizaremos una notación más compacta. En primer lugar, omitiremos declarar el índice. Además, en vez de escribir

```
k:= n
do k ≤ m → C
            k:= k+1
od
```

escribiremos

```
for k:= n to m do C od
```

Para que esta notación tenga sentido es fundamental que k no sea modificado en el cuerpo C del ciclo. El propósito de esta notación es hacer más evidente que C se ejecuta una vez para cada valor de k desde n hasta m. Observar que si n es mayor que m, no se ejecuta el cuerpo del ciclo **for** ni siquiera una vez.

A veces es deseable que el primer valor del índice sea la cota superior y que gradualmente se decremente hasta alcanzar la cota inferior. Para esos casos, reemplazamos **to** por **downto**.

Por ejemplo, la función factorial puede programarse de cualquiera de las formas siguientes

```
{Pre: 0 ≤ n}
fun fact (n: nat) ret r: nat
    r:= 1
    for i:= 1 to n do
        r:= r * i
    od
end fun
{Post: r = n!}
```

que realiza la multiplicación $((((1 * 1) * 2) * 3) \ldots) * n$ y

```
fun fact (n: nat) ret r: nat
    r:= 1
    for i:= n downto 1 do
        r:= r * i
    od
end fun
```

que realiza la multiplicación $((((1 * n) * (n-1)) * (n-2)) * \ldots) * 1$.

Utilizando ciclos **for** el procedimiento selection_sort puede escribirse

```
{Pre: n ≥ 0 ∧ a = A}
proc selection_sort (in/out a: array[1..n] of T)
    var minp: nat
    for i:= 1 to n-1 do        {Inv: a es permutación de A ∧ a[1,i) está ordenado ∧}
                               {∧ los elementos de a[1,i) son menores o iguales a los de a[i,n]}
        minp:= min_pos_from(a,i)
        swap(a,i,minp)
    od
end proc
{Post: a está ordenado y es permutación de A}
```

Asimismo, la función de **selección** min_pos_from se puede escribir

```
{Pre: 0 < i ≤ n}
fun min_pos_from (a: array[1..n] of T, i: nat) ret minp: nat
    minp:= i
    for j:= i+1 to n do                    {Inv: a[minp] es el mínimo de a[i,j)}
        if a[j] < a[minp] then minp:= j fi
    od
end fun
{Post: a[minp] es el mínimo de a[i,n]}
```

Ahora que tenemos un algoritmo de ordenación, podemos intentar resolver el problema del bibliotecario. Para ello, debemos analizar el algoritmo con el propósito de averiguar cuánto más trabajo implica ordenar 2000 expedientes que 1000 expedientes. Debemos hallar algún parámetro respecto del cual el trabajo sea proporcional. Con el propósito de medir el trabajo que realiza el algoritmo, volvemos nuestra atención sobre el mismo y observamos que contiene distintas tareas u operaciones: realiza asignaciones, sumas, comparaciones, llamadas a funciones y procedimientos, ejecuciones de ciclos, intercambios.

Una posibilidad sería contabilizar todas estas operaciones. Sería complicado ya que no todas ellas son equivalentes, no todas ellas requieren el mismo tiempo para realizarse. Deberíamos contabilizarlas por separado. Realizaremos un análisis más sencillo, que consiste en elegir una sola operación y contar cuántas veces se repite. Por supuesto que para que el análisis sea correcto, debemos ser criteriosos al elegir dicha operación: debe ser una que sea representativa del trabajo que realiza el algoritmo. Para ello se requiere:

- que sea constante (es decir, que el trabajo que implica realizar dicha operación debe ser el mismo, independientemente del número de celdas del arreglo),
- que "ninguna otra operación se repita más que la elegida"; más precisamente, toda otra operación puede repetirse a lo sumo en forma proporcional al modo en que se repite la operación elegida.

Por ejemplo, en el caso que nos ocupa, el procedimiento selection_sort contiene un ciclo; debemos buscar dentro de él la tarea a elegir ya que allí se encuentran las

operaciones que más se repiten. El **for** contiene implícitamente operaciones que se repiten: se incrementa el índice y se chequea la condición luego de cada ejecución del cuerpo del mismo. Además, cada vez que se ejecuta el cuerpo se invoca a la función min_pos_from y al procedimiento swap. Cada una de estas operaciones se realiza una vez para cada valor de i, es decir, un total de n-1 veces.

Todas estas operaciones son constantes, con la sola excepción de la ejecución de la función min_pos_from. En efecto, la función min_pos_from contiene a su vez (además de la inicialización de minp) un ciclo **for** que requiere la repetición de ciertas operaciones. Como todo ciclo **for** contiene incrementos y chequeos de condición de salida implícitos luego de cada ejecución. Además, cada vez que se ejecuta el ciclo se realizan accesos al arreglo, una comparación entre celdas del mismo y posiblemente una asignación a minp. Todas estás operaciones (salvo la asignación a minp que requiere que la condición del **if** sea verdadera para ejecutarse) se realizan para cada valor de j, es decir, un total de n-i veces.

Cualquiera de estas operaciones es representativa del comportamiento del algoritmo. Además de ser constantes, son las que más se repiten porque fueron encontradas en el ciclo **for** de la función min_pos_from, que a su vez se invoca desde el ciclo **for** del procedimiento selection_sort. Elegimos, como es habitual al analizar algoritmos de ordenación, la comparación entre elementos del arreglo a[j] $<$ a[minp] que se encuentra en la condición del **if**. Como ya observamos, la misma se realiza n-i veces cuando se invoca la función select con el parámetro i. También observamos que la función min_pos_from se invoca n-1 veces, cada vez con distintos valores de i $\in \{1 \ldots \text{n-1}\}$. Entonces, el número total de veces que se realiza dicha comparación es n-1 + n-2 + $\ldots$ + 1. Esto es, un total de $\frac{n(n-1)}{2}$ veces, o distribuyendo, $\frac{n^2}{2} - \frac{n}{2}$ veces.

Resolviendo el problema del bibliotecario. Este análisis nos permite concluir que el trabajo que realiza el procedimiento selection_sort para ordenar un arreglo de longitud n es proporcional a $\frac{n^2}{2} - \frac{n}{2}$. Para el problema del bibliotecario, entonces, tenemos que ordenar 1000 expedientes con este algoritmo involucra 499500 comparaciones mientras que ordenar 2000, involucra 1999000 comparaciones. Es decir, ordenar 2000 expedientes es aproximadamente 4 veces más trabajoso que ordenar 1000. Si tarda un día en ordenar 1000, tardará 4 en ordenar 2000.

Puede observar que la fórmula utilizada, $\frac{n^2}{2} - \frac{n}{2}$, es innecesariamente complicada para resolver el problema. El término que predomina en esta ecuación es el de grado 2. Podríamos decir que el trabajo de selection_sort es proporcional a $\frac{n^2}{2}$. Por lo tanto, también es proporcional a n^2.

Repetimos el cálculo con esta fórmula más sencilla: ordenar 1000 expedientes involucra 1 millón de comparaciones, y ordenar 2000 involucra 4 millones. Arribamos a la misma conclusión (que ordenar 2000 expedientes es 4 veces más trabajo que ordenar 1000) que con la fórmula innecesariamente complicada. Por esta razón es habitual simplificar la notación lo más posible, ignorando constantes multiplicativas (en nuestro caso, $\frac{1}{2}$) y términos de crecimiento despreciable comparado con otros (en nuestro caso, $\frac{n}{2}$ que crece más lentamente que $\frac{n^2}{2}$).

Número de operaciones de un comando. Frecuentemente vamos a querer contar o estimar el número de operaciones que se realizan al ejecutarse un comando determinado. Como no todas las operaciones son igualmente significativas para la performance de un programa dado, frecuentemente se cuentan sólo operaciones de un cierto tipo. La cuenta que se realice dependerá de las operaciones que se pretenden contar y del comando que se está analizando.

Si bien lo habitual es contabilizar las operaciones intuitivamente como hicimos en el caso de selection_sort, basándonos en comprender cómo se ejecuta el algoritmo, a continuación se da una descripción informal para contar metódicamente las operaciones que se realizan durante la ejecución de un comando dado.

Si queremos contar el número de operaciones que se realizan al ejecutarse la secuencia de comandos $C_1;C_2;\ldots;C_n$, sumamos las operaciones que se realizan durante la ejecución de cada comando de la secuencia:

$$\mathrm{ops}(C_1;C_2;\ldots;C_n) = \mathrm{ops}(C_1) + \mathrm{ops}(C_2) + \ldots + \mathrm{ops}(C_n)$$

o más brevemente

$$\mathrm{ops}(C_1;C_2;\ldots;C_n) = \sum_{i=1}^{n} \mathrm{ops}(C_i)$$

En particular, como **skip** representa una secuencia vacía de comandos, ops(**skip**) = 0.

El ciclo **for** k:= n **to** m **do** C(k) **od** puede verse intuitivamente como una abreviatura de la secuencia de comandos C(n);C(n+1);...;C(m).[2] Por ello, si queremos contar el número de operaciones de un ciclo **for**, sumamos las operaciones que se realizan en cada ejecución del cuerpo del mismo. Podemos utilizar por ejemplo la fórmula:

$$\mathrm{ops}(\textbf{for k:= n to m do C(k) od}) = \sum_{\mathrm{k}=n}^{m} \mathrm{ops}(\mathrm{C(k)})$$

Notar que en el lado derecho de la ecuación hemos utilizado n y m para referirnos, respectivamente, a los valores de las expresiones n y m.

Esta fórmula no tiene en cuenta las operaciones necesarias para evaluar n y m ni las necesarias para inicializar y modificar k y para compararlo con el valor de m. La fórmula:

$$\mathrm{ops}(\textbf{for k:= n to m do C(k) od}) = \mathrm{ops(n)} + \mathrm{ops(m)} + \sum_{\mathrm{k}=n}^{m} \mathrm{ops}(\mathrm{C(k)})$$

sí tiene en cuenta las operaciones necesarias para evaluar n y m. Es importante tenerla en cuenta, especialmente cuando n o m son expresiones complejas, como llamadas a funciones tales como factorial o fibonacci, por dar tan solo algunos ejemplos.

[2] En realidad, como n y m no necesariamente son constantes, el ciclo **for** no puede reemplazarse en el código por C(n);C(n+1);...;C(m). Por ejemplo, en el caso de la ordenación por selección, cada llamada a la función min_pos_from, es con un parámetro i diferente. Por lo tanto, el comando **for** en el cuerpo de dicha función itera, para cada i, un número diferente de veces.

La fórmula

$$\text{ops}(\textbf{for k:= n to m do C(k) od}) = (m - n + 2) * \text{ops}(\text{k}\leq m) + \sum_{\text{k}=n}^{m} \text{ops}(\text{C(k)})$$

no tiene en cuenta las operaciones necesarias para evaluar n y m pero sí las necesarias para comparar k con m. Observar que se contabilizan $m - n + 2$ comparaciones,[3] pero se utiliza m en vez de m dado que no se evalúa la expresión m cada vez que se ejecuta el cuerpo del **for** sino una sola vez para toda la ejecución del comando **for**.

Se deja como ejercicio proponer una fórmula que tenga en cuenta todas las operaciones que se requieren para ejecutar el **for**, incluso la inicialización y las modificaciones de k.

Si queremos contar el número de operaciones que se realizan al ejecutarse el comando condicional **if** b **then** c **else** d **fi**, debemos considerar dos casos: b verdadero ó b falso:

$$\text{ops}(\textbf{if b then C else D fi}) = \begin{cases} \text{ops(b)+ops(C)} & b = \text{verdadero} \\ \text{ops(b)+ops(D)} & b = \text{falso} \end{cases}$$

Nuevamente convenimos en utilizar b para referirnos al valor de la expresión b.

Si queremos contar el número de operaciones que se realizan al ejecutarse la asignación x:= e, tenemos 2 fórmulas dependiendo de si queremos o no contar la asignación en sí como una operación. En el primer caso tenemos ops(x:=e) = ops(e)+1 mientras que en el segundo tenemos simplemente ops(x:=e) = ops(e).

En los casos del comando condicional y de la asignación, ops(b) y ops(e) representan los números de operaciones necesarios para evaluar las expresiones b y e.

Para contar el número de operaciones que se realizan al ejecutarse un ciclo **do** es necesario establecer el número de veces que se ejecutará el cuerpo del ciclo. No siempre es fácil obtener dicho número. Una vez determinado este número, llamémosle n, se obtiene una fórmula parecida a la del **for**,

$$\text{ops}(\textbf{do b} \rightarrow \text{C } \textbf{od}) = \text{ops(b)} + \sum_{\text{k}=1}^{n} d_k$$

donde d_k es el número de operaciones que realiza la k-ésima ejecución del cuerpo C del ciclo y la subsiguiente evaluación de la condición o guarda b.

Finalmente, se deja como ejercicio definir las ecuaciones correspondientes a operadores lógicos, relacionales y aritméticos, que seguramente se parecerán a las de la asignación explicada más arriba.

Número de comparaciones de la ordenación por selección. A modo de ejemplo, contemos el número de **comparaciones entre elementos del arreglo a** en el algoritmo de ordenación por selección. Ya lo hicimos intuitivamente, ahora utilizaremos las ecuaciones que definen la función ops. Los cálculos se encuentran a continuación.

[3] ¿Por qué se suma 2?

$$\begin{aligned}
\text{ops(selection_sort(a))} &= \sum_{\text{i=1}}^{\text{n-1}} \text{ ops(min_pos_from(a,i))} \\
&= \sum_{\text{i=1}}^{\text{n-1}} \sum_{\text{j=i+1}}^{\text{n}} \text{ops(a[j] < a[minp])} \\
&= \sum_{\text{i=1}}^{\text{n-1}} \sum_{\text{j=i+1}}^{\text{n}} 1 \\
&= \sum_{\text{i=1}}^{\text{n-1}} \text{(n-i)} \\
&= \sum_{\text{i=1}}^{\text{n-1}} \text{i} \\
&= \frac{\text{n*(n-1)}}{2} \\
&= \frac{\text{n}^2}{2} - \frac{\text{n}}{2}
\end{aligned}$$

La expresión que obtenemos -idéntica a la obtenida apelando exclusivamente a nuestra intuición- indica que el número de comparaciones de la ordenación por selección **es del orden de** n^2, terminología que haremos más precisa pronto. Intuitivamente, el número de comparaciones de la ordenación por selección es proporcional a n^2.

Número de intercambios (swaps) de la ordenación por selección. Observemos que sólo se realizan swaps durante la ejecución del procedimiento selection_sort y no durante la de la función min_pos_from. Por cada valor de i se hace exactamente un intercambio (swap) entre a[i] y a[minp] al final del cuerpo del **for** de selection_sort. Como i toma $n-1$ valores, son $n-1$ intercambios.

Utilizando las fórmulas se obtiene lo mismo

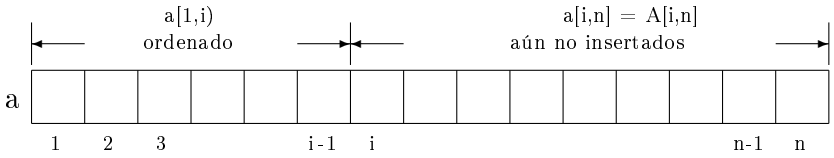
$$\begin{aligned}
\text{ops(selection_sort(a))} &= \sum_{\text{i=1}}^{\text{n-1}} \text{ops(swap(a,i,minp))} \\
&= \sum_{\text{i=1}}^{\text{n-1}} 1 \\
&= \text{n-1}
\end{aligned}$$

Es decir que el número de intercambios (swaps) de la ordenación por selección **es del orden de** n, es decir, es proporcional a n.

Esto confirma que la operación de intercambio no es representativa del comportamiento de la ordenación por selección porque se realizan del orden de n, contra n^2 comparaciones.

Otro ejemplo: ordenación por inserción. No siempre es posible calcular el número exacto de operaciones, dado que puede depender de los valores de entrada. El siguiente algoritmo de ordenación por inserción es un ejemplo de ello.

La ordenación por inserción, frecuentemente utilizada en juegos de cartas que requieran tener un gran número de cartas en la mano ordenadas de manera de que sean fácilmente localizables, procede de la siguiente manera. Al igual que la ordenación por selección, mantiene como invariante que un segmento inicial del arreglo a está ordenado, pero a diferencia de aquélla, sus elementos no son menores a los del segmento restante de a. Paso a paso se inserta un nuevo elemento en el segmento ordenado hasta que al finalizar, dicho segmento es la totalidad del arreglo.



En la ordenación por inserción, no se elige el elemento a insertar, se inserta el primero que aún no se insertó (es decir, el que se encuentra en la celda i).

{Pre: n $\geq$ 0 $\wedge$ a = A}
proc insertion_sort (**in**/**out** a: **array**[1..n] **of T**)
    for i:= 2 **to** n **do** {Inv: a[1,i) está ordenado $\wedge$ a es permutación de A}
        insert(a,i)
    od
end proc
{Post: a está ordenado y es permutación de A}

El procedimiento insertion_sort es muy sencillo: simplemente llama repetidamente al procedimiento insert, que se supone que inserta un elemento (el i-ésimo) en el segmento de a[1,i) que ya está ordenado. Luego de esta inserción, el segmento ordenado tendrá un elemento más, será a[1,i].

El algoritmo de inserción que damos a continuación "arrastra" al nuevo elemento hacia la izquierda hasta que el mismo alcanza su posición. Para arrastrarlo lo intercambia con el de su izquierda tantas veces como sea necesario. Por lo tanto, el elemento nuevo -que originariamente se encontraba en la posición i del arreglo- va cambiando gradualmente de posición (hacia la izquierda). La variable j indica en todo momento dónde se encuentra el elemento nuevo.

{Pre: 0 < i $\leq$ n $\wedge$ a = A}
proc insert (**in**/**out** a: **array**[1..n] **of T**, **in** i: **nat**) **ret**
    j:= i {Inv: a[1,j] y a[j,i] están ordenados $\wedge$ a es permutación de A}
    do $j > 1 \wedge a[j] < a[j-1] \rightarrow$ swap(a,j-1,j)
                                  j:= j-1
    od
end proc
{Post: a[1,i] está ordenado $\wedge$ a es permutación de A}

Para escribir el invariante del procedimiento insert denotamos por a[1,$\overset{j}{,}$i] la secuencia de celdas de a desde la posición 1 hasta la i salteando la j-ésima (para $1 \leq j \leq i$).

Como no se sabe a priori cuántos lugares hacia "la izquierda" será necesario "arrastrar" el elemento que se está insertando, debemos utilizar un ciclo **do** en vez de **for**.

¿Por qué el **for** del procedimiento insertion_sort comienza desde 2?

Número de comparaciones de la ordenación por inserción. Nuevamente contamos **comparaciones entre elementos del arreglo a**, que en este caso son de la forma $a[j] < a[j-1]$ y sólo se realizan en el procedimiento insert. Intuitivamente, cuando i=2 se realiza 1, cuando i=3 se realizan al menos 1 y a lo sumo 2, ..., cuando i=n se realizan al menos 1 y a lo sumo n-1. No podemos obtener el número exacto de comparaciones ya que éste depende de cuántos lugares hacia la izquierda es necesario

"arrastrar" al nuevo elemento. En el **mejor caso** (cuando el arreglo original ya está ordenado) cada llamada a insert realiza una sola comparación (ya que la misma devuelve falso haciendo que el ciclo **do** termine) serán $1 + 1 + \ldots + 1 =$ n-1 comparaciones (es decir, del orden de n comparaciones). En el **peor caso**, (cuando el arreglo inicial está ordenado al revés, de mayor a menor) cada llamada a insert con parámetros a e i realiza i-1 comparaciones (ya que el nuevo elemento debe arrastrarse i-1 pasos hasta la celda 1). Las comparaciones en este caso sumarán $1 + 2 + \ldots +$ (n-1) $= \frac{n^2}{2} - \frac{n}{2}$ (es decir, del orden de n^2 comparaciones).

Obtener el número de comparaciones en el peor caso es importante pues establece una **cota**, una **garantía** sobre el comportamiento del algoritmo. ¿Cuándo se da el peor caso de la ordenación por inserción?

Obtener el número de comparaciones en el mejor caso no tiene la misma importancia, sólo establece una **posibilidad** sobre el comportamiento del algoritmo. ¿Cuándo se da el mejor caso de la ordenación por inserción?

Sí es importante establecer el número de comparaciones en el **caso promedio** o **caso medio** ya que éste determina el comportamiento del algoritmo en la **práctica**. Para calcularlo se necesitan conocimientos de probabilidades. No tiene sentido simplemente promediar el peor caso con el mejor caso, esto daría siempre algo equivalente al peor caso.[4] El número de comparaciones de la ordenación por inserción en el caso promedio asumiendo que los valores en las celdas del arreglo fueron generados al azar es del orden de n^2 (no lo demostramos porque requiere conocimientos de probabilidades).

Número de intercambios de la ordenación por inserción. Para el conteo del número de swaps del algoritmo de ordenación por inserción, nuevamente se distinguen varios casos. En el mejor de ellos (cuando el arreglo inicial está ordenado), como la condición del **do** será siempre falsa no se realizará ninguna llamada al procedimiento swap. En el peor caso (cuando el arreglo inicial esté ordenado al revés), la comparación a[j] < a[j-1] será siempre verdadera y se realizarán i swaps por cada llamada a insert(a,i). En total serán $\frac{n^2}{2} - \frac{n}{2}$ (es decir, del orden de n^2) swaps. El número de intercambios en el caso promedio (asumiendo que los valores en las celdas del arreglo fueron generados al azar) es del orden de n^2 (nuevamente la prueba requiere conocimientos de probabilidades).

Este análisis demuestra que en el caso de la ordenación por inserción, la operación de intercambio es representativa del comportamiento del algoritmo en el peor caso y en el caso promedio. No así en el mejor caso en que las comparaciones son del orden de n pero no hay ningún intercambio.

Resolviendo el problema del bibliotecario. Si bien en el peor caso este algoritmo se comporta igual que el de ordenación por selección, presenta un mejor caso en que su comportamiento es del orden de n. Por ello, si los expedientes están casi ordenados, este algoritmo hará del orden de n comparaciones (y casi ningún swap). Con ese número de comparaciones, ordenar 2000 expedientes llevaría aproximadamente el doble que ordenar 1000.

[4] Ejercicio: ¿por qué?

Sin embargo el enunciado del problema del bibliotecario no asumía que los expedientes se encontraran casi ordenados, por lo que habría que considerar el caso promedio. Dijimos que el algoritmo de ordenación por inserción es en este caso del orden de n^2. Por consiguiente obtenemos la misma respuesta que para el algoritmo de ordenación por selección: ordenar 2000 expedientes es 4 veces más trabajo que ordenar 1000 expedientes.

A pesar de ello, en virtud del mejor caso que presenta la ordenación por inserción, podríamos afirmar que es preferible frente a la ordenación por selección. Por ejemplo, si un programa mantiene una base de datos a la que cada tanto se le agregan registros y cada tanto se la ordena, puede ser muy conveniente utilizar para su ordenación el algoritmo de ordenación por inserción, ya que al ser ordenada la base con cierta frecuencia, su comportamiento se asemejará a su mejor caso. No sería del todo extraño que éste sea el caso del bibliotecario, ya que las bibliotecas acostumbran mantener sus objetos (bastante) ordenados.

Análisis de Algoritmos

El ejemplo ilustra varios aspectos del análisis del comportamiento de un algoritmo:

casos: no siempre es posible contar exactamente. Se distingue entre mejor caso, peor caso y caso promedio. El estudio del peor caso permite establecer una cota superior, una garantía de comportamiento. El caso medio revela el comportamiento en la práctica, pero es más difícil de establecer. El mejor caso expresa apenas un comportamiento posible.

operación elemental: se elige una operación elemental y se cuenta el número de veces que la misma se repite durante la ejecución del algoritmo. Ésta debe:

- ser de duración constante, es decir, su duración no debe depender del tamaño de la entrada n,
- ser representativa del comportamiento, es decir, toda otra operación debe repetirse a lo sumo en forma proporcional a la operación elemental elegida. En la ordenación por selección, el intercambio (swap) no era una operación representativa.

aproximación: se ignoran constantes multiplicativas y los términos que resultan despreciables cuando n crece. Esto puede justificarse de varias maneras:

- la duración de la operación elemental depende del hardware, mejor hardware modificaría esas constantes.
- uno cuenta operaciones, pero no hace precisa la unidad de tiempo, no se sabe si cuenta segundos, días, microsegundos, años, etc. No estando determinada la unidad, las constantes multiplicativas pierden sentido.
- uno puede querer comparar 2 algoritmos cuyos análisis fueron hechos eligiendo operaciones elementales distintas en uno y en otro, sin saber a priori si dichas operaciones elementales tienen igual o diferente duración.

Pero hay que tener presente que se está haciendo una aproximación. Las constantes y términos que acabamos de llamar "despreciables" pueden ser significativos para valores suficientemente bajos de n, o en casos en que se pretende realizar un análisis más fino, más detallado, más preciso.

La Notación $\mathcal{O}$

Sean $c_s(n)$ el número de comparaciones de la ordenación por selección para arreglos de tamaño n y $c_i(n)$ el número de comparaciones de la ordenación por inserción para arreglos de tamaño n. Vimos que $c_s(n) = \frac{n^2}{2} - \frac{n}{2}$ y $n - 1 \leq c_i(n) \leq \frac{n^2}{2} - \frac{n}{2}$.

Decimos que $c_s(n)$ es del orden de n^2 o cuadrático. También se suele decir que $c_i(n)$ es del orden de n^2, aunque en realidad el número de comparaciones de la ordenación por inserción puede ser significativamente menor en ciertos casos (puede ser del orden de n, por ejemplo). Notacionalmente se puede hacer una diferencia para expresar cuando el orden es exacto y cuando el orden es una cota. Por ejemplo, escribiremos $c_s(n) \in \Theta(n^2)$ para expresar que el orden de $c_s(n)$ es **exactamente** cuadrático. Significa que para n suficientemente grande $c_s(n)$ es proporcional a n^2. En cambio, escribiremos $c_i(n) \in \mathcal{O}(n^2)$ para expresar que el orden de $c_i(n)$ es **a lo sumo** cuadrático, o sea, que para n suficientemente grande $c_i(n)$ es a lo sumo proporcional a n^2. También escribiremos $c_i(n) \in \Omega(n)$ para expresar que el orden de $c_i(n)$ es **como mínimo** n o lineal, es decir, que para n suficientemente grande $c_i(n)$ es como mínimo proporcional a n.

A continuación definiremos formalmente $\mathcal{O}$, Ω y Θ para justificar la notación utilizada en el párrafo anterior ($c_s(n) \in \Theta(n^2)$, $c_i(n) \in \mathcal{O}(n^2)$ y $c_i(n) \in \Omega(n)$). Vale la pena observar que estamos interesados en el comportamiento de las funciones para n suficientemente grande. Utilizaremos frecuentemente la expresión "para todo $n \in \mathbb{N}$ suficientemente grande, $\mathcal{P}(n)$" o "para casi todo $n \in \mathbb{N}$, $\mathcal{P}(n)$", que denotamos $\forall^\infty n \in \mathbb{N}.\mathcal{P}(n)$ y es equivalente a $\exists n_0 \in \mathbb{N}.\forall n \in \mathbb{N}.(n \geq n_0 \Rightarrow \mathcal{P}(n))$.

En particular nos interesa trabajar con funciones que sean no negativas en casi todo su dominio: denotamos por $f : \mathbb{N} \overset{\infty}{\to} \mathbb{R}^{\geq 0}$ que $\forall^\infty n \in \mathbb{N}.f(n) \in \mathbb{R}^{\geq 0}$.

Definición: Dada $f : \mathbb{N} \overset{\infty}{\to} \mathbb{R}^{\geq 0}$, se define el conjunto

$$\begin{aligned}
\mathcal{O}(f(n)) &= \{t : \mathbb{N} \overset{\infty}{\to} \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+.\forall^\infty n \in \mathbb{N}.t(n) \leq cf(n)\} \\
\Omega(f(n)) &= \{t : \mathbb{N} \overset{\infty}{\to} \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+.\forall^\infty n \in \mathbb{N}.t(n) \geq cf(n)\} \\
\Theta(f(n)) &= \mathcal{O}(f(n)) \cap \Omega(f(n))
\end{aligned}$$

que denotan, respectivamente, el conjunto de todas las funciones que para n suficientemente grande son a lo sumo proporcionales a $f(n)$, el de las que son como mínimo proporcionales a $f(n)$, y el de las que son proporcionales a $f(n)$.

Por ejemplo, regresando a la ordenación por selección, $c_s(n) = \frac{n^2}{2} - \frac{n}{2} \in \mathcal{O}(n^2)$, dado que $c_s(n) \leq n^2$ para todo $n \in \mathbb{N}$. También puede comprobarse que $c_s(n) \in \Omega(n^2)$, y por lo tanto $c_s(n) \in \Theta(n^2)$.

Otro ejemplo interesante lo da la función $t(n) = 5n^2 + 300n + 15$. A pesar de las constantes 5, 300 y 15, se comprueba como en el ejemplo anterior que $t(n) \in \Theta(n^2)$.

En cambio, para la ordenación por inserción, puede comprobarse que $c_i(n) \in \mathcal{O}(n^2)$ pero $c_i(n) \notin \Omega(n^2)$ y por ello $c_i(n) \notin \Theta(n^2)$. También se obtiene $c_i(n) \in \Omega(n)$ pero $c_i(n) \notin \mathcal{O}(n)$ y por ello $c_i(n) \notin \Theta(n)$.

A partir de ahora, concentraremos la atención solamente en $\mathcal{O}$. Esto se debe a que es el más utilizado de los tres, y que tanto Ω como Θ pueden ser expresados en términos de $\mathcal{O}$. En efecto, Θ está definido en términos de $\mathcal{O}$ y Ω, y además se tiene la siguiente proposición.

Proposición: $f(n) \in \mathcal{O}(g(n))$ sii $g(n) \in \Omega(f(n))$.

Demostración: $\Rightarrow$) Sea $c \in \mathbb{R}^+$ tal que $f(n) \leq cg(n)$ se cumple para casi todo $n \in \mathbb{N}$. Entonces, $g(n) \geq \frac{1}{c} f(n)$ y $\frac{1}{c} \in R^+$. $\Leftarrow$) similar.

También podemos comprobar que la definición de $\mathcal{O}$ determina que las constantes multiplicativas sean ignoradas. Esta propiedad es esperada ya que si d_1 y d_2 son positivas, $d_1 t(n)$ y $d_2 f(n)$ son proporcionales a $t(n)$ y $f(n)$ respectivamente:

Proposición: Si $t(n) \in \mathcal{O}(f(n))$, entonces $d_1 t(n) \in \mathcal{O}(d_2 f(n))$ para todo $d_1, d_2 \in \mathbb{R}^+$.

Demostración: Sea $c \in \mathbb{R}^+$ tal que $t(n) \leq cf(n)$ se cumple para casi todo $n \in \mathbb{N}$. Entonces, $d_1 t(n) \leq d_1 cf(n) = (\frac{d_1 c}{d_2}) d_2 f(n)$.

Otra propiedad que expresa la insignificancia de las constantes multiplicativas es que $df(n) \in \mathcal{O}(f(n))$ para todo $d > 0$. Sigue de la proposición anterior y reflexividad.

Proposición: La relación "es a lo sumo del orden de" es reflexiva y transitiva.

Demostración: Reflexividad: como $\forall n \in \mathbb{N}, f(n) \leq f(n)$ trivialmente, tomando $c = 1$ tenemos $f(n) \in \mathcal{O}(f(n))$.

Transitividad: Sean $c_1, c_2 \in \mathbb{R}^+$ tales que $\forall^\infty n \in \mathbb{N}, f(n) \leq c_1 g(n)$ y $\forall^\infty n \in \mathbb{N}, g(n) \leq c_2 h(n)$. Los naturales que satisfacen ambas desigualdades siguen siendo todos salvo a lo sumo un número finito. Por lo tanto, $\forall^\infty n \in \mathbb{N}, f(n) \leq c_1 g(n) \leq (c_1 c_2) h(n)$.

Corolario: $g(n) \in \mathcal{O}(h(n))$ sii $\mathcal{O}(g(n)) \subseteq \mathcal{O}(h(n))$.

Demostración: El "sólo si" se cumple por transitividad, y el "si" por reflexividad.

Corolario: $f(n) \in \mathcal{O}(g(n)) \wedge g(n) \in \mathcal{O}(f(n))$ sii $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$.

Observar que esto no significa que la relación "es a lo sumo del orden de" sea antisimétrica, de hecho no lo es. Antisimetría debería establecer que $f = g$, el corolario sólo dice que son proporcionales.

La siguiente proposición establece que para las funciones "interesantes" las constantes aditivas no importan.

Proposición: Si $f(n)$ está acotado inferiormente por un número positivo $(\exists \varepsilon > 0. \forall^\infty n \in \mathbb{N}. f(n) > \varepsilon)$, entonces $g(n) \in \mathcal{O}(f(n))$ implica $g(n) + d \in \mathcal{O}(f(n))$ para todo $d \geq 0$.

Demostración: Sean $\varepsilon, c > 0$ tales que $\forall^\infty n \in \mathbb{N}. f(n) > \varepsilon$ y $\forall^\infty n \in \mathbb{N}. g(n) \leq cf(n)$. Sea $c' = c + d/\varepsilon$. Ahora $\forall^\infty n \in \mathbb{N}. g(n) + d \leq cf(n) + \frac{d}{\varepsilon}\varepsilon \leq cf(n) + \frac{d}{\varepsilon} f(n) \leq c' f(n)$.

Proposición: Para todo $a, b > 1$, $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$.

Demostración: Para todo $n \in \mathbb{N}^+$, $\log_a n = \log_a b \ \log_b n$. Luego, $\log_a b$ es la constante (positiva por $a, b > 1$) que prueba que $\log_a n \in \mathcal{O}(\log_b n)$. Igual para $\log_b n \in \mathcal{O}(\log_a n)$.

Esto demuestra que en este contexto la base del logaritmo es irrelevante y puede ignorarse.

Regla del Límite

La siguiente regla es útil para demostrar cuándo una función es de un cierto orden, y cuándo no lo es.

Proposición (Regla de Límite): $\forall f, g \in \mathbb{N} \xrightarrow{\infty} \mathbb{R}^{\geq 0}$, si $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ existe los siguientes enunciados se cumplen:

(1) $\lim_{n\to\infty} \frac{f(n)}{g(n)} \in \mathbb{R}^+ \Rightarrow \mathcal{O}(f(n)) = \mathcal{O}(g(n))$.

(2) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow \mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$.

(3) $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty \Rightarrow \mathcal{O}(g(n)) \subset \mathcal{O}(f(n))$.

Observar que en los dos últimos casos, la inclusión es estricta.

Demostración:

(1) Alcanza con comprobar que $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$, dado que si $\lim_{n\to\infty} \frac{f(n)}{g(n)} = l \in \mathbb{R}^+$ entonces $\lim_{n\to\infty} \frac{g(n)}{f(n)} = \frac{1}{l} \in \mathbb{R}^+$. Veamos entonces que $f(n) \in \mathcal{O}(g(n))$. Sea $\varepsilon \in \mathcal{R}^+$, se tiene que $\forall^\infty n \in \mathbb{N}. \frac{f(n)}{g(n)} - l < \varepsilon$, luego $f(n) < (\varepsilon + l)g(n)$.

(2) El razonamiento anterior funciona incluso si $l = 0$. Tenemos pues $\mathcal{O}(f(n)) \subseteq \mathcal{O}(g(n))$. Veamos que esta inclusión es estricta verificando que $g(n) \notin \mathcal{O}(f(n))$. Si $g(n) \in \mathcal{O}(f(n))$, entonces hay una constante $c \in \mathbb{R}^+, \forall^\infty n \in \mathbb{N}, g(n) \leq cf(n)$. Entonces $\forall^\infty n \in \mathbb{N}. \frac{f(n)}{g(n)} \geq \frac{1}{c}$, con ello $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ no podría ser menor que $\frac{1}{c}$. Luego $\mathcal{O}(f(n)) \subset \mathcal{O}(g(n))$.

(3) Como $\lim_{n\to\infty} \frac{f(n)}{g(n)} = +\infty$ implica que $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$, vale por (2).

Corolario:

(1) $x < y \Rightarrow \mathcal{O}(n^x) \subset \mathcal{O}(n^y)$,
(2) $x \in \mathbb{R}^+ \Rightarrow \mathcal{O}(\log n) \subset \mathcal{O}(n^x)$,
(3) $x \in \mathbb{R}^{\leq 0} \Rightarrow \mathcal{O}(n^x) \subset O(\log n)$,
(4) $c > 1 \Rightarrow \mathcal{O}(n^x) \subset \mathcal{O}(c^n)$.
(5) $0 \leq c < 1 \Rightarrow \mathcal{O}(c^n) \subset \mathcal{O}(n^x)$.
(6) $c > d \geq 0 \Rightarrow \mathcal{O}(d^n) \subset \mathcal{O}(c^n)$.
(7) $d \geq 0 \Rightarrow \mathcal{O}(d^n) \subset \mathcal{O}(n!)$.
(8) $\mathcal{O}(n!) \subset \mathcal{O}(n^n)$.

Demostración:

(1) $\lim_{n\to\infty} \frac{n^x}{n^y} = \lim_{n\to\infty} \frac{1}{n^{y-x}} = 0$.
(2) Como $\lim_{n\to\infty} \log_a n = +\infty = \lim_{n\to\infty} n^x$, por la Regla de L'Hôpital tenemos $\lim_{n\to\infty} \frac{\log_a n}{n^x} = \lim_{n\to\infty} \frac{\frac{1}{n \ln a}}{xn^{x-1}} = \lim_{n\to\infty} \frac{1}{xn^x \ln a} = 0$.
(3) $\lim_{n\to\infty} \frac{\log n}{n^0} = \lim_{n\to\infty} \frac{\log n}{1} = \lim_{n\to\infty} \log n = +\infty$. Luego, $\mathcal{O}(n^0) \subset O(\log n)$. Para todo $x \in \mathbb{R}^-$, $\mathcal{O}(n^x) \subset \mathcal{O}(n^0) \subset O(\log n)$.
(4) Demostraremos por inducción que para todo $k \in \mathbb{N}$, $\lim_{n\to\infty} \frac{n^k}{c^n} = 0$. Para $k = 0$ la prueba es trivial. Asumimos como hipótesis inductiva que $\lim_{n\to\infty} \frac{n^k}{c^n} = 0$. Por la Regla de L'Hôpital, $\lim_{n\to\infty} \frac{n^{k+1}}{c^n} = \lim_{n\to\infty} \frac{(k+1)n^k}{c^n \log_e c} = \frac{k+1}{\log_e c} \lim_{n\to\infty} \frac{n^k}{c^n} = 0$. Por lo tanto, para todo $k \in \mathbb{N}$, $\mathcal{O}(n^k) \subset \mathcal{O}(c^n)$. Sea $x \in \mathbb{R}$. Sea $k \in \mathbb{N}$ tal que $x \leq k$. Se obtiene $\mathcal{O}(n^x) \subseteq \mathcal{O}(n^k) \subset \mathcal{O}(c^n)$.
(5) Similar al anterior, demostrando que para todo $k \in \mathbb{N}$, $\lim_{n\to\infty} \frac{n^k}{c^n} = +\infty$.
(6) Sigue de $\lim_{n\to\infty} \frac{c^n}{d^n} = \lim_{n\to\infty} (\frac{c}{d})^n = +\infty$, que vale pues $\frac{c}{d} > 1$.
(7) Sea $c > d$. Para todo $n \geq 2c^2$ se puede ver que $n! \geq n(n-1)\dots(n - \lfloor \frac{n}{2} \rfloor) \geq \lceil \frac{n}{2} \rceil^{\lceil \frac{n}{2} \rceil} \geq c^{2\lceil \frac{n}{2} \rceil} \geq c^n$. Por lo tanto, $c^n \in \mathcal{O}(n!)$ que implica $\mathcal{O}(c^n) \subseteq \mathcal{O}(n!)$. Por el inciso anterior, tenemos $\mathcal{O}(d^n) \subset \mathcal{O}(n!)$.
(8) Para todo $n \geq 2$, $\frac{n!}{n^n} = \frac{1}{n} * \frac{2*\dots*n}{n*\dots*n} \leq \frac{1}{n} * \frac{n*\dots*n}{n*\dots*n} = \frac{1}{n}$. Luego, $\lim_{n\to\infty} \frac{n!}{n^n} \leq \lim_{n\to\infty} \frac{1}{n} = 0$ y $\mathcal{O}(n!) \subset \mathcal{O}(n^n)$.

Los incisos 3 y 5 demostrados son de poco interés ya que no es esperable tener programas cuyo número de comparaciones decrece a medida que n crece. Se enunciaron para proporcionar una visión un poco más completa de la jerarquía de funciones determinada por la notación $\mathcal{O}$.

Proposición: $g(n) \in \mathcal{O}(f(n)) \Rightarrow \mathcal{O}(f(n)+g(n)) = \mathcal{O}(f(n))$.

Demostración: Para $c \in \mathbb{R}^+, \forall^\infty n \in \mathbb{N}. g(n) \leq cf(n)$. Luego, $f(n)+g(n) \leq (1+c)f(n)$.

Proposición: Sean $f, g : \mathbb{N} \stackrel{\infty}{\to} \mathbb{R}^{\geq 0}$. Si $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$, entonces $f(n) - g(n) : \mathbb{N} \stackrel{\infty}{\to} \mathbb{R}^{\geq 0}$ y $\mathcal{O}(f(n) - g(n)) = \mathcal{O}(f(n))$.

Demostración: $\forall^\infty n \in \mathbb{N}. \frac{g(n)}{f(n)} < \frac{1}{2}$. Así, $g(n) < \frac{1}{2}f(n)$ y $\frac{1}{2}f(n) \leq f(n) - g(n) \leq f(n)$.

Corolario: $\mathcal{O}(a_k n^k + \ldots + a_1 n + a_0) = \mathcal{O}(n^k)$, si $a_k \neq 0$.

Demostración: Usando las dos últimas proposiciones.

Proposición: $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\max(f(n), g(n)))$.

Demostración: $f(n) + g(n) \leq 2\max(f(n), g(n))$ y $\max(f(n), g(n)) \leq f(n) + g(n)$.

Proposición: Si $\forall^\infty n \in \mathbb{N}. f(n) > 0$, entonces $\mathcal{O}(g(n)) \subseteq \mathcal{O}(h(n))$ sii $\mathcal{O}(f(n)g(n)) \subseteq \mathcal{O}(f(n)h(n))$.

Demostración: Fácil pues $\forall^\infty n \in \mathbb{N}$, se verifica $g(n) \leq ch(n)$ sii $f(n)g(n) \leq cf(n)h(n)$.

Proposición: Si $f(n) \in \mathcal{O}(g(n))$ y $\lim_{n\to\infty} h(n) = \infty$, entonces $f(h(n)) \in \mathcal{O}(g(h(n)))$.

Demostración: Si $\forall^\infty n \in \mathbb{N}. f(n) \leq cg(n)$, como a partir de cierto n, $h(n)$ es suficientemente grande, $\forall^\infty n \in \mathbb{N}. f(h(n)) \leq cg(h(n))$.

Ejemplo: Como $n^{1/2} \in \mathcal{O}(n)$ y $log(n)$ tiende a infinito, $\sqrt{\log n} = \log^{1/2} n \in \mathcal{O}(\log n)$.

Definición: Si $\mathcal{O}(t(n)) = \mathcal{O}(1)$, t se dice **constante**. Si $\mathcal{O}(t(n)) = \mathcal{O}(\log n)$, t se dice **logarítmico**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n)$, t se dice **lineal**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n^2)$, t se dice **cuadrática**. Si $\mathcal{O}(t(n)) = \mathcal{O}(n^3)$, t se dice **cúbica**. Si $\mathcal{O}(n^k) \subset \mathcal{O}(t(n)) \subseteq \mathcal{O}(n^{k+1})$ para algún $k \in \mathbb{N}$, t se dice **polinomial**. Si $\mathcal{O}(t(n)) = \mathcal{O}(c^n)$, para algún $c > 1$, t se dice **exponencial**. Si $t(n)$ expresa la eficiencia de un algoritmo, éste se dice, respectivamente, constante, logarítimico, lineal, cuadrático, cúbico, polinomial, exponencial.

Proposición:

(1) $f(n) \in \Omega(g(n))$ sii $g(n) \in \mathcal{O}(f(n))$ sii $\mathcal{O}(g(n)) \subseteq \mathcal{O}(f(n))$ sii $\Omega(f(n)) \subseteq \Omega(g(n))$
(2) $f(n) \in \Theta(g(n))$ sii $g(n) \in \Theta(f(n))$ sii $\mathcal{O}(f(n)) = \mathcal{O}(g(n))$ sii $\Omega(f(n)) = \Omega(g(n))$

Ejemplos. A continuación se ven dos ejemplos de algoritmos de búsqueda: búsqueda lineal y búsqueda binaria.

El de búsqueda lineal es el siguiente, que recorre el arreglo de izquierda a derecha buscando la primer ocurrencia de x.

```
{Pre: n ≥ 0}
fun linear_search (a: array[1..n] of T, x:T) ret i:nat
    i:= 1                                          {Inv: x no está en a[1,i)}
    do i ≤ n ∧ a[i] ≠ x → i:= i+1 od
end fun
{Post: x está en a sii i ≤ n ∧ x=a[i]}
```

Para este algoritmo es sencillo analizar el mejor caso, el peor caso y el caso medio. Sean $t_1(n)$, $t_2(n)$ y $t_3(n)$ las funciones que cuentan la cantidad de comparaciones con x en el mejor caso, peor caso y caso medio respectivamente.

mejor caso: Ocurre cuando x se encuentra en la primera posición del arreglo. Se realiza una comparación: $t_1(n) = 1 \in \Theta(1)$.

peor caso: Ocurre cuando x no se encuentra en el arreglo. Se realizan n comparaciones: $t_2(n) = n \in \Theta(n)$.

caso medio: A priori, depende de la probabilidad de que x esté en el arreglo y, en caso de que esté, la probabilidad de que esté en cada posición. Si consideramos el caso en que está, y si consideramos equiprobable que esté en una u otra posición, el promedio del número de comparaciones que hará falta en cada caso es $t_3(n) = \frac{1+2+...+(n-1)+n}{n}$. Simplificando: $t_3(n) = \frac{n(n+1)}{2n} = \frac{n+1}{2} \in \Theta(n)$.

Observar que $t_1(n), t_2(n)$ y $t_3(n)$ cuentan el número de comparaciones en diferentes casos. Si definiéramos $t(n)$ como el número de comparaciones que realiza el algoritmo para arreglos de tamaño n (sin especificar los diferentes casos), obtendríamos $t(n) \in \mathcal{O}(n)$ y $t(n) \in \Omega(1)$. Es similar al análisis que hicimos del algoritmo de ordenación por inserción.

Ejercicio: ¿Cómo puede modificarse la búsqueda lineal si se asume que el arreglo está ordenado? ¿De qué orden serían $t_1(n), t_2(n)$ y $t_3(n)$ en tal caso?

El segundo ejemplo es el de búsqueda binaria, que requiere que el arreglo esté ordenado de menor a mayor. Es similar a cuando uno busca una palabra en un diccionario: uno lo abre al medio y (a menos que tenga la suerte de encontrarla justo donde abrió el diccionario, en cuyo caso la búsqueda termina) si la palabra que uno busca es anterior a las que se ven donde se abrió el diccionario uno limita la búsqueda a la parte izquierda (abriendo nuevamente al medio, etc), si en cambio la palabra que uno busca es posterior a las que se ven, uno limita la búsqueda a la parte derecha, etc.

```
{Pre: n ≥ 0 ∧ a ordenado}
fun binary_search (a: array[1..n] of T, x:T) ret i:nat
    var izq,med,der: nat
    izq:= 1
    der:= n
    i:= 0                {Inv: (x está en a sii x está en a[izq,der]) ∧ (i ≠ 0 ⇒ x = a[i])}
    do izq ≤ der ∧ i = 0 →
       med:= (izq+der) ÷ 2
       if x < a[med] → der:= med-1
          x = a[med] → i:= med
          x > a[med] → izq:= med+1
       fi
    od
end fun
{Post: (x está en a sii i ≠ 0) ∧ (i ≠ 0 ⇒ x = a[i]))}
```

La complejidad del algoritmo está dada por la cantidad de veces que se ejecuta el ciclo, dado que cada ejecución del ciclo insume tiempo constante (a lo sumo 2 comparaciones). En cada ejecución del ciclo, o bien se encuentra x, o bien el espacio de búsqueda se

reduce a la mitad. En efecto, si izq_j y der_j denotan los valores de izq y der después de la j-ésima ejecución del ciclo, sabemos que $d_j = \text{der}_j - \text{izq}_j + 1$ son la cantidad de celdas que nos quedan por explorar para encontrar x. Se puede ver que si $d_j > 1$, entonces $d_{j+1} \leq d_j/2$. Como $d_0 = \text{n}$ y el ciclo termina cuando $d_j = 1$, esto pasará -en el peor caso- cuando $j = \lceil \log_2 n \rceil$. Por lo tanto, si $t(n)$ es el número de comparaciones que se realizan en el peor caso, $t(n) \in \mathcal{O}(\log n)$.

De la misma forma puede obtenerse una cota inferior de $t(n)$ (es decir, del peor caso), por ejemplo, demostrando que $t(n) \geq \lfloor \log_4 n \rfloor$ partiendo de que $d_{j+1} \geq d_j/4$. Por ello, se tiene que $t(n) \in \Omega(\log n)$ y por lo tanto $t(n) \in \Theta(\log n)$.

Puede notarse que este algoritmo se comporta mucho mejor que el anterior para grandes valores de n. Supongamos una búsqueda lineal sobre un arreglo de tamaño 1.000.000. Si ahora repitiéramos la búsqueda lineal sobre uno de tamaño 2.000.000, dado que $t_2(n) \in \mathcal{O}(n)$, nos llevaría el doble de tiempo. En cambio, si estuviéramos utilizando búsqueda binaria, como $t(n) \in \mathcal{O}(\log n)$, la diferencia de tardanza sería casi imperceptible, dado que $\log(2n) = \log 2 + \log n = 1 + \log n$. En efecto, se puede observar que el algoritmo hace sólo una comparación más.

Recurrencias

El algoritmo de búsqueda binaria que acabamos de dar, puede también definirse por recursión:

```
{Pre: 0 ≤ izq ≤ der ≤ n}
fun binary_search_rec (a: array[1..n] of T, x:T, izq, der : nat) ret i:nat
    var med: int
    if izq > der → i = 0
      izq ≤ der → med:= (izq+der) ÷ 2
                  if x < a[med] → i:= binary_search_rec(a, x, izq, med-1)
                     x = a[med] → i:= med
                     x > a[med] → i:= binary_search_rec(a, x, med+1,der)
                  fi
    fi
end fun
{Post: (x está en a[izq,der] sii i ≠ 0) ∧ (i ≠ 0 ⇒ x = a[i])}
```

Hace falta una función principal:

```
{Pre: n ≥ 0 }
fun binary_search (a: array[1..n] of T, x:T) ret i:int
    i:= binary_search_rec(a, x, 1, n)
end fun
{Post: (x está en a sii i ≠ 0) ∧ (i ≠ 0 ⇒ x = a[i]))}
```

El algoritmo es esencialmente el mismo que en la versión iterativa, veremos a continuación cómo calcular el orden de algoritmos recursivos como éste. Sea $t(n)$ el número de comparaciones entre elementos de a que realiza binary_search_rec(a,x,izq,der) cuando n es igual a der - izq + 1. Por simplicidad, consideraremos que evaluar las condiciones del **if** requiere una comparación (frecuentemente es posible implementarlo de esa forma).

Entonces, a menos que izq sea mayor que der en cuyo caso n es cero y no hay comparaciones entre elementos de a ($t(0) = 0$), la ejecución de binary_search_rec(a,x,izq,der) realiza una comparación y:

- o bien x = a[med], en cuyo caso el algortimo termina,
- o bien x < a[med], en cuyo caso hay una llamada recursiva,
- o bien x > a[med], en cuyo caso también hay una llamada recursiva,

Sabiendo que $t(n)$ es el número de comparaciones entre elementos de a que realiza binary_search(a,x,izq,der) cuando n es igual a der - izq +1, tenemos:

$$t(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n \neq 0 \wedge \text{x} = \text{a[med]} \\ 1 + t(n \div 2) & \text{caso contrario} \end{cases}$$

En el peor caso, $t(n) = 1 + t(n \div 2) = 1 + 1 + t(n \div 4) = \ldots = j + t(n \div 2^j)$. Para $j = \log_2 n$ obtenemos $t(n) = j + t(1) = j = \log_2 n$.

Ordenación por intercalación. Recordemos el problema formulado en una clase anterior:

> **Pregunta 4:** Un bibliotecario demora 1 día en ordenar alfabéticamente una biblioteca con 1000 expedientes. ¿Cuánto demorará en ordenar una con 2000 expedientes?

Le proponemos al bibliotecario que ordene la primera mitad de la biblioteca (tarea A $\approx$ 1.000.000 de comparaciones = 1 día de trabajo), luego la segunda mitad (tarea B $\approx$ 1.000.000 de comparaciones = 1 día de trabajo), y luego efectúe la intercalación obvia entre las 2 bibliotecas ordenadas (tarea C $\approx$ 2.000 comparaciones = unos minutos de trabajo). Esto le llevará mucho menos que los 4 días que le llevaría hacer todo con el algoritmo de ordenación por selección.

Esto a su vez puede mejorarse: la tarea A puede desdoblarse en ordenar 500 expedientes (tarea AA $\approx$ 250.000 comparaciones), ordenar los otros 500 expedientes (tarea AB $\approx$ 250.000 comparaciones), e intercalar (tarea AC $\approx$ 1.000 comparaciones). Similarmente para la tarea B. En total, tendríamos ahora $\approx$ 1.004.000 comparaciones. ¡Poco más de 1 día para ordenar los 2.000 expedientes! La idea puede mejorarse aún más: en vez de ordenar grupos de 500, serán grupos de 250 ó 125 ó 68 ó 34 ó ...

¿Cuán chicos pueden ser estos grupos? Una biblioteca de 1 solo expediente es trivial, no requiere ordenación. Una de 2 expedientes ya puede subdivirse en 2 partes de un expediente cada una, ordenar cada parte (trivial) e intercalar.

De esta manera, la utilización de la ordenación por selección finalmente queda eliminada: toda la ordenación se realiza dividiendo la biblioteca en 2 ordenando (utilizando recursivamente esta idea) e intercalando.

El algoritmo de ordenación que acabamos de explicar se llama mergeSort (ordenación por intercalación). En un estilo funcional se puede escribir así:

```
merge_sort :: [T] → [T]
merge_sort [] = []
merge_sort [t] = [t]
merge_sort ts = merge sts1 sts2
```

```
            where
                sts1 = merge_sort ts1
                sts2 = merge_sort ts2
                (ts1,ts2) = split ts

split :: [T] → ([T],[T])
split ts = (take n ts, drop n ts)
        where n = length ts ÷ 2

merge :: [T] → [T] → [T]
merge [] sts2 = sts2
merge sts1 [] = sts1
merge (t1:sts1) (t2:sts2) = if t1 ≤ t2
                            then t1:merge sts1 (t2:sts2)
                            else t2:merge (t1:sts1) sts2
```

donde split divide ts en 2 partes de igual longitud (± 1) y merge realiza la intercalación. La administración de la memoria en estilo funcional se deja en manos del compilador. Eso facilita su definición. En estilo imperativo la principal dificultad es justamente la de realizar la intercalación: o bien es necesario complicar significativamente el algoritmo, o bien utilizar arreglos auxiliares para realizarla. Volviendo a nuestro ejemplo motivador, se puede ver que no es fácil intercalar 2 bibliotecas (cada una de ellas ordenada) llenas de expedientes si el resultado de la intercalación debe quedar en las mismas bibliotecas, a menos que se cuente con una biblioteca vacía donde poner los expediente temporariamente. Tomando b y c como arreglos temporarios para realizar la intercalación (en realidad sólo uno es necesario, incluimos el otro para simplificar el algoritmo de intercalación) se lo puede escribir de la siguiente manera:

```
{Pre: n ≥ der ≥ izq > 0 ∧ a = A}
proc merge_sort_rec (in/out a: array[1..n] of T, in izq,der: nat)
    var b,c: array[1..n] of T
    var med: nat
    if der > izq → med:= (der+izq) ÷ 2
                   merge_sort_rec(a,izq,med)
                                   {a[izq,med] permutación ordenada de A[izq,med]}
                   merge_sort_rec(a,med+1,der)
                                   {a[med+1,der] permutación ordenada de A[med+1,der]}
                   for i:= izq to med do b[i-izq+1]:=a[i] od
                                                   {b[1,med-izq+1] = a[izq,med]}
                   for j:= med+1 to der do c[j-med]:=a[j] od
                                                   {c[1,der-med] = a[med+1,der]}
                   intercalar(a,b,c,izq,med,der)
                                   {a[izq,der] permutación ordenada de A[izq,der]}
    fi
end proc
```

{Post: a permutación de A ∧ a[izq,der] permutación ordenada de A[izq,der]}

En realidad, los arreglos b y c no necesitan ser tan largos, pero por simplicidad se declaran de n celdas, igual que el arreglo a.

La ejecución de la ordenación por intercalación comienza llamando al procedimiento con izq igual a 1 y der igual a n:

```
{Pre: n ≥ 0 ∧ a = A}
proc merge_sort (in/out a: array[1..n] of T)
      merge_sort_rec(a,1,n)
end proc
{Post: a está ordenado y es permutación de A}
```

El procedimiento merge_sort_rec utiliza el procedimiento intercalar que puede programarse de la siguiente manera.

```
{Pre: n ≥ der > med ≥ izq > 0 ∧ a = A ∧ b = B ∧ c = C ∧
      B[1,med-izq+1] = A[izq,med] ∧ C[1,der-med] = A[med+1,der] ∧
      B[1,med-izq+1] y C[1,der-med] ordenados}
proc intercalar (in/out a: array[1..n] of T, in b, c: array[1..n] of T, in izq,med,der: nat)
      var i,j,maxi,maxj: nat
      i:= 1
      j:= 1
      maxi:= med-izq+1
      maxj:= der-med
                                        {Inv: a[izq,k) = intercalación de b[1,i) con c[1,j)}
      for k:= izq to der do
            if i ≤ maxi ∧ (j > maxj ∨ b[i] ≤ c[j]) then a[k]:= b[i]
                                                        i:=i+1
            else a[k]:= c[j]
                  j:=j+1
            fi
      od
end proc  {a permutación de A ∧ a[izq,der] permutación ordenada de A[izq,der]}
```

Esta técnica para resolver un problema, consistente en dividir el problema en problemas menores (de idéntica naturaleza pero de menor tamaño), asumir los problemas menores resueltos y utilizar dichos resultados para resolver el problema original, se conoce por "divide and conquer", es decir, "divide y vencerás", o "divide y reinarás". Justamente el hecho de que los problemas menores sean de igual naturaleza que el original, es lo que permite que el mismo algoritmo pueda aplicarse recursivamente para resolver los problemas menores. Los casos más sencillos (como el del fragmento de arreglo de longitud menor o igual que 1, para el problema de ordenación) deben resolverse aparte. Estos casos frecuentemente son triviales. Usualmente el término "divide y vencerás" se aplica a aquellos casos en que el problema se subdivide en problemas menores "fraccionando" el tamaño de la entrada.

Número de Comparaciones. Sea $t(n)$ el número de comparaciones entre elementos de **T** que realiza el procedimiento merge_sort_rec con una entrada de tamaño n

en el peor caso. Cuando la entrada es de tamaño 1, no hace ninguna comparación, $t(1) = 0$. Cuando la entrada es de tamaño mayor a 1, hace todas las comparaciones que hace la primera llamada recursiva al procedimiento merge_sort_rec, esto es, $t(\lceil n/2 \rceil)$, más todas las comparaciones que hace la segunda llamada recursiva al procedimiento merge_sort_rec, esto es, $t(\lfloor n/2 \rfloor)$, más todas las comparaciones que hace el proceso de intercalación en el peor caso, esto es, $n - 1$. Tenemos entonces:

$$t(n) = t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n - 1.$$

Esto es una recurrencia, es decir, se define la función t en términos de sí misma. ¿Cómo obtener explícitamente el orden de $t(n)$? Usaremos que $t(n)$ está acotada:

$$\begin{array}{rcl} t(n) & \leq & t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n \\ t(n) & \geq & t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + \frac{n}{2} \end{array}$$

Para simplificar el cálculo, consideramos primero aquellos valores de n para los que las operaciones $\lceil\ \rceil$ y $\lfloor\ \rfloor$ pueden ser ignoradas: potencias de 2.

Sea entonces $n = 2^m$. A partir de la primera desigualdad, para $m > 0$ tenemos $t(2^m) \leq t(2^{m-1}) + t(2^{m-1}) + 2^m = 2t(2^{m-1}) + 2^m$ de donde se obtiene dividiendo por 2^m,

$$\frac{t(2^m)}{2^m} \leq \frac{t(2^{m-1})}{2^{m-1}} + 1.$$

Iterando este proceso, se llega a

$$\frac{t(2^m)}{2^m} \leq \frac{t(2^{m-1})}{2^{m-1}} + 1 \leq \frac{t(2^{m-2})}{2^{m-2}} + 2 \leq \frac{t(2^{m-k})}{2^{m-k}} + k \leq \frac{t(2^0)}{2^0} + m = \frac{t(1)}{1} + m = 0 + m = m.$$

Despejando, $t(2^m) \leq 2^m m$. Como $n = 2^m$, sabemos que $m = \log_2 n$. Reemplazando queda $t(n) \leq n \log_2 n$, para todo n que sea potencia de 2. Podemos concluir entonces que $t(n) \in \mathcal{O}(n \log n | n \text{ potencia de } 2)$.

Análogamente, usando la segunda desigualdad obtenemos $t(n) \geq \frac{1}{2} n \log n$ y concluimos que $t(n) \in \Omega(n \log n | n \text{ potencia de } 2)$ y finalmente de ambas conclusiones, obtenemos que $t(n) \in \Theta(n \log n | n \text{ potencia de } 2)$.

Resta ver cuál es la cantidad de comparaciones para el resto de los valores de n.

Primero se puede ver que $t(n)$ es creciente. Por inducción en n, $t(n+1) > t(n)$. El caso base es sencillo, $t(2) = t(1) + t(1) + 2 - 1 = 1 > 0 = t(1)$. Supongamos que para todo $1 \leq k < n$, $t(k+1) > t(k)$. Entonces, $t(n+1) = t(\lceil (n+1)/2 \rceil) + t(\lfloor (n+1)/2 \rfloor) + n + 1 - 1 > t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n - 1 = t(n)$.

Para n suficientemente grande, sea $k \geq 1$ tal que $2^k \leq n < 2^{k+1}$. Esto implica que $k \leq \log_2 n < k + 1$. Como t es creciente, tenemos $t(n) < t(2^{k+1}) \leq 2^{k+1}(k+1) = 2(2^k k + 2^k) \leq 2(2^k k + 2^k k) = 4\ 2^k k \leq 4\ 2^{\log_2 n} \log_2 n = 4n \log_2 n$. Por lo tanto $t(n) \in \mathcal{O}(n \log n)$. También por t creciente, se tiene que $t(n) \geq t(2^k) \geq \frac{1}{2} 2^k k \geq \frac{1}{2} 2^{k+1}(k+1) > \frac{1}{8} 2^{\log_2 n} \log_2 n = \frac{1}{8} n \log_2 n$. Entonces $t(n) \in \Omega(n \log n)$, o sea, $t(n) \in \Theta(n \log n)$.

Al analizar el procedimiento merge_sort_rec utilizamos intuitivamente la siguiente notación.

Definición: Dada $f : \mathbb{N} \overset{\infty}{\to} \mathbb{R}^{\geq 0}$, se definen los conjuntos

$$\begin{aligned}
\mathcal{O}(f(n)|P(n)) &= \{t : \mathbb{N} \overset{\infty}{\to} \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+ . \forall^\infty n \in \mathbb{N} . P(n) \Rightarrow t(n) \leq cf(n)\} \\
\Omega(f(n)|P(n)) &= \{t : \mathbb{N} \overset{\infty}{\to} \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+ . \forall^\infty n \in \mathbb{N} . P(n) \Rightarrow t(n) \geq cf(n)\} \\
\Theta(f(n)|P(n)) &= \mathcal{O}(f(n)|P(n)) \bigcap \Omega(f(n)|P(n))
\end{aligned}$$

El razonamiento hecho para el análisis del procedimiento merge_sort_rec puede generalizarse:

Definición: Dada $f : \mathbb{N} \overset{\infty}{\to} \mathbb{R}^{\geq 0}$, se dice que f es **asintóticamente** o **eventualmente no decreciente** si $\forall^\infty n \in \mathbb{N}, f(n) \leq f(n+1)$. Sea $i \in \mathbb{N}^{\geq 2}$, f es **i-suave** o **i-uniforme** si es eventualmente no decreciente y $\exists d \in \mathbb{R}^+, \forall^\infty n \in \mathbb{N}, f(in) \leq df(n)$. Finalmente, f es **suave** o **uniforme** si es i-suave para todo $i \in \mathbb{N}^{\geq 2}$.

Ejemplos: En el análisis del procedimiento merge_sort_rec, $t(n)$ es eventualmente no decreciente. La función $n \log_2 n$ es eventualmente no decreciente (ya que es producto de dos funciones que lo son). También es 2-suave ya que para todo $n \geq 2$ se cumple $2n \log_2(2n) = 2n(\log_2 2 + \log_2 n) = 2n + 2n \log_2 n \leq 2n \log_2 n + 2n \log_2 n = 4n \log_2 n$ (en realidad, también podemos concluir que es 2-suave por ser producto de dos funciones que son 2-suave). Del resultado que sigue se desprende que $n \log_2 n$ es también suave.

Lema: f es i-suave si y sólo si f es suave.

Demostración: El "si" es trivial, demostramos el "sólo si" suponiendo que f es i-suave y demostrando que entonces también es j-suave. Sea n suficientemente grande,

$$\begin{aligned}
f(jn) &\leq f(i^{\lceil \log_i j \rceil} n) && f \text{ es eventualmente no decreciente y } jn \leq i^{\lceil \log_i j \rceil} n \\
&\leq d^{\lceil \log_i j \rceil} f(n) && f \text{ es } i\text{-suave}
\end{aligned}$$

por lo tanto, f es j-suave (con constante $d^{\lceil \log_i j \rceil}$).

Los siguientes son ejemplos de funciones suaves: n^k, $\log n$, $n^k \log n$, mientras que $n^{\log n}$, 2^n o $n!$ son ejemplos de funciones no suaves.

Regla de la suavidad o uniformidad: Sea $f : \mathbb{N} \overset{\infty}{\to} \mathbb{R}^{\geq 0}$ suave y $t : \mathbb{N} \overset{\infty}{\to} R^{\geq 0}$ eventualmente no decreciente, si $t(n) \in \mathcal{O}(f(n)|n \text{ potencia de } b)$, entonces $t(n) \in \mathcal{O}(f(n))$. Análogamente para Θ y Ω.

Demostración: Sea n suficientemente grande, y sea m tal que $b^m \leq n < b^{m+1}$. Entonces

$$\begin{aligned}
t(n) &\leq t(b^{m+1}) && t \text{ es eventualmente no decreciente} \\
&\leq cf(b^{m+1}) && t(n) \in \mathcal{O}(f(n)|n \text{ potencia de } b) \\
&\leq cdf(b^m) && f \text{ es } b\text{-suave} \\
&\leq cdf(n) && f \text{ es eventualmente no decreciente}
\end{aligned}$$

por lo tanto $t(n) \in \mathcal{O}(f(n))$.

Ejemplo: sea $t(n)$ el número de comparaciones que realiza el procedimiento merge_sort_rec: $t(n)$ es eventualmente no decreciente, $t(n) \in \Theta(n \log n | n \text{ potencia de } 2)$ y $n \log n$ es suave. Luego $t(n) \in \Theta(n \log n)$. Volviendo al problema del bibliotecario, ordenar 1000 expedientes requiere 1000*10 = 10000 comparaciones. Ordenar 2000 expedientes, en cambio, requiere 2000*11 = 22000 comparaciones. Por lo tanto, si 1000 expedientes requirieron 1 día de trabajo, 2000 requerirá poco más de 2 días.

Al estudiar las recurrencias en forma general, se desarrollarán técnicas generales para la resolución de recurrencias, de manera de no verse uno siempre obligado a realizar una prueba detallada como la que hicimos en el caso del procedimiento merge_sort_rec.

Recurrencias/relaciones "divide y vencerás" (divide and conquer). (Introduction to Algorithms: A Creative Approach, páginas 50 y 51)

(1) comprobar que $t(n)$ es eventualmente no decreciente.
(2) llevar la recurrencia a una ecuación de la forma $t(n) = at(n/b) + g(n)$ para $b \in \mathbb{N}^{\geq 2}$, $g(n) \in \Theta(n^k)$, $k \in \mathbb{N}$ y n **potencia de** b. Observar que $t(n)$ puede tener una expresión general más compleja, pero para el caso de n potencia de b, puede reducirse como la ecuación mencionada.
(3) según sea $a > b^k$, o $a = b^k$ o $a < b^k$, se obtienen los siguientes resultados para n potencia de b:

$$t(n) \in \begin{cases} \Theta(n^{\log_b a}) & \text{si} \quad a > b^k \\ \Theta(n^k \log n) & \text{si} \quad a = b^k \\ \Theta(n^k) & \text{si} \quad a < b^k \end{cases}$$

(4) Si la ecuación inicial no contiene una igualdad sino sólo una cota: $t(n) \leq at(n/b)+g(n)$ (resp. $t(n) \geq at(n/b)+g(n)$) para n **potencia de** b, $g(n) \in \mathcal{O}(n^k)$ (resp. $g(n) \in \Omega(n^k)$) se obtiene el mismo resultado en los 3 casos, sólo que escribiendo $\mathcal{O}$ (resp. Ω) en vez de Θ.

Ejemplos de recurrencias "divide y vencerás" son el número de comparaciones que realiza el procedimiento merge_sort, o el número de comparaciones que realiza la búsqueda binaria en el peor caso. El primer ejemplo fue desarrollado en detalle recientemente.

Para la búsqueda binaria hicimos las cuentas detalladamente, pero ahora podemos volver a hacerla utilizando recurrencias. Si pensamos en el número de comparaciones que hacen falta para buscar en un arreglo de longitud n, observamos que es 1 más el número de comparaciones que hacen falta para buscar en un arreglo de longitud $\lfloor n/2 \rfloor$. Eso nos dá $t(n) = t(\lfloor n/2 \rfloor) + 1$. Aplicando el método presentado más arriba, como $t(n)$ es eventualmente no decreciente y $a = 1$, $b = 2$ y $k = 0$, tenemos $a = b^k$ y por ello $t(n) \in \Theta(\log n)$.

Recurrencias lineales homogéneas. (Fundamentos de Algoritmia, páginas 135 a 140)

(1) llevar la recurrencia a una **ecuación característica** de la forma $a_k t_n + \ldots + a_0 t_{n-k} = 0$,
(2) considerar el **polinomio característico asociado** $a_k x^k + \ldots + a_0$,
(3) determinar las raíces $r_1, \ldots, r_j$ del polinomio característico, de multiplicidad $m_1, \ldots, m_j$ respectivamente (se tiene $m_i \geq 1$ y $m_1 + \ldots + m_j = k$),
(4) considerar la forma general de las soluciones de la ecuación característica:

$$\begin{array}{rcl} t(n) & = & c_1 r_1^n + c_2 n r_1^n + \ldots + c_{m_1} n^{m_1-1} r_1^n + \\ & + & c_{m_1+1} r_2^n + c_{m_1+2} n r_2^n + \ldots + c_{m_1+m_2} n^{m_2-1} r_2^n + \\ \vdots & \vdots & \vdots \\ & + & c_{m_1+\ldots+m_{j-1}+1} r_j^n + c_{m_1+\ldots+m_{j-1}+2} n r_j^n + \ldots + c_{m_1+\ldots+m_j} n^{m_j-1} r_j^n \end{array}$$

como $m_1 + \ldots + m_j = k$, tenemos k incógnitas: $c_1, \ldots, c_k$,
(5) con las k **condiciones iniciales** $t_{n_0}, \ldots, t_{n_0+k-1}$ (n_0 es usualmente 0 ó 1) plantear un sistema de k ecuaciones con k incógnitas:

$$\begin{array}{rcl} t(n_0) & = & t_{n_0} \\ t(n_0+1) & = & t_{n_0+1} \\ \vdots & \vdots & \vdots \\ t(n_0+k-1) & = & t_{n_0+k-1} \end{array}$$

(6) obtener de este sistema los valores de $c_1, \ldots, c_k$,
(7) escribir la **solución final** de la forma $t_n = t'(n)$, donde $t'(n)$ se obtiene a partir de $t(n)$ reemplazando c_i y r_i por sus valores y simplificando la expresión final.
(8) **corroborar** que efectivamente $t(n_0 + k) = t_{n_0+k}$, donde t_{n_0+k} puede obtenerse utilizando la ecuación característica.

Un ejemplo puede darse contando t_n, el número de veces que se ejecuta la acción A en el siguiente procedimiento cuando se llama con parámetro n:

```
proc p (in n: int)                                   {pre : n ≥ 0}
    if n = 0 → skip
       n = 1 → A
       n > 1 → p(n-1)
               p(n-2)
    fi
end proc
```

Al calcular t_n, obtenemos

$$t_n = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ t_{n-1} + t_{n-2} & \end{cases}$$

que es la secuencia de fibonacci.

Apliquemos el método presentado para las recurrencias homogéneas:

(1) ecuación característica $t_n - t_{n-1} - t_{n-2} = 0$, $k = 2$.
(2) polinomio característico asociado $x^2 - x - 1$.

(3) raíces $r_1 = \frac{1+\sqrt{5}}{2}$ de multiplicidad $m_1 = 1$ y $r_2 = \frac{1-\sqrt{5}}{2}$ de multiplicidad $m_2 = 1$.
(4) forma general $t(n) = c_1(\frac{1+\sqrt{5}}{2})^n + c_2(\frac{1-\sqrt{5}}{2})^n$.
(5) condiciones iniciales $t_0 = 0$ y $t_1 = 1$. Sistema de ecuaciones:

$$\begin{array}{rcll} c_1 + c_2 & = & 0 & (t(0) = t_0) \\ c_1(\frac{1+\sqrt{5}}{2}) + c_2(\frac{1-\sqrt{5}}{2}) & = & 1 & (t(1) = t_1) \end{array}$$

(6) despejando, $c_1 = \frac{1}{\sqrt{5}}$ y $c_2 = -\frac{1}{\sqrt{5}}$.
(7) solución final $t_n = \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^n - \frac{1}{\sqrt{5}}(\frac{1-\sqrt{5}}{2})^n$.
(8) efectivamente, con la solución final $t_2 = 1$ coincidiendo con el resultado obtenido para calcular t_2 usando la recurrencia original.

El siguiente ejemplo no proviene de un algoritmo. Calcular explícitamente t_n donde

$$t_n = \begin{cases} n & n = 0, 1, 2 \\ 5t_{n-1} - 8t_{n-2} + 4t_{n-3} & \end{cases}$$

usando la técnica presentada.

(1) ecuación característica $t_n - 5t_{n-1} + 8t_{n-2} - 4t_{n-3} = 0$, $k = 3$.
(2) polinomio característico asociado $x^3 - 5x^2 + 8x - 4$.
(3) raíces $r_1 = 1$ de multiplicidad $m_1 = 1$ y $r_2 = 2$ de multiplicidad $m_2 = 2$.
(4) forma general $t(n) = c_1 1^n + c_2 2^n + c_3 n 2^n$, simplificando $t(n) = c_1 + c_2 2^n + c_3 n 2^n$.
(5) condiciones iniciales $t_0 = 0$, $t_1 = 1$ y $t_2 = 2$. Sistema de ecuaciones:

$$\begin{array}{rcll} c_1 + c_2 & = & 0 & (t(0) = t_0) \\ c_1 + 2c_2 + 2c_3 & = & 1 & (t(1) = t_1) \\ c_1 + 4c_2 + 8c_3 & = & 2 & (t(2) = t_2) \end{array}$$

(6) despejando, $c_1 = -2$, $c_2 = 2$ y $c_3 = -1/2$.
(7) solución final $t_n = -2 + 2 * 2^n - \frac{1}{2}n2^n$, simplificando $t_n = 2^{n+1} - n2^{n-1} - 2$.
(8) efectivamente, con la solución final $t_3 = 2$ coincidiendo con el resultado obtenido para calcular t_3 usando la recurrencia original.

En clase resolvimos también el siguiente ejemplo:

$$t_n = \begin{cases} 0 & n = 0, 1, 2 \\ 8 & n = 3 \\ 7t_{n-1} - 18t_{n-2} + 20t_{n-3} - 8t_{n-4} & \end{cases}$$

Recurrencias no homogéneas. (Fundamentos de Algoritmia, páginas 140 a 148)

(1) llevar la recurrencia a una ecuación característica de la forma $a_k t_n + \ldots + a_0 t_{n-k} = b^n p(n)$, donde $p(n)$ es un polinomio no nulo de grado d.
(2) considerar el polinomio característico asociado $(a_k x^k + \ldots + a_0)(x-b)^{d+1}$,
(3) determinar las raíces $r_1, \ldots, r_j$ del polinomio característico, de multiplicidad $m_1, \ldots, m_j$ respectivamente (se tiene $m_i \geq 1$ y $m_1 + \ldots + m_j = k + d + 1$),
(4) considerar la forma general de las soluciones de la ecuación característica:

$$\begin{array}{rcl} t(n) & = & c_1 r_1^n + c_2 n r_1^n + \ldots + c_{m_1} n^{m_1-1} r_1^n + \\ & + & c_{m_1+1} r_2^n + c_{m_1+2} n r_2^n + \ldots + c_{m_1+m_2} n^{m_2-1} r_2^n + \\ \vdots & \vdots & \vdots \\ & + & c_{m_1+\ldots+m_{j-1}+1} r_j^n + c_{m_1+\ldots+m_{j-1}+2} n r_j^n + \ldots + c_{m_1+\ldots+m_j} n^{m_j-1} r_j^n \end{array}$$

como $m_1 + \ldots + m_j = k + d + 1$, tenemos $k + d + 1$ incógnitas: $c_1, \ldots, c_{k+d+1}$,
(5) a partir de las k condiciones iniciales $t_{n_0}, \ldots, t_{n_0+k-1}$ (n_0 es usualmente 0 ó 1), obtener usando la ecuación característica, los valores de $t_{n_0+k}, \ldots, t_{n_0+k+d}$,
(6) con los $k+d+1$ valores $t_{n_0}, \ldots, t_{n_0+k+d}$ plantear un sistema de $k+d+1$ ecuaciones con $k + d + 1$ incógnitas:

$$\begin{array}{rcl} t(n_0) & = & t_{n_0} \\ t(n_0+1) & = & t_{n_0+1} \\ \vdots & \vdots & \vdots \\ t(n_0+k+d) & = & t_{n_0+k+d} \end{array}$$

(7) obtener de este sistema los valores de $c_1, \ldots, c_{k+d+1}$,
(8) escribir la **solución final** de la forma $t_n = t'(n)$, donde $t'(n)$ se obtiene a partir de $t(n)$ reemplazando c_i y r_i por sus valores y simplificando la expresión final.
(9) **corroborar** que efectivamente $t(n_0 + k + d + 1) = t_{n_0+k+d+1}$, donde $t_{n_0+k+d+1}$ puede obtenerse utilizando la ecuación característica.

Ejemplo: calcular explícitamente t_n donde $t_n = \begin{cases} 0 & n = 0 \\ 2t_{n-1} + n & \end{cases}$
usando la técnica presentada para las recurrencias no homogéneas.

(1) ecuación característica $t_n - 2t_{n-1} = n$, $k = 1$, $b = 1$, $p(n) = n$, $d = 1$.
(2) polinomio característico asociado $(x-2)(x-1)^2$.
(3) raíces $r_1 = 2$ de multiplicidad $m_1 = 1$ y $r_2 = 1$ de multiplicidad $m_2 = 2$.
(4) forma general $t(n) = c_1 2^n + c_2 1^n + c_3 n 1^n$, simplificando $t(n) = c_1 2^n + c_2 + c_3 n$.
(5) condiciones iniciales $t_0 = 0$. También podemos obtener usando la recurrencia $t_1 = 2t_0 + 1 = 1$ y $t_2 = 2t_1 + 2 = 4$.
(6) sistema de ecuaciones:

$$\begin{array}{rcll} c_1 + c_2 & = & 0 & (t(0) = t_0) \\ 2c_1 + c_2 + c_3 & = & 1 & (t(1) = t_1) \\ 4c_1 + c_2 + 2c_3 & = & 4 & (t(2) = t_2) \end{array}$$

(7) despejando, $c_1 = 2$, $c_2 = -2$ y $c_3 = -1$.
(8) solución final $t_n = 2 * 2^n - 2 - n$, simplificando $t_n = 2^{n+1} - n - 2$.
(9) efectivamente, con la solución final $t_3 = 11$ coincidiendo con el resultado obtenido para calcular t_3 usando la recurrencia original.

Ordenación rápida (quick_sort). La ordenación por intercalación es la más rápida que hemos visto, realiza apenas $\Theta(n \log n)$ comparaciones. El inconveniente que tiene es la necesidad de un arreglo auxiliar para realizar la intercalación. La ordenación rápida logra evitar este problema. La idea es descomponer el problema en 2 subproblemas de manera que no requieran intercalación. ¿Cómo? Separando el arreglo en 2 fragmentos de manera de que todos los que se encuentren en el primer fragmento sean menores que todos los que se encuentren en el segundo fragmento. Para ello se elige un **pivote**, es decir, un elemento que se utilizará para separar ambos fragmentos. Aquellos elementos que sean menores o iguales al pivote pertenecerán al primer fragmento, aquéllos que no, al segundo. Llamaremos pivot al procedimiento que realiza dicha fragmentación. El procedimiento quick_sort invocará al procedimiento pivot.

```
{Pre: 1 ≤ izq < der ≤ n ∧ a = A}
proc pivot (in/out a: array[1..n] of elem, in izq, der: nat, out piv: nat)
    var i,j: nat
    piv:= izq
    i:= izq+1
    j:= der
    do i ≤ j →
                    {piv < i ≤ j+1 ∧ todos los elementos en a[izq,i) son ≤ que a[piv]}
                                {∧ todos los elementos en a(j,der] son > que a[piv]}
              if a[i] ≤ a[piv] → i:= i+1
                 a[j] > a[piv] → j:= j-1
                 a[i] > a[piv] ∧ a[j] ≤ a[piv] → swap(a,i,j)
                                                 i:= i+1
                                                 j:= j-1
              fi
    od          {i = j+1, por eso todos los elementos en a[izq,j] son ≤ que a[piv]}
                                {∧ todos los elementos en a[i,der] son > que a[piv]}
    swap(a,piv,j)                   {dejando el pivote en una posición más central}
    piv:= j                                   {señalando la nueva posición del pivot}
end proc
{Post: a[1,izq) = A[1,izq) ∧ a(der,n] = A(der,n] ∧ a[izq,der] permutación de A[izq,der]
       ∧ izq ≤ piv ≤ der ∧ todos los elementos de a[izq,piv] son ≤ que a[piv]
       ∧ todos los elementos de a(piv,der] son > que a[piv]}
```

Este procedimiento elige un pivote (se elige arbitrariamente a[izq]) y lo utiliza para clasificar los elementos que se encuentran entre las posiciones izq y der: por un lado los que son menores o iguales al pivote, y por el otro los que son mayores a él. El procedimiento modifica el arreglo y la variable piv de forma que los primeros queden entre izq y piv y los segundos entre piv+1 y der. El índice i se utiliza para recorrer desde la posición izq+1 hacia la derecha, avanzando mientras encuentre elementos menores o iguales al pivote. El índice j se utiliza para recorrer desde la posición der hacia la izquierda, retrocediendo mientras encuentre elementos mayores al pivote. Cuando i no puede avanzar ni j retroceder es porque a[i]>a[piv] y a[j]≤a[piv]. En ese caso, se intercambian los contenidos de a[i] y a[j], tras lo cual i puede avanzar y j retroceder.

Cuando termina el ciclo, i es j+1, todos los anteriores a i son menores o iguales que el pivote y todos los posteriores a j son mayores que el pivote. Para finalizar, se ubica el pivote al final del primer fragmento y se asigna a piv esa nueva posición.

Para ordenar el fragmento del arreglo a que va de las posiciones izq a der, el algoritmo de ordenación primero realiza la clasificación que se acaba de explicar, y luego ordena recursivamente los dos segmentos: el anterior al pivote y el posterior al pivote.

```
{Pre: 0 ≤ der ≤ n ∧ 1 ≤ izq ≤ n+1 ∧ izq-1 ≤ der ∧ a = A}
proc quick_sort_rec (in/out a: array[1..n] of elem, in izq, der: nat)
    var piv: nat
    if der > izq → pivot(a,izq,der,piv)          {permuta los elementos de a[izq,der]}
                                                 {devuelve piv tal que izq ≤ piv ≤ der}
                                    {∧ todos los elementos en a[izq,piv] son ≤ que a[piv]}
                                    {∧ todos los elementos en a(piv,der] son > que a[piv]}
                    quick_sort_rec(a,izq,piv-1)
                    quick_sort_rec(a,piv+1,der)
    fi
end proc
{Post: a[1,izq) = A[1,izq) ∧ a(der,n] = A(der,n]
       ∧ a[izq,der] permutación ordenada de A[izq,der]}
```

El programa principal que dispara la ordenación rápida es simplemente

```
{Pre: n ≥ 0 ∧ a = A}
proc quick_sort (in/out a: array[1..n] of T)
    quick_sort_rec(a,1,n)
end proc
{Post: a está ordenado y es permutación de A}
```

El algoritmo de ordenación rápida es muy utilizado en la práctica ya que su comportamiento en el caso medio es eficiente. En efecto, asumiendo que piv siempre quede en el medio entre izq y der, su número de comparaciones estaría dado por la recurrencia

$$t(n) = 2t(n/2) + \Theta(n).$$

es decir, una recurrencia divide y vencerás con a=2, b=2 y k=1. Por ello $t(n) \in \Theta(n \log n)$. En la práctica, asumiendo arreglos con información aleatoria, obtenemos el mismo resultado. Esto se debe a que piv tiene una alta probabilidad de quedar cerca del medio entre izq y der. Por ello, si tomamos t(n) como el número de comparaciones que realiza la ordenación rápida en el caso medio obtenemos también $t(n) \in \Theta(n \log n)$.

De todas formas, no siempre es razonable asumir que la información de los arreglos es aleatoria. En numerosas aplicaciones, uno ordena un arreglo, realiza un número de modificaciones y luego vuelve a ordenarlo. La segunda vez que se ordena, el arreglo estará casi ordenado. En ese caso utilizar esta versión de ordenación rápida puede ser muy ineficiente.

En efecto, pensemos qué pasaría si aplicamos quick_sort a un arreglo perfectamente ordenado. La primera vez que se ejecute el procedimiento pivot, piv va a quedar en la posición 1 después de haber realizado n-1 comparaciones. Ordenar los anteriores a piv, evidentemente, será trivial. Ordenar los posteriores a piv, en cambio, determinará

un nuevo piv, esta vez en la posición 2, luego de haber realizado n-2 comparaciones. Iterando esto hasta terminar, vemos que el algoritmo realiza (n-1) + (n-2) + ...+ 1 comparaciones, es decir, es cuadrático. ¿Qué ocurrió? Ocurrió que piv quedó siempre en un extremo del segmento de arreglo a ordenar.

Podemos verlo también utilizando recurrencias. En el caso de aplicar quick_sort a un arreglo ordenado obtenemos

$$t(n) = t(0) + t(n-1) + n - 1 = t(n-1) + n - 1$$

ya que $t(0) = 0$ comparaciones. Esto nos da una recurrencia no homogénea con b=1 y d=1. El polinomio característico resultante es $(x-1)(x-1)^2$, o sea, $(x-1)^3$. Tenemos una sola raíz, 1, de multiplicidad 3. Por ello, obtendremos finalmente

$$t(n) = c_1 * 1^n + c_2 n 1^n + c_3 n^2 1^n = c_1 + c_2 n + c_3 n^2$$

para c_1, c_2 y c_3 que no nos molestamos en calcular acá. Es fácil comprobar que $c_3 \neq 0$ y por lo tanto, $t(n) \in \Theta(n^2)$.

Exactamente lo mismo ocurre al aplicar quick_sort a un arreglo ordenado al revés. Conclusión: quick_sort es un algoritmo que se comporta como $n \log n$ en el caso medio pero es cuadrático en el peor caso. Para confiar en el caso medio, es necesario comprobar que los datos a ordenar son aleatorios.

Sin embargo, existen modificaciones muy sencillas a la versión presentada acá que permiten confiar en el caso medio aún si se aplica a un arreglo ordenado. La más sencilla consiste en elegir (pseudo)aleatoriamente el pivote, en vez de tomar siempre el primero del segmento.