APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

DANIEL FRIDLENDER

Por favor, reportar errores, omisiones y sugerencias a fridlend@famaf.unc.edu.ar

Observaciones Preliminares

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 detalladamente las pre- y pos- condiciones que el programa debía satisfacer y se derivaba un programa que satisfaciera dicha especificación.

En **Algoritmos y Estructuras de Datos II**, sin descuidar las especificaciones de pre- y pos- condiciones ni la formulación de los invariantes de los ciclos, 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.

Ejemplos motivadores. Considere las siguientes preguntas:

1. Un pintor demora 1 hora y media en pintar una pared de 3m de largo. ¿Cuánto demorará en pintar una de 5m de largo?
2. Un pintor demora 1 hora y media en pintar una pared cuadrada de 3m de lado. ¿Cuánto demorará en pintar una de 5m de lado?
3. Si lleva 5 horas inflar un globo aerostático esférico de 4m de diámetro, ¿cuánto llevará inflar uno de 8m de diámetro?
4. Un bibliotecario demora 1 día en ordenar alfabéticamente una biblioteca con 1000 expedientes. ¿Cuánto demorará en ordenar una con 2000 expedientes?

La respuesta a las primeras 3 preguntas puede darse en forma precisa porque conocemos la relación entre el dato (longitud del lado o del diámetro) y la magnitud de la tarea. Además, se asume que se conoce un método simple para pintar una pared o inflar un globo. La respuesta a la cuarta pregunta, en cambio, no parece fácil de responder. No conocemos cuánto más trabajo es ordenar 2000 expedientes que 1000. Existen numerosos métodos de ordenación que usamos en la práctica sin preguntarnos realmente cuál es el mejor. La respuesta a la pregunta **depende** del algoritmo de ordenación que esté utilizando el bibliotecario. Supongamos que está utilizando el algoritmo de **ordenación por selección** (selection sort):

```
proc ssort (a: array[1..n] of T)                    {pre : n ≥ 0 ∧ a = A}
    var i, minp: int
    i:= 1                                  {inv : a[1, i) está ordenado ∧ a es permutación de A}
    do i < n → minp:= select(a,i)
               swap(a,i,minp)
               i:= i+1
    od
end proc                                   {pos : a está ordenado y es permutación de A}
```

En la primera ejecución del ciclo, este algoritmo utiliza la función select para encontrar la posición minp donde se encuentra el mínimo de todo el arreglo y luego ubica el mínimo encontrado en la primera posición del arreglo utilizando el procedimiento swap. En la segunda ejecución del ciclo, vuelve a utilizar la función select para encontrar la posición del segundo mínimo y luego ubica el segundo mínimo en la segunda posición del arreglo. En general, una vez ubicado el i-1 ésimo mínimo del arreglo en la posición i-1, el algoritmo utiliza select para encontrar la posición del i-ésimo mínimo del arreglo y luego el procedimiento swap para colocarlo en la posición i. Toda modificación del arreglo se realiza a través de swap que, como vemos a continuación, produce una permutación del arreglo dado. Esto garantiza que ningún valor de a se pierde ni duplica.

```
proc swap (a: array[1..n] of T, i,j: int)          {pre : a = A ∧ 1 ≤ i, j ≤ n}
    var tmp: int
    tmp:= a[i]
    a[i]:= a[j]
    a[j]:= tmp
end proc          {pos : a[i] = A[j] ∧ a[j] = A[i] ∧ ∀k.k ∉ {i, j} ⇒ a[k] = A[k]}
```

A continuación, detallamos el algoritmo utilizado para encontrar la posición del i-ésimo mínimo. Dado que el algoritmo ssort va ubicando los primeros i-1 mínimos en los primeros i-1 lugares del arreglo, para encontrar el i-ésimo mínimo de a basta con buscar el mínimo de a[i,n].

```
fun select (a: array[1..n] of T, i: int) ret minp: int          {pre : 0 < i ≤ n}
    var j: int
    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                                        {pos : a[minp] es el mínimo de a[i, n]}
```

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

¿Por qué dice i < n en vez de i ≤ n en la condición del **do** del procedimiento ssort?

El comando for. En el algoritmo presentado los 2 ciclos **do** se utilizan para recorrer un arreglo desde una posición predeterminada hasta otra también predeterminada. Además, el índice utilizado para recorrer el arreglo (i en un caso y j en el otro) sólo son

modificados al final del cuerpo 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:= a
do k $\leq$ b $\rightarrow$ c
k:= k+1
od

escribiremos

for k:= a **to** b **do** c **od**

Para que esta notación tenga sentido es fundamental que k no sea modificado en el cuerpo c del ciclo. Observar que esta notación hace más evidente que c se ejecuta una vez para cada valor de k desde a hasta b.

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

proc ssort (a: **array**[1..n] **of T**) $\{pre : n \geq 0 \wedge a = A\}$
var minp: **int**
for i:= 1 **to** n-1 **do** $\{inv : a[1, i)$ *está ordenado* $\wedge\ a$ *es permutación de* $A\}$
minp:= select(a,i)
swap(a,i,minp)
od
end proc $\{pos : a$ *está ordenado y es permutación de* $A\}$

Asimismo, la función select se puede escribir

fun select (a: **array**[1..n] **of T**, i: **int**) **ret** minp: **int** $\{pre : 0 < i \leq n\}$
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 $\{pos : a[minp]$ *es el mínimo de* $a[i, n]\}$

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. A continuación, una descripción intuitiva de cómo se cuentan 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 c1;c2;...;cn, sumamos las operaciones que se realizan durante la ejecución de cada comando de la secuencia:

$$\text{ops(c1;c2;...;cn)} = \text{ops(c1)} + \text{ops(c2)} + \ldots + \text{ops(cn)}$$

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

El ciclo **for** k:= a **to** b **do** c(k) **od** puede verse intuitivamente como una abreviatura de la secuencia de comandos c(a);c(a+1);...;c(b). 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:

$$\text{ops}(\textbf{for } \text{k:= a } \textbf{to } \text{b } \textbf{do } \text{c(k) } \textbf{od}) = \sum_{\text{k=a}}^{\text{b}} \text{ops(c(k))}$$

Esta fórmula no tiene en cuenta las operaciones necesarias para modificar k ni para compararlo con b. **Ejercicio:** Proponer una fórmula que sí las tenga en cuenta. La fórmula a aplicar en un caso concreto dependerá de qué operaciones uno desea contar.

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:

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

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 contar el número de veces que se ejecutará el cuerpo del ciclo. Una vez determinado este número, se suman las operaciones que se realizan en cada ejecución del cuerpo y en cada evaluación de la condición o guarda.

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. Observemos que sólo se realizan comparaciones entre elementos de a durante la ejecución de la función select. Para cada valor de i entre 1 y n esta función se ejecuta una vez. Cuando i=1, ésta realiza para cada j desde 2 hasta n la comparación a[j] < a[minp], totalizando n-1 comparaciones. Cuando i=2, las comparaciones que realiza la función select son n-2, y así sucesivamente. Esto da (n-1) + (n-2) + ... + 1 = $\frac{\text{n*(n-1)}}{2} = \frac{\text{n}^2}{2} - \frac{\text{n}}{2}$ comparaciones.

Utilizando las fórmulas propuestas anteriormente, enseguida llegamos al mismo resultado:

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

Estas expresiones indican 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 ssort y no durante la de la función select. Por cada valor de i se hace exactamente un intercambio (swap) entre a[i] y a[minp] al final del cuerpo del **for** de ssort. Como i toma n-1 valores, son n-1 intercambios.

Utilizando las fórmulas se obtiene lo mismo

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

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

Contestando la pregunta 4. Asumiendo que el bibliotecario utiliza el método de ordenación por selección, hace del orden de n^2 comparaciones. Le llevó un día hacer 1.000.000 de ellas, por lo que le llevará aproximadamente 4 días hacer 4.000.000 de ellas.

Existen sin embargo algoritmos de ordenación mejores que le permitirían ordenar 2000 expedientes en poco más del doble del tiempo que le lleva ordenar 1000.

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

proc isort (a: **array**[1..n] **of T**) $\{pre : n \geq 0 \wedge a = A\}$
  for i:= 1 **to** n-1 **do** $\{inv : a[1,i)$ *está ordenado* $\wedge\ a$ *es permutación de* $A \wedge\}$
    $\{\wedge$ *los elementos de* $a[1,i)$ *son menores o iguales a los de* $a[i,n]\}$
    insert(a,i)
  od
end proc $\{pos : a$ *está ordenado y es permutación de* $A\}$

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

proc insert (a: **array**[1..n] **of T**, i: **int**) **ret** $\{pre : 0 < i \leq n \wedge a = A\}$
  j:= i $\{inv : a[1\hat{}j\hat{}i]$ *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) **od**
end proc $\{pos : a[1,i]$ *está ordenado* $\wedge\ a$ *es permutación de* $A\}$

Número de comparaciones de la ordenación por inserción. Contamos comparaciones de la forma $a[j] < a[j-1]$, que sólo se realizan en el procedimiento insert. Intuitivamente, cuando i=1, no se realiza ninguna, 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 del input. Pero podemos afirmar que en el **mejor caso** serán $0 + 1 + 1 + \ldots + 1 =$ n-1 comparaciones (es decir, del orden de n comparaciones) y en el **peor caso**, $0 + 1 + 2 + \ldots +$ (n-1) $= \frac{n^2}{2} - \frac{n}{2}$ comparaciones (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.

Obtener el número de comparaciones en el mejor caso no tiene la misma importancia, sólo establece una **posibilidad** sobre el comportamiento del algoritmo.

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. El número de comparaciones de la ordenación por inserción en el caso promedio es del orden de n^2.

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.

operación elemental: se cuenta el número de operaciones elementales. Ésta debe:

- ser de duración constante, su duración no debe depender de n,
- ser representativa del comportamiento, 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 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.

La Notación $\mathcal{O}$

Sean $c(n)$ el número de comparaciones de la ordenación por selección para arreglos de tamaño n, $s(n)$ el número de swaps del mismo algoritmo para arreglos del mismo

tamaño. Por lo visto anteriormente, $c(n) = \frac{n^2}{2} - \frac{n}{2}$ y $s(n) = n - 1$. Observemos el crecimiento de estas funciones cuando n se duplica o triplica:

n	1.000	2.000	3.000
$c(n) = \frac{n^2}{2} - \frac{n}{2}$	499.500	1.999.000	4.498.500
n^2	1.000.000	4.000.000	9.000.000
$s(n) = n - 1$	999	1.999	2.999

¿Cuánto crece $c(n)$ cuando n se duplica o triplica? Puede observarse que los valores de $c(n)$ se cuadruplican o nonuplican. Lo mismo ocurre con n^2. Podemos decir que "a la larga, $c(n)$ crece al mismo ritmo que n^2" o que "a la larga, $c(n)$ es proporcional a n^2" o, como dijimos antes, que "$c(n)$ es del orden de n^2". De la misma manera, decimos que "a la larga, $s(n)$ crece al mismo ritmo que n" o que "a la larga, $s(n)$ es proporcional a n" o que "$s(n)$ es del orden de n".

Cuando lo que uno quiere expresar es el ritmo de crecimiento de una función $t(n)$ resulta conveniente utilizar funciones que tengan el mismo ritmo de crecimiento que $t(n)$ pero cuya expresión sea más simple. En el ejemplo anterior, si lo que se intenta comunicar es el ritmo de crecimiento, conviene decir que el número de comparaciones es proporcional a n^2 que decir que es exactamente igual a $\frac{n^2}{2} - \frac{n}{2}$. La primera expresión, más simple, me dice con la mayor sencillez posible que cuando n se duplica el número de comparaciones de la ordenación por selección se cuadruplica.

Pronto desarrollaremos una notación para expresar justamente que una cierta función crece a la larga al mismo ritmo que otra función. Antes de eso es conveniente introducir una notación para un concepto más sencillo de definir: "a la larga, $t(n)$ crece **a lo sumo** al ritmo de $f(n)$", que denotaremos $t(n) \in \mathcal{O}(f(n))$:

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

$$\mathcal{O}(f(n)) = \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists c \in \mathbb{R}^+ . \forall^\infty n \in \mathbb{N} . t(n) \leq c f(n)\}$$

de todas las funciones que a la larga crecen a lo sumo al ritmo de $f(n)$.

La notación $\forall^\infty n \in \mathbb{N} . \mathcal{P}(n)$ expresa que $\mathcal{P}$ se cumple para casi todo $n \in \mathbb{N}$, es decir, se cumple salvo a lo sumo en una cantidad finita de números naturales. Otra forma de expresar lo mismo es escribiendo $\exists n_0 \in \mathbb{N} . \forall n \in \mathbb{N} . n \geq n_0 \Rightarrow \mathcal{P}(n)$. Preferimos la notación utilizada en la definición por ser más compacta y evitar mencionar n_0.

Otra observación es que sólo importa el comportamiento de las funciones para n suficientemente grande, por lo que uno podría debilitar la condición $t, f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, requiriendo sólo que $\forall^\infty n \in \mathbb{N}$. $t(n), f(n) \in \mathbb{R}^{\geq 0}$, es decir, permitiendo que $t(n)$ y $f(n)$ devuelvan valores negativos para una cantidad finita de n's. Éste es el caso de la función $\log_a n$ más abajo.

Si $t(n) \in \mathcal{O}(f(n))$ decimos que $t(n)$ es (a lo sumo) **del orden de** $f(n)$.

En particular, para el caso de la ordenación por selección, el número de comparaciones $t(n) = \frac{n^2}{2} - \frac{n}{2}$ es del orden de n^2, dado que $t(n) \leq n^2$ para todo $n \in \mathbb{N}$.

Otro ejemplo interesante lo proporciona la función $t(n) = 5n^2 + 300n + 15$. A pesar de las constantes 5, 300 y 15, se puede comprobar fácilmente que $t(n)$ es del orden de n^2.

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 no nulas, $d_1 t(n)$ 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, $d_1 t(n) \leq d_1 cf(n) = (\frac{d_1 c}{d_2}) d_2 f(n)$.

Proposición: La relación "es a lo sumo del orden de" entre funciones de $\mathbb{N}$ en $\mathbb{R}^{\geq 0}$ 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 $f = g$.

Corolario: Para todo $a, b \in \mathbb{R}^+$ tales que $a, b \neq 1$, $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$.

Demostración: Para todo $n \in \mathbb{N}^+$, $\log_a n = \log_a b \ \log_b n$. Luego, $\log_a b$ es la constante que prueba que $\log_a n \in \mathcal{O}(\log_b n)$. Similarmente se prueba que $\log_b n \in \mathcal{O}(\log_a n)$.

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): $\forall f, g \in \mathbb{N} \to \mathbb{R}^{\geq 0}$, 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 $\delta \in \mathcal{R}^+$, se tiene que $\forall^\infty n \in \mathbb{N}. \frac{f(n)}{g(n)} - l < \delta$, luego $f(n) < (\delta + l) g(n)$.
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 para una constante $c \in \mathbb{R}^+, \forall^\infty n \in \mathbb{N}, g(n) \leq cf(n)$. Entonces tendríamos $\frac{f(n)}{g(n)} \geq \frac{1}{c}$, con lo cual $\lim_{n\to\infty} \frac{f(n)}{g(n)}$ 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. $d \geq 0 \Rightarrow \mathcal{O}(d^n) \subset \mathcal{O}(n!)$.

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

Demostración:

1. $\lim_{n\to\infty} \frac{n^x}{n^y} = \lim_{n\to\infty} \frac{1}{n^{y-x}} = 0$.
2. Como $\lim_{n\to\infty} \log n = +\infty = \lim_{n\to\infty} n^x$, por la Regla de L'Hôpital tenemos $\lim_{n\to\infty} \frac{\log n}{n^x} = \lim_{n\to\infty} \frac{\frac{1}{n}}{xn^{x-1}} = \lim_{n\to\infty} \frac{1}{xn^x} = 0$.
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 $n! \geq n(n-1)\ldots(n - \lfloor \frac{n}{2} \rfloor) \geq \lceil \frac{n}{2} \rceil^{\lceil \frac{n}{2} \rceil} \geq c^{2\lceil \frac{n}{2} \rceil} \geq c^n$. Por lo tanto, $c^n \in \mathcal{O}(n!)$ que implica $\mathcal{O}(c^n) \subseteq \mathcal{O}(n!)$. Por el inciso anterior, tenemos $\mathcal{O}(d^n) \subset \mathcal{O}(n!)$.

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: $\forall f, g \in \mathbb{N} \to \mathbb{R}^{\geq 0}$, si $\lim_{n\to\infty} \frac{g(n)}{f(n)} = 0$ entonces para todo $d \in \mathbb{R}$, $\mathcal{O}(f(n) + dg(n)) = \mathcal{O}(f(n))$.

Demostración: $\lim_{n\to\infty} \frac{f(n)+dg(n)}{f(n)} = \lim_{n\to\infty} (\frac{f(n)}{f(n)} + \frac{dg(n)}{f(n)}) = \lim_{n\to\infty} (1 + d\frac{g(n)}{f(n)}) = 1 + d \lim_{n\to\infty} \frac{g(n)}{f(n)} = 1 + d * 0 = 1$

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

Demostración: Usando la proposición anterior y que $\lim_{n\to\infty} \frac{a_{k-1}n^{k-1}+\ldots+a_1 n+a_0}{a_k n^k} = 0$.

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

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

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 $\mathcal{O}(t(n)) = \mathcal{O}(n^k)$ 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. Si $t(n)$ expresa la eficiencia del algoritmo más eficiente posible para resolver un determinado problema, también se habla de problema constante, logarítmico, lineal, cuadrático, polinomial, exponencial.

Notación Complementaria. La notación $\mathcal{O}$ está destinada especialmente a establecer cotas superiores a la eficiencia de un programa, es decir, para afirmar que una cierta función a la larga crece a lo sumo al ritmo de otra. También suele ser útil establecer cotas inferiores, para expresar que una cierta función a la larga crece **por lo menos** al ritmo de otra. Y como anunciamos en la página 7, también pretendemos definir una notación que exprese que una cierta función a la larga crece al mismo ritmo que otra, es decir, que una cierta cota es ajustada.

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

$$\Omega(f(n)) = \{t : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists d \in \mathbb{R}^+ . \forall^\infty n \in \mathbb{N} . t(n) \geq df(n)\}.$$

Así, $t(n) \in \Omega(f(n))$ dice que a la larga $t(n)$ crece como mínimo al ritmo de $f(n)$. Se puede comprobar inmediatamente que $g(n) \in \Omega(f(n))$ sii $f(n) \in \mathcal{O}(g(n))$.

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, se define el conjunto, $\Theta(f(n)) = \mathcal{O}(f(n)) \bigcap \Omega(f(n))$.

Así, $t(n) \in \Theta(f(n))$ dice que a la larga $t(n)$ crece al mismo ritmo que $f(n)$.

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 $\Omega(f(n)) = \Omega(g(n))$

Operaciones sobre la notación $\mathcal{O}$. Extendemos operadores entre funciones (como la suma, producto, etc.) como operadores entre conjuntos de funciones. Si X e Y son conjuntos de funciones de $\mathbb{N} \to \mathbb{R}^{\geq 0}$ y $\odot$ un operador entre funciones, se define

$$X \odot Y = \{h : \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists f \in X, g \in Y, \forall n \in \mathbb{N} . h(n) = f(n) \odot g(n)\}$$

Las siguientes operaciones, son las que permiten utilizar la notación $\mathcal{O}$ cuando se calcula la complejidad de un programa. La suma corresponde a secuencia de fragmentos de programas. La multiplicación corresponde a la iteración.

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

Demostración: $\subseteq$) Sea $h(n) \in \mathcal{O}(f(n)) + \mathcal{O}(g(n))$, es decir, $h(n) = f'(n) + g'(n)$ con $f'(n) \leq c_1 f(n)$ y $g'(n) \leq c_2 g(n)$ para casi todo $n \in \mathbb{N}$. Eligiendo $c \geq c_1, c_2$, queda $h(n) \leq c(f(n) + g(n))$, es decir, $h(n) \in \mathcal{O}(f(n) + g(n))$, o sea, $\mathcal{O}(f(n)) + \mathcal{O}(g(n)) \subseteq \mathcal{O}(f(n) + g(n))$.

$\supseteq$) Sean f' y g' tales que $f'(n) = f(n)$ y $g'(n) = 0$ si $f(n) \geq g(n)$, en caso contrario, $f'(n) = 0$ y $g'(n) = g(n)$. Claramente $f'(n) \in \mathcal{O}(f(n))$ y $g'(n) \in \mathcal{O}(g(n))$. Entonces $\text{máx}(f(n), g(n)) = f'(n) + g'(n) \in \mathcal{O}(f(n)) + \mathcal{O}(g(n))$. Es decir, $\mathcal{O}(f(n) + g(n)) = \mathcal{O}(\text{máx}(f(n), g(n))) \subseteq \mathcal{O}(f(n)) + \mathcal{O}(g(n))$.

Proposición: $\mathcal{O}(f(n)) * \mathcal{O}(g(n)) = \mathcal{O}(f(n) * g(n))$.

Demostración: $\subseteq$) Sea $h(n) \in \mathcal{O}(f(n)) * \mathcal{O}(g(n))$, es decir, $h(n) = h_1(n) * h_2(n)$ con $h_1(n) \in \mathcal{O}(f(n))$ y $h_2(n) \in \mathcal{O}(g(n))$. Entonces $h(n) = h_1(n) * h_2(n) \leq cf(n) * dg(n) = cd * f(n) * g(n)$ para casi todo $n \in \mathbb{N}$. Es decir, $\mathcal{O}(f(n)) * \mathcal{O}(g(n)) \subseteq \mathcal{O}(f(n) * g(n))$.

$\supseteq$) Sea $h(n) \in \mathcal{O}(f(n) * g(n))$, y c tal que $h(n) \leq c * f(n) * g(n)$ para casi todo $n \in \mathbb{N}$. Sea $f'(n) = 0$ si $g(n) = 0$ y $f'(n) = h(n)/g(n)$ en caso contrario. Como $g(n) = 0$ implica $h(n) = 0$, $h(n) = f'(n) * g(n)$ para todo $n \in \mathbb{N}$. Además, $f'(n) = 0 \leq cf(n)$ o $f'(n) = h(n)/g(n) \leq cf(n)$, es decir, $f'(n) \in \mathcal{O}(f(n))$. Tenemos entonces que $h(n) = f'(n) * g(n) \in \mathcal{O}(f(n)) * \mathcal{O}(g(n))$.

Más notación. También se escribe $f(n)+\mathcal{O}(g(n))$ por $\{f(n)\}+\mathcal{O}(g(n))$, etc. Ejemplos: $4n^2 + \mathcal{O}(n)$, $2^n\mathcal{O}(n^2)$, $n^{\mathcal{O}(1)}$, etc.

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.

fun lsearch (a: **array**[1..n] **of T**, x:**T**) **ret** i:**int** $\{pre : n \geq 0\}$
  i:= 1 $\{inv : x \text{ no está en } a[1, i)\}$
  do 1 $\geq$ n $\wedge$ a[i] $\neq$ x $\rightarrow$ i:= i+1 **od**
end fun $\{pos : x \text{ está en } a \text{ sii } i \leq n \wedge x = a[i]\}$

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: $t_3(n) = \frac{n(n+1)}{2n} = \frac{n+1}{2} \in \Theta(n)$.

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 ese 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 si la palabra que uno busca está en la parte izquierda uno limita la búsqueda en esa parte (abriendo nuevamente al medio, etc), si en cambio está en la parte derecha, uno limita la búsqueda a esa parte, etc.

fun bsearch (a: **array**[1..n] **of T**, x:T) **ret** i:**int** $\{pre : n \geq 0\}$
  var izq,med,der: **int**
  izq:= 0
  der:= n+1 $\{inv : x \text{ está en } a \text{ sii } x \text{ está en } a(izq, der)\}$
  do izq + 2 < der $\rightarrow$
    med:= (izq+der) $\div$ 2

```
        if x < a[med] → der:= med
           x = a[med] → izq:= med-1
                        der:= med+1
           x > a[med] → izq:= med
        fi
    od
    i := izq + 1
end fun                                   {pos : x está en a sii i ≤ n ∧ 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 izqi y deri denotan los valores de izq y der después de la i-ésima ejecución del ciclo, sabemos que di = deri-izqi-1 son la cantidad de celdas de a que nos quedan por explorar para encontrar x. Se puede ver que si di > 1, entonces d(i+1) $\leq$ di/2. Como d0 = n y el ciclo termina cuando di = 1, esto pasará -en el peor caso- cuando i = $\lceil \lg n \rceil$. Por lo tanto, $t(n) \in \mathcal{O}(\log n)$, donde $t(n)$ es el número de comparaciones que se realizan en el peor caso.

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(i+1) $\geq$ di/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.

1. Recurrencias

Recordemos el problema formulado en una clase anterior:

> **Pregunta 2:** 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 de 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 = 1/4 día de trabajo), ordenar los otros 500 expedientes (tarea AB $\approx$ 250.000 comparaciones = 1/4 día de trabajo), 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. Esta idea puede iterarse: 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:

```
mergeSort :: [T] → [T]
mergeSort [] = []
mergeSort [t] = [t]
mergeSort ts = merge sts1 sts2
               where
                    sts1 = mergeSort ts1
                    sts2 = mergeSort 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 ($\pm$ 1) y merge realiza la intercalación. En estilo imperativo la principal dificultad es la necesidad de utilizar arreglos auxiliares para realizar la intercalación. Se lo puede escribir de la siguiente manera:

```
var a: array[1..n] of T
var b,c: array[1..n÷2+1] of T
proc mergeSort (izq,der: int)                    {pre : n ≥ der ≥ izq > 0 ∧ a = A}
     var med: int
     if der > izq → med:= (der+izq) ÷ 2
                    mergeSort(izq,med)
                                  {a[izq, med] permutación ordenada de A[izq, med]}
                    mergeSort(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[izq] 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(izq,med,der)

$\{a[izq, der]$ *permutación ordenada de* $A[izq, der]\}$

fi

end proc $\{a$ *permutación de* $A \wedge a[izq, der]$ *permutación ordenada de* $A[izq, der]\}$

Asimismo, el procedimiento intercalar puede escribirse de la siguiente manera.

proc intercalar (izq,med,der: **int**)

$\{pre : n \geq der > med \geq izq > 0 \wedge a = A \wedge b = B \wedge c = C\}$

$\{pre : B[1, med - izq + 1] = A[izq, med] \wedge C[1, der - med] = A[med + 1, der]\}$

$\{pre : B[1, med - izq + 1]$ *y* $C[1, der - med]$ *ordenados*$\}$

var i,j,maxi,maxj: **int**

i:= 1

j:= 1

maxi:= med-izq+1

maxj:= der-med

$\{inv : b = B \wedge c = C \wedge a[1, izq) = A[1, izq) \wedge a[k, n] = A[k, n]\}$

$\{inv : a[izq, k) =$ *intercalación de* $b[1, i)$ *con* $c[1, j)\}$

for k:= izq **to** der **do**

if i $\leq$ maxi $\wedge$ (j $>$ maxj $\vee$ b[i] $\leq$ 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* $A \wedge a[izq, der]$ *permutación ordenada de* $A[izq, der]\}$

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

mergeSort(1,n)

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". Justamente el hecho de que los problemas menores sean de igual naturaleza que el original, es lo que permite que el mismo algoritmo puede 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 mergeSort 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 primer llamada recursiva a mergeSort, esto es, $t(\lceil n/2 \rceil)$, más todas las comparaciones que hace la segunda llamada recursiva a mergeSort, 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 $m = \lg n$. Reemplazando queda $t(n) \leq n \lg n$, para todo n que sea potencia de 2. Concluimos que $t(n) \in \mathcal{O}(n \lg n | n$ potencia de 2).

Análogamente, usando la segunda desigualdad obtenemos $t(n) \geq \frac{1}{2} n \lg n$ y concluimos que $t(n) \in \Omega(n \lg n | n$ potencia de 2) y de ambas conclusiones, que $t(n) \in \Theta(n \lg n | n$ 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, $t(n+1) = t(\lceil (n+1)/2 \rceil) + t(\lfloor (n+1)/2 \rfloor) + n + 1 - 1 > t(\lceil n/2 \rceil) + t(\lfloor n/2 \rfloor) + n = t(n)$.

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

Al analizar el algoritmo mergeSort utilizamos intuitivamente la siguiente notación.

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

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

El razonamiento hecho para el análisis del mergeSort puede generalizarse:

Definición: Dada $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$, 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 mergeSort, $t(n)$ es eventualmente no decreciente. La función $n \lg n$ es 2-suave ya que para todo $n \geq 2$ se cumple $2n \lg(2n) = 2n(\lg 2 + \lg n) = 2n + 2n \lg n \leq 2n \lg n + 2n \lg n = 4n \lg n$. Del resultado que sigue se desprende que $n \lg n$ 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,

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

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

Los siguiente 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 $f : \mathbb{N} \to \mathbb{R}^{\geq 0}$ suave y $t : \mathbb{N} \to R^{\geq 0}$ 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

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

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

Ejemplo: sea $t(n)$ el número de comparaciones que realiza el mergeSort: $t(n)$ es eventualmente no decreciente, $t(n) \in \Theta(n \log n | n \text{potencia de} 2)$ y $n \log n$ es suave. Luego $t(n) \in \Theta(n \log n)$.

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 mergeSort. Este tema se dicta siguiendo la presentación de los libros "Fundamentos de Algoritmia" de Brassard y Bratley, páginas 135 a 148 e "Introduction to Algorithms: A Creative Approach" de Manber, páginas 50 a 55. Sólo se reproduce acá una breve descripción del método.

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 merge sort en el peor caso, o el número de comparaciones que realiza la búsqueda binaria en el peor caso. El primer caso 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 $t(n_0) = t_{n_0}, \ldots, t(n_0 + k) = t_{n_0+k}$, donde este último valor, 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 (n: int)                                          {pre : n ≥ 0}
    if i = 0 → skip
       i = 1 → A
       i > 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 $t_0 = 0$, $t_1 = 1$, $t_2 = 1$ coincidiendo con el resultado obtenido para calcular t_0, t_1, 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 $t_0 = 0$, $t_1 = 1$, $t_2 = 2$, $t_3 = 2$ coincidiendo con el resultado obtenido para calcular t_0, t_1, t_2, t_3 usando la recurrencia original.

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 despejando sucesivamente de 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 $t(n_0) = t_{n_0}, \ldots, t(n_0 + k + d + 1) = t_{n_0+k+d+1}$, donde este último valor, $t_{n_0+k+d+1}$, puede obtenerse utilizando la ecuación característica.

Ejemplo: calcular explícitamente t_n donde

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

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 $t_2 = 2t_1 + 2 = 4$. Sistema de ecuaciones:

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

6. despejando, $c_1 = 2$, $c_2 = -2$ y $c_3 = -1$.
7. solución final $t_n = 2 * 2^n - 2 - n$, simplificando $t_n = 2^{n+1} - n - 2$.
8. efectivamente, con la solución final $t_0 = 0$, $t_1 = 1$, $t_2 = 4$, $t_3 = 11$ coincidiendo con el resultado obtenido para calcular t_0, t_1, t_2, t_3 usando la recurrencia original.

Estructuras de Datos

La elección de las **estructuras de datos** o **tipos de datos** son determinantes a la hora de diseñar un programa. Una elección correcta dará lugar a programas elegantes, breves, eficientes, legibles, fáciles de mantener, reusables. Una elección equivocada o simplemente apurada puede tener como consecuencia programas sin ninguna de las virtudes mencionadas, incorrectos (que no resuelven el problema) y cuyas fallas, además, son difíciles de encontrar, no hablemos de reparar.

Un lenguaje de programación normalmente proporciona ciertos tipos básicos, e incluso maneras de definir ciertos tipos más complejos. Podemos llamar a éstos **tipos concretos**. Distintos lenguajes proporcionan distintos tipos concretos, es un concepto relativo al lenguaje de programación que se utilice.

Los **tipos abstractos** no están asociados al lenguaje de programación que uno utilice sino más bien al problema que uno intenta resolver. Los tipos abstractos surgen cuando uno analiza el problema, e identifica qué información debe manipular el programa, y cómo debe manipularla. Se denominan tipos abstractos para resaltar el hecho de que uno está pensando en las propiedades del tipo de dato, y no en su implementación. Lo que uno elabora al pensar en tipos abstractos es independiente de toda posible implementación del mismo.

Por ejemplo, si debemos escribir un programa *rutaArg* que informe cuál es el camino más corto para trasladarse en auto de una ciudad a otra del país, surgirá naturalmente pensar que la información que dicho programa maneja es un grafo, donde los nodos son ciudades (y cruces de rutas) y las aristas son (segmentos de) rutas con sus longitudes. Grafo será entonces un tipo abstracto que se utilizará para resolver el problema, a pesar de que usualmente los lenguajes de programación no cuentan con un tipo concreto Grafo.

Finalmente, cuando el problema tenga que ser programado para su ejecución por una computadora, los tipos abstractos deberán ser implementados en el lenguaje de programación utilizando tipos concretos. Ésta será la **implementación** o **representación** del tipo abstracto o el tipo concreto correspondiente al tipo abstracto. Normalmente, un mismo tipo abstracto puede ser implementado de numerosas maneras diferentes.

Sin embargo, identificar y especificar los tipos abstractos involucrados es muy importante porque proporciona un entendimiento cabal del problema que se intenta resolver (en vez de abocarse directamente a intentar resolver un problema no entendido completamente); porque permite escribir los algoritmos en un lenguaje comprensible por el ser humano; permite poner a prueba tempranamente si la solución que se está diseñando

está bien encaminada; facilita el desarrollo del programa en equipo; y proporciona una solución **independiente de la representación** del tipo abstracto. Esto último quiere decir que uno obtiene una solución que admite diferentes maneras de implementar el tipo abstracto; en particular, es provechoso para facilitar el reemplazo de una implementación del tipo abstracto por otra en caso de que uno lo considere luego conveniente.

En esta parte de la materia, desarrollaremos la técnica de la especificación de tipos abstractos de datos e ilustraremos sobre posibles implementaciones.

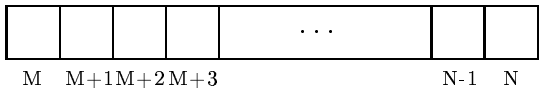
Primeramente haremos un breve repaso de los tipos concretos usualmente proporcionados por los lenguajes de programación imperativos. Además de los tipos básicos enteros, caracteres, booleanos, strings, flotantes, etc. los lenguajes imperativos normalmente proporcionan maneras de definir tipos compuestos. Describiremos arreglos, listas, tuplas y punteros.

Luego estudiaremos los tipos abstractos de datos (TADs) pila, cola, cola de prioridades y las estructuras árbol binario, árbol binario de búsqueda, heap.

Arreglos. Dado un tipo **T**, normalmente declaramos arreglos de la siguiente forma:

type tarreglo = **array**[M..N] **of T**
var a: tarreglo

El tipo tarreglo así definido corresponde al producto Cartesiano $\mathbf{T}^{\text{N-M+1}}$. Los arreglos se alojan normalmente en **espacios contiguos** de memoria. El arreglo a declarado recientemente suele representarse gráficamente de la siguiente manera:



En la representación gráfica se observa una celda para cada índice entre M y N. El valor alojado en la celda i se obtiene evaluando la expresión a[i] y se modifica asignando a a[i] (ejemplo, a[i]:= e). Al estar alojado en espacio contiguo de memoria acceder o modificar cualquier celda lleva **tiempo constante**[1] $\Theta(1)$ (una cuenta sencilla permite hallar la posición de memoria donde se aloja a[i]).

Los arreglos tienen longitud prefijada: N-M+1. Normalmente N > M, pero se puede admitir también N=M (longitud 1) e incluso N < M (longitud 0). El tamaño total del arreglo (espacio ocupado en memoria) es la longitud del mismo multiplicada por el tamaño de cada celda, que depende del tipo **T**. Es decir, ocupa espacio $\Theta(n)$ donde n es su longitud.

A veces conviene utilizar índices que no sean simplemente enteros. Por ejemplo:

type ttabla = **array**['a'..'z'] **of int**
var d: ttabla

Dado un diccionario D, el arreglo d podría servir para informar en qué página del diccionario D comienza cada letra, por ejemplo, d['g'] = 271 significaría que la letra g comienza en la página 271 del diccionario D.

Otras posibilidades son

type tdias = (domingo, lunes, martes, miercoles, jueves, viernes, sabado)

[1] El tiempo no depende del número de celdas sino del tamaño de las celdas.

type tagenda = **array** [lunes..viernes] **of T**
var ag: tagenda

En este caso, la variable ag es un arreglo con cinco celdas, una para cada día hábil de la semana. Por ejemplo, ag podría almacenar las tareas a desarrollar cada uno de esos días.

Esto muestra que se puede utilizar una variedad de conjuntos como índices de arreglos. Lo importante es que haya una clara noción de "el siguiente de", cosa que ocurre con enteros (el siguiente de 4 es 5), caracteres (el siguiente de 'h' es 'i') y tipos enumerados como tdias (el siguiente de miercoles es jueves).

También es frecuente la utilización de arreglos multidimensionales, ejemplos:

type tarreglo1 = **array**[1..N,1..M] **of T**
type tarreglo2 = **array**[1..N,'a'..'z',domingo..sabado] **of T**
var b: tarreglo2

dando lugar a celdas que se acceden con notaciones tales como b[3,'k',martes]. Para inicializar y modificar arreglos es muy común utilizar el comando **for**, por ejemplo

for i:= M **to** N **do** a[i]:= 0 **od**

este comando adquiere en general la forma

for i:= M **to** N **do** c **od**
for l:= 'a' **to** 'z' **do** c **od**
for d:= martes **to** viernes **do** c **od**

donde c (llamado **el cuerpo** del **for**) es cualquier comando que **no modifica** el valor de la variable que se usa como índice[2] (i, l y d respectivamente en estos ejemplos). Se puede traducir directamente usando **while**. El primer ejemplo se traduce a:

var i: **int**
i:= M
while i $\leq$ N **do**
    c
    i:=i+1
od

Es decir que cuando M>N, el comando c no se ejecuta. Una variante del **for** es usando **downto** en vez de **to**:

for i:= M **downto** N **do** c **od**

que se traduce a

var i: **int**
i:= M
while i $\geq$ N **do**
    c
    i:=i-1
od

[2]Lamentablemente, la mayoría de los lenguajes de programación permiten que dicha variable sea modificada en el cuerpo del **for**. Se considera una **pésima práctica** de programación escribir un **for** en el que eso ocurre.

Ahora el comando c no se ejecuta cuando M<N.

En el ejemplo del arreglo tridimensional b declarado más arriba, si se quiere inicializar todo el arreglo (asumiendo que **T** es, por ejemplo, **int**), se lo puede hacer a través de 3 lazos **for** anidados:

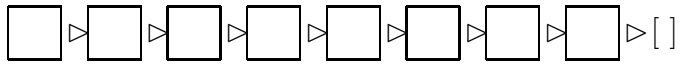
```
for i:= M to N do
    for l:= 'a' to 'z' do
        for d:= martes to viernes do
            b[i,k,d]:= 0
        od
    od
od
```

Por último, decimos que un invariante de **for** i:= M **to** N **do** c **od** es un predicado $\mathcal{I}(i)$, tal que $\mathcal{I}$(M) vale antes de la ejecución del **for** y la validez de $\mathcal{I}(i) \wedge M \leq i \leq N$ antes de cada ejecución de c garantiza la validez $\mathcal{I}(i+1)$ después de dicha ejecución de c. Entonces, si N $\geq$ M-1, al finalizar el **for** se cumple $\mathcal{I}$(N+1).

Listas. Algunos lenguajes de programación permiten declarar listas:

type tlista = **list of T**
var l: tlista

El tipo tlista así definido corresponde a la unión de los productos Cartesianos $\mathbf{T}^i$, es decir, corresponde a $\bigcup_{i=0}^{\infty} \mathbf{T}^i$. Así, a diferencia del arreglo cuya longitud está predeterminada, el número de elementos de una lista no lo está. A priori, puede contener una cantidad arbitraria de elementos de **T** (en la práctica evidentemente existirán limitaciones de espacio). Otra diferencia importante con el arreglo es que la lista **no** se aloja en **espacios contiguos** de memoria. La lista l declarada recientemente puede representarse gráficamente de la siguiente manera:



En la representación gráfica se ve una celda para cada elemento de la lista. Una lista puede modificarse agregando (ejemplos, l1:= e ▷ l2 o también l:= e ▷ l) o quitando un elemento (ejemplos, si l es no vacía, l1:= tl(l) o también l:= tl(l)). Son justamente estas operaciones las que impiden alojar una lista en espacios contiguos de memoria. Al agregarse un elemento a una lista, no hay ninguna garantía de que haya espacio libre justo en la posición de memoria anterior a donde se encuentra el primer elemento de la lista. Si se quisiera alojar la nueva lista en espacios contiguos habría que copiar la lista entera en una parte de la memoria donde haya suficiente espacio libre para toda la lista. En principio, esto es posible (aunque presenta numerosos inconvenientes), pero no es lo que normalmente se hace ya que significa que las modificaciones serían lineales, o sea, del orden de $\Theta(n)$ donde n es la longitud de la lista. En vez de esto, se aloja el elemento que se quiere agregar en un nuevo lugar de memoria (cualquiera que esté libre) y se mantiene la información que dice en qué lugar de la memoria se encuentran los siguientes elementos de la lista. Así, estas modificaciones resultan constantes, o sea, del orden de $\Theta(1)$. Luego de una secuencia de modificaciones, los elementos de una lista

pueden quedar desperdigados en la memoria. Siempre se puede recorrer la lista ya que se cuenta con la información necesaria para ir de cada elemento de la lista al siguiente.

Esto significa que para acceder al i-ésimo elemento de una lista es necesario recorrerla secuencialmente hasta encontrarlo, operación que resulta lineal $\Theta(n)$ en el peor caso y en el caso promedio.

Numerosos lenguajes de programación no tienen el tipo concreto lista. Los más importantes ofrecen sin embargo alguna forma de implementarlo. Veremos luego que se pueden implementar listas usando los tipos concretos tupla y puntero.

Por último, puede convenir a veces extender la notación del **for**. Por ejemplo, si se quiere ejecutar c una vez para cada elemento e de la lista l (del primero al último) se escribe:

for e ∈ l **do** c **od**

La misma notación puede utilizarse al recorrer arreglos si el cuerpo del **for** no necesita referirse a las posiciones. Por ejemplo,

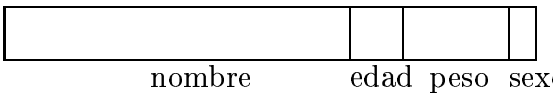
for e ∈ a **do** c **od**

ejecuta el cuerpo c para todos los elementos alojados en celdas de a, independientemente de la dimensión del arreglo a.

Tuplas. También llamados registros o estructuras, se utilizan para representar productos Cartesianos, pero ahora cuando los conjuntos entre los que se hace el producto son distintos, es decir, de la forma $\mathbf{T}_1 \times \mathbf{T}_2 \times \mathbf{T}_3$ donde los $\mathbf{T}_i$ pueden ser tipos distintos. Se declaran de la siguiente forma

```
type ttupla = tuple
                nombre: string
                edad: int
                peso: real
                sexo: (masculino, femenino)
              end
var t: ttupla
```

El tipo ttupla así definido corresponde a $\mathbf{string} \times \mathbf{int} \times \mathbf{real} \times \{\text{masculino,femenino}\}$, y nombre, edad, peso y sexo se llaman **campos** (o **fields**). Las tuplas se alojan normalmente en **espacios contiguos** de memoria. Se puede representar gráficamente de la siguiente manera:



En la misma se ven los campos de distinto tamaño porque cada uno de ellos puede ocupar un espacio diferente. Lo alojado en cada campo se obtiene evaluando las expresiones t.nombre, t.edad, t.peso y t.sexo, y se modifica asignando (ejemplo, t.nombre:= "Martina"). Al estar alojado en espacio contiguo de memoria acceder o modificar cualquier campo lleva **tiempo constante**[3] $\Theta(1)$ (una cuenta sencilla permite hallar la

[3]El tiempo depende del tamaño del campo.

posición de memoria donde se aloja cada campo). Claramente el tamaño de la tupla es la suma de los tamaños de sus campos.

Punteros. Dado un tipo T, se puede declarar el tipo "puntero a T". Por ejemplo, si ttupla es el tipo definido más arriba

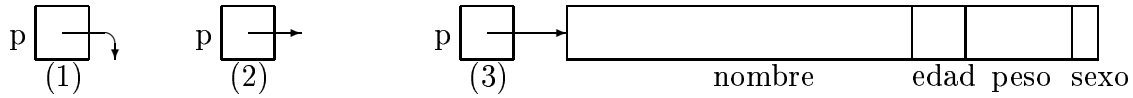
type tptupla = **pointer to** ttupla
var p: tptupla

La variable p así declarada es un puntero a ttupla. Esto significa que p puede almacenar una dirección de memoria donde se aloja una ttupla. Las operaciones con punteros son las siguientes:

p:= e
alloc(p)
free(p)

El primer comando es una asignación: e es una expresión cuyo valor es la dirección de memoria de una ttupla, la asignación tiene por efecto que dicha dirección sea alojada ahora en p. El segundo comando reserva un nuevo espacio de memoria capaz de almacenar una ttupla, y la dirección de ese nuevo espacio de memoria se aloja en p. El tercer comando libera el espacio de memoria señalado por p, es decir, cuya dirección se encuentra alojada en p. Puede darse que p tenga como valor una dirección de memoria que no está actualmente reservada (por ejemplo, inmediatamente después de haber ejecutado free(p)). Para evitar permanecer en este estado, existe un valor especial que puede adoptar un puntero, llamado **null**. Cuando el valor de p es **null**, p no señala ninguna posición de memoria.

Hay distintas representaciones gráficas, una para cada una de las posibles situaciones:

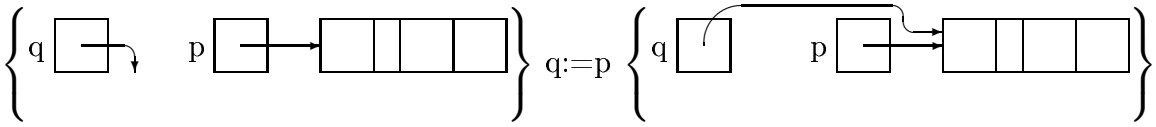


En la situación (1), el valor de p es **null**, p no señala ninguna posición de memoria. En la situación (2) la posición de memoria señalada por p no está reservada, por ejemplo inmediatamente después de free(p). En la situación (3) el valor de p es la dirección de memoria donde se aloja la ttupla representada gráficamente al final de la flecha, por ejemplo, inmediatamente después de alloc(p). En la situación (3), $\star$p denota la ttupla que se encuentra señalada por p, y por lo tanto, $\star$p.nombre, $\star$p.edad, $\star$p.peso y $\star$p.sexo, sus campos. Esta notación permite acceder a la información alojada en la ttupla y modificarla mediante asignaciones a sus campos. Siempre en la situación (3), si r es una ttupla, también se puede hacer $\star$p:= r alojando r en la posición señalada por p (pero perdiendo lo que había anteriormente en esa posición de memoria). Esta última asignación no es válida en algunos lenguajes (por ejemplo, no es válida en C). Una notación conveniente para acceder a los campos de una tupla señalada por un puntero es la flecha "->". Así, en vez de escribir $\star$p.nombre, podemos escribir p->nombre tanto para leer ese campo como para modificarlo. Esta notación reemplaza el uso de dos operadores ("$\star$" y ".") por uno ("->") visualmente más apropiado.

En la situación (2) el valor de p es inconsistente, no debe utilizarse ni accederse una dirección de memoria no reservada ya que no se sabe, a priori, qué hay en ella

(en particular puede haber sido reservada para otro uso y al modificarlo se estaría corrompiendo información importante para tal uso).

Como vemos, los punteros permiten manejar explícitamente direcciones de memoria. Esto no es sencillo, aparecen situaciones que con los tipos de datos usuales no se daban. Por ejemplo:



Como se ve, después de la asignación q y p señalan a la misma tupla, por lo que cualquier modificación en campos de $\star$q también modifican los de $\star$p (claro, ya que son los mismos) y viceversa. Estamos en presencia de lo que se llama **aliasing**, es decir, hay 2 nombres distintos para el mismo objeto y al modificar uno se modifica el otro. Programar correctamente en presencia de aliasing es muy delicado y requiere gran atención.

Siempre hemos asumido que no es necesario ocuparse de reservar y liberar espacios de memoria para las variables. Los punteros como p y q son variables, así que tampoco es necesario reservar y liberar espacio para ellos. Pero las operaciones alloc y free permiten reservar y liberar explícitamente espacio para los objetos que p y q señalan.

Esta posibilidad significa ciertas libertades: el programador puede decidir exactamente cuándo reservar espacio para una tupla. Por otro lado, significa también más responsabilidad: el programador es el que debe encargarse de liberar el espacio cuando deje de ser necesario.

Pero el verdadero beneficio de los punteros radica en que permiten una gran flexibilidad para representar estructuras complejas, y por lo tanto, para implementar diferentes tipos abstractos de datos.

Tipos Abstractos de Datos (TADs)

Recordemos que cuando hablamos de tipos abstractos de datos, lo esencial es que estamos pensando en caracterizar un tipo o conjunto de datos por sus propiedades y no por la manera en que puedan implementarse. Son descripciones independientes de cualquier posible implementación, de ahi el vocablo **abstracto**.

TAD lista. Si bien muchos lenguajes de programación carecen del tipo concreto lista mencionado anteriormente, las listas son necesarias en una multitud de aplicaciones. Por ello, naturalmente aparecen en la tarea del programador. Presentamos a continuación brevemente una descripción de **lista** como tipo abstracto. Tratándose de un tipo familiar, éste servirá como ejemplo para ilustrar la notación que utilizaremos para especificar/definir tipos abstractos de datos.

TAD list[elem]
constructores
[ ] : list
$\rhd$: elem $\times$ list $\rightarrow$ list
operaciones
$\lhd$: list $\times$ elem $\rightarrow$ list

hd : list → elem {se aplica sólo a una lista no vacía}
tl : list → list {se aplica sólo a una lista no vacía}
length: list → nat
is_empty : list → bool
. : list × nat → elem {se aplica sólo a una lista con suficientes elementos}

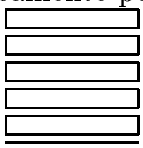
ecuaciones

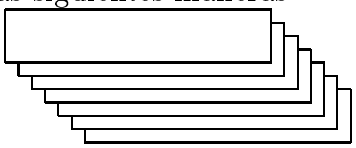
[ ]⊲ f = f ⊳[ ]
(e ⊳ l) ⊲ f = e ⊳ (l ⊲ f)
hd(e ⊳ l) = e
tl(e ⊳ l) = l
length([ ]) = 0
length(e ⊳ l) = 1 + length(l)
is_empty([ ]) = true
is_empty(e ⊳ l) = false
(e ⊳ l).1 = e
1 ≤ n ≤ length(l) ⇒ (e ⊳ l).(n+1) = l.n

Más adelante presentaremos **listas enlazadas**, la manera más utilizada de implementar listas en lenguajes en los que las mismas no existen como tipo concreto.

TAD pila. El tipo abstracto **pila** (o **stack**) se puede explicar sencillamente pensando en una pila de platos, por ejemplo que deben ser lavados. Cuando no hay platos para lavar tenemos una pila **vacía**, cuando se ensucia un plato se lo **pone arriba** de la pila, uno sólo puede **ver** el plato que está **más alto** en la pila y cuando uno va a lavar un plato **saca** de la pila el que está **más alto** (y lo lava). Uno siempre puede ver si hay platos para lavar, controlando si la pila **está vacía o no**.

Gráficamente puede representarse de las siguientes maneras





El gráfico de la derecha resalta el hecho de que sólo puede observarse lo que se encuentra en el primer lugar de la pila. El de la izquierda permite graficar lo que se encuentra en los otros lugares también, simplemente hay que tener presente que lo único que uno puede verdaderamente observar es lo que se encuentra en el lugar más alto.

Hemos resaltado las 5 propiedades que definen el TAD pila: la existencia de una pila vacía (**empty**), la posibilidad de apilar (**push**), la de ver el último que se agregó (**top**) si la pila no está vacía, la de desapilar el último que se agregó (**pop**) si la pila no está vacía, la de controlar si la pila está vacía (**is_empty**). La característica de la pila es que se extrae el último que se introdujo (LIFO = Last In, First Out).

Utilizando la notación introducida para especificar tipos abstactos de datos:

TAD stack[elem]

constructores

empty : stack
push : elem × stack → stack

operaciones

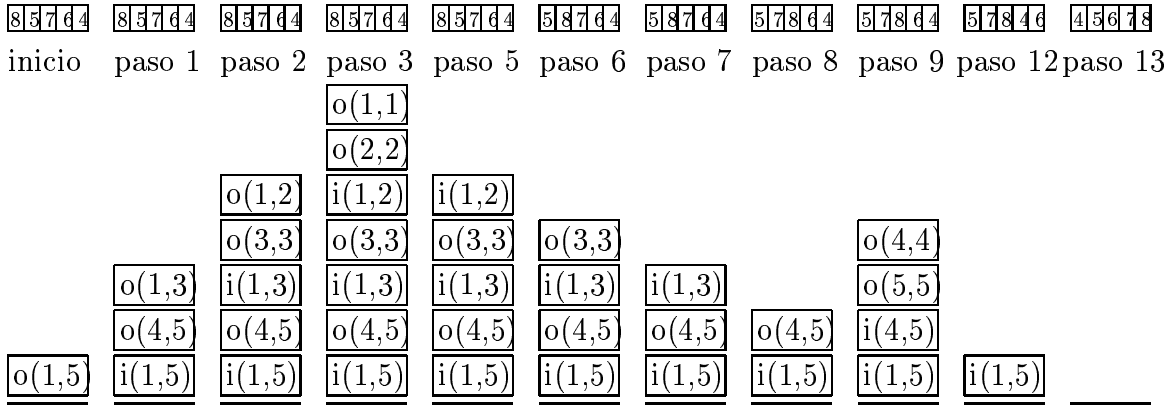
```
    top : stack → elem                          {se aplica sólo a una pila no vacía}
    pop : stack → stack                         {se aplica sólo a una pila no vacía}
    is_empty : stack → bool
ecuaciones
    top(push(e,s)) = e
    pop(push(e,s)) = s
    is_empty(empty) = true
    is_empty(push(e,s)) = false
```

Las ecuaciones establecen el significado de top, pop, is_empty en función de los **constructores** empty y push. No hay ecuaciones, en cambio, que establezcan el significado de empty y push. Esto se debe a que empty y push son precisamente los **constructores** del TAD pila, empty construye la pila vacía y push construye pilas complejas a partir de pilas más simples. Todas las pilas se construyen usando empty y push, y todas ellas son pilas distintas entre sí.

La pila es un tipo de dato muy utilizado en computación. El siguiente ejemplo ilustra cómo el tipo pila puede utilizarse para implementar la recursión. Pensemos por ejemplo en el algoritmo imperativo definido usando recursión dado anteriormente para realizar un mergeSort. Reescribiremos ahora dicho algoritmo utilizando, en vez de recursión, una pila de tareas pendientes.

```
type ttarea = tuple
                tipo: (ordenar, intercalar)
                desde: int
                hasta: int
              end
proc mergeSort (a: array[1..n] of T)
    var p: stack of ttarea                          {pila de tareas pendientes}
    var med: int
    empty(p)
    push((ordenar,1,n),p)          {inicialmente la tarea pendiente es ordenar todo}
    while ¬ is_empty(p) do                      {mientras queden tareas pendientes}
        t:= top(p)                                {tomar la primera tarea a realizar}
        pop(p)                                 {eliminarla de la pila de las pendientes}
        med:= (t.desde+t.hasta) ÷ 2
        if t.tipo == ordenar ∧ t.desde==t.hasta → skip
          t.tipo == ordenar ∧ t.desde<t.hasta →
                                          {agregar las siguientes 3 tareas pendientes}
                push((intercalar,t.desde,t.hasta),p)
                push((ordenar,m+1,t.hasta),p)
                push((ordenar,t.desde,m),p)
          t.tipo == intercalar →
                realizar intercalación a[t.desde,med] con a[med+1,t.hasta]
        fi
    od
end
```

Gráficamente, se puede seguir este algoritmo para el arreglo 8,5,7,6,4 de la siguiente forma. En el gráfico, o(i,j) es la tarea "ordenar a desde i hasta j" mientras que i(j,k) es "intercalar j a m con m+1 a k, donde m = (i+j) ÷ 2'. En la parte superior está el arreglo que hay que ordenar y en la parte inferior la pila. Entre ambos, "paso i" significa que se ilustra el estado después de i ejecuciones del ciclo **while**.



Otro ejemplo de la utilización de la pila es la evaluación de expresiones en **notación polaca inversa**. Ésta es una notación según la cual los operadores se escriben después de sus operandos. Sabiendo la aridad de cada operador, el uso de paréntesis se vuelve innecesario. Por ejemplo, la expresión $(32 + 25) * 14 + (8 - 2) \div 7 - 29$ se escribe $[32, 25, +, 14, *, 8, 2, -, 7, \div, +, 29, -]$ asumiendo que todos los operadores son binarios.

Con la ayuda de una pila es sencillo evaluar. Simplemente comenzamos con una pila vacía y leemos secuencialmente la expresión en notación polaca inversa. Cada número que se lee es ingresado a la pila. Cada operador de aridad n que se lee opera sobre los primeros n valores que hay en la pila, éstos son desapilados y se apila el resultado de hacer esa aplicación. El algoritmo puede escribirse así, asumiendo como en nuestro ejemplo que todos los operadores son binarios. También se asume que la expresión en polaca inversa es correcta (esto garantiza que no se hace top ni pop de la pila vacía).

```
var l: list                                   {expresión en notación polaca inversa}
var p: stack of int                                {pila de resultados parciales}
empty(p)
while l ≠ [ ] do                                  {mientras no se agota la expresión}
        h:= hd(l)
        l:= tl(l)
        if h es un número → push(h,p)
          h es un operador binario "op"→
                m:= top(p)                          {m es el segundo argumento de "op"}
                pop(p)                                                  {desapilamos m}
                n:= top(p)                           {n es el primer argumento de "op"}
                pop(p)                                                  {desapilamos n}
                push(n "op" m,p)    {apilamos el resultado de realizar la operación "op"}
        fi
od
r:= top(p)                                          {r es el resultado de la expresión}
```

Una forma sencilla de implementar pilas es con un arreglo donde se van alojando los elementos. Como los arreglos tienen tamaño fijo y las pilas no, hace falta un indicador de cuántos elementos hay en la pila.

```
type stack = tuple
               elems: array[1..N] of elem
               size: int
             end
```

Con esta representación, el campo elems es el arreglo en el que se alojan los elementos de la pila y el campo size indica cuántos elementos se encuentran alojados actualmente en la pila. La pila p se inicializa como la pila vacía, asignando 0 al campo size de p.

```
proc empty(out p:stack)
    p.size:= 0
end
```

El primer elemento que se agrega a la pila p se aloja en p.elems[1], el segundo en p.elems[2], etc. A medida que se agregan dichos elementos, el campo p.size adopta los valores 1, 2, etc. Es fácil observar que el campo size indicará la última celda del arreglo elems ocupada por la pila. En efecto, los elementos de la pila p se encontrarán en las posiciones p.elems[1], p.elems[2], ..., p.elems[p.size] en el orden en que ingresaron a p.

Implementada la pila de esta manera, para agregarle un elemento es necesario que quede espacio en el arreglo. Eso se indica a continuación con la precondición ¬is_full(p). Al agregar un elemento a la pila, su número de elementos se incrementa en 1, lo que debe consignarse incrementando el campo size. Como el campo size indicaba la última celda del arreglo elems ocupada por la pila, ahora que fue incrementada indica la primera celda libre. Allí es donde se aloja el nuevo elemento e.

```
proc push(in e:elem,in/out p:stack)          {se aplica a p sólo cuando ¬is_full(p)}
    p.size:= p.size+1
    p.elems[p.size]:= e
end
```

La función top debe devolver el elemento más nuevo de la pila. Como se observó, éste se encontrará en la celda indicada por el campo size.

```
fun top(p:stack) ret e:elem                  {se aplica a p sólo cuando ¬is_empty(p)}
   e:= p.elems[p.size]
end
```

El procedimiento pop debe suprimir el elemento más nuevo de la pila. Esto se realiza decrementando el campo size. Efectivamente, esto no sólo indica que la pila se ha achicado, sino también que la última celda ocupada por la pila se encuentra una posición antes de lo que se encontraba en la pila original. De este modo, el elemento que se pretendía suprimir permanece en el arreglo pero no en la parte del arreglo ocupado por la pila. Es decir, dicho elemento ha sido suprimido de la pila.

```
proc pop(in/out p:stack)                     {se aplica a p sólo cuando ¬is_empty(p)}
    p.size:= p.size-1
end
```

Para comprobar si una pila es vacía o no, alcanza con observar su campo size. Se trata de una pila vacía sii ese campo es 0.

```
fun is_empty(p:stack) ret b:bool
    b:= (p.size == 0)
end
```

Por último, esta implementación impone una restricción al tamaño máximo de la pila. Se implementa una función is_full que comprueba si el arreglo está lleno, en cuyo caso no se pueden agregar elementos a la pila hasta que vuelva a generarse espacio mediante el procedimiento pop o empty:

```
fun is_full(p:stack) ret b:bool
    b:= (p.size == N)
end
```

Cada una de las operaciones que se presentadas realizan un número fijo de operaciones elementales, por lo que todas ellas son de tiempo constante $\Theta(1)$.

Como acabamos de ver, la implementación con arreglos impone una restricción en el tamaño de la pila: no puede exceder al del arreglo, N. Si el lenguaje de programación contiene un tipo concreto para las listas, entonces se puede implementar pilas con listas, evitando al menos en principio la restricción del tamaño.

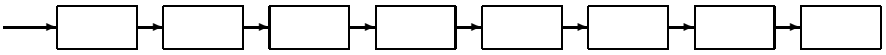
```
type stack = [elem]
proc empty(out p:stack)
     p:= [ ]
end
proc push(in e:elem,in/out p:stack)
     p:= (e ▷ p)
end
fun top(p:stack) ret e:elem                    {se aplica a p sólo cuando p≠ [ ]}
    e:= hd(p)
end
proc pop(in/out p:stack)                       {se aplica a p sólo cuando p ≠ [ ]}
     p:= tl(p)
end
fun is_empty(p:stack) ret b:Bool
    b:= (p = [ ])
end
```

Observemos que esta implementación también da operaciones constantes $\Theta(1)$.

TAD cola. El tipo abstracto **cola** (o **queue**) es el que se encuentra usualmente en casi cualquier lugar donde haya, como su nombre lo indica, una cola para atención: caja de un banco, o de supermercado, etc. Un ejemplo en computación son los programas que controlan la atención de pedidos a una impresora: manejan una cola y van atendiendo los pedidos en orden de llegada. Lo mismo con las listas de espera para conseguir pasajes en un vuelo totalmente lleno. Se establece una cola de espera y a medida que se liberan asientos en ese vuelo se otorgan según el orden de llegada a la cola de espera.

El funcionamiento es sencillo: incialmente tenemos la cola **vacía**, cuando alguien llega a la cola, se **agrega al final**, uno puede **ver** al que está **primero en la cola**, cuando uno atiende a alguien, atiende **al primero** y **lo elimina** de la cola. Uno siempre puede fijarse si la cola **está vacía o no**.

Gráficamente la representación más usual es



donde a la izquierda se encuentra el primero a ser atendido, los que se agreguen lo harán por la derecha.

Hemos resaltado las 5 propiedades que definen el TAD cola: la existencia de una cola vacía (**empty**), la posibilidad de encolar (**enqueue**), la de ver el próximo en atender (**first**) si la cola no está vacía, la de decolar el primero que se agregó (**dequeue**) si la cola no está vacía, la de controlar si la cola está vacía (**is_empty**). La característica de la cola es que se extrae el primero que se introdujo (FIFO = Last In, First Out).

La especificación del TAD cola se detalla a continuación.

TAD queue[elem]
constructores
empty : queue
enqueue : queue × elem → queue
operaciones
first : queue → elem {se aplica sólo a una cola no vacía}
dequeue : queue → queue {se aplica sólo a una cola no vacía}
is_empty : queue → bool
ecuaciones
first(enqueue(empty,e)) = e
first(enqueue(enqueue(q,e'),e)) = first(enqueue(q,e'))
dequeue(enqueue(empty,e)) = empty
dequeue(enqueue(enqueue(q,e'),e)) = enqueue(dequeue(enqueue(q,e')),e)
is_empty(empty) = true
is_empty(enqueue(q,e)) = false

Así como pilas pueden implementarse con arreglos, también las colas se pueden implementar con arreglos. Nuevamente el tamaño de la cola no podrá exceder al del arreglo.

La primera idea que se podría intentar es la de imitar en todo lo posible la implementación de pilas usando arreglos. La diferencia es que en el caso de la cola se elimina el elemento que ingresó primero. Suponiendo que se encuentran k elementos en la cola q en las posiciones q.elems[1], q.elems[2], ..., q.elems[k], al eliminar un elemento habría que eliminar el que se encuentra en q.elems[1] a diferencia de lo que ocurría en el caso de la pila, donde se eliminaba el que se encontraba en q.elems[k]. Deben permanecer en la cola los siguientes k-1 elementos: q.elems[2], ..., q.elems[k]. Éstos pueden reubicarse en las posiciones q.elems[1], ..., q.elems[k-1] haciendo las asignaciones

q.elem[1]:= q.elems[2]
q.elems[2]:= q.elems[3]
...
q.elems[k-1]:= q.elems[k]

que se pueden expresar simplemente usando un **for** adecuado, pero demandaría, como se ve, k-1 asignaciones. Esto volvería al algoritmo lineal en vez de constante como han sido todas las operaciones de pilas que se implementaron recientemente.

¿Pueden implementarse las colas sobre arreglos de modo que las operaciones sean constante? La respuesta es afirmativa. Para ello debemos pensar en algo más ingenioso que reubicar los elementos luego de eliminar el primero.

Una posibilidad es dejarlos donde están y agregar, además del campo size, un campo fst que señale la celda donde se encuentra el primer elemento de la cola.

Así, tendremos entonces que los elementos de la cola q estarán en las posiciones q.elems[q.fst], q.elems[q.fst+1], . . . , q.elems[q.fst+q.size]. Una desventaja que se observa fácilmente es que las celdas anteriores a q.elems[q.fst] quedan desaprovechadas y, por ello, el tamaño máximo de la cola limitado aún más.

Para no desaprovechar esas celdas, se piensa a q.elems como un **arreglo circular**. Es decir, si el último elemento de la cola ocupa la celda q.elems[N], el siguiente elemento debería alojarse en la celda q.elems[1] (si esta ya se desocupó). El siguiente se aloja en q.elems[2], etc.

Para lograr ese comportamiento circular, es conveniente indexar el arreglo elems con números de 0 a N-1 en vez de números de 1 a N y utilizar aritmética módulo N para acceder a las celdas del arreglo.

```
type queue = tuple
                elems: array[0..N-1] of elem
                size: int
                fst: int
             end
```

La cola q se inicializa como la cola vacía asignando 0 al campo size de q. También debe asignarse cualquier valor entre 0 y N-1 a su campo fst para que el mismo tenga un valor consistente (debe señalar una celda de q.elems).

```
proc empty(out q:queue)
    q.size:= 0
    q.fst:= 0
end
```

Como la cola puede llenarse, antes de agregar un elemento debemos asegurarnos que la cola no esté llena. El elemento nuevo se agrega al final, es decir, en la primera celda libre de q.elems "después"de la última celda ocupada por la cola. Dicha celda se encuentra q.size lugares "después"de la primer celda ocupada por la cola, es decir, de q.elems[q.fst]. Las comillas se deben a que "después"debe interpretarse teniendo en cuenta que se trata de un arreglo circular. Es decir, la celda en cuestión, que se encuentra q.size lugares "después"de q.elems[q.fst] es la celda q.elems[(q.fst+q.size) mod N]. Luego de alojar e en dicha posición se consigna que la cola tiene un elemento más incrementando el campo size de q.

```
proc enqueue(in/out q:queue,in e:elem)      {se aplica a q sólo cuando ¬is_full(q)}
    q.elems[(q.fst+q.size) mod N]:= e
    q.size:= q.size+1
end
```

La función que devuelve el primer elemento de la cola no tiene más que buscarlo donde se sabe que está: en la posición de q.elems indicada por q.fst.

```
fun first(q:queue) ret e:elem                {se aplica a q sólo cuando ¬is_empty(q)}
    e:= q.elems[q.fst]
end
```

El procedimiento que elimina el primer elemento de la cola q debe decrementar el campo size de q. Además, una vez eliminado el primer elemento de la cola pasa a ser el que anteriormente era el segundo, que se encuentra 1 lugar "después"de q.elems[q.fst], es decir en q.elems[(q.fst+1) mod N]. Se debe modificar q.fst para que señale ésta posición.

```
proc dequeue(in/out q:queue)                 {se aplica a q sólo cuando ¬is_empty(q)}
     q.size:= q.size-1
     q.fst:= (q.fst+1) mod N
end
```

La función que determina si una cola es vacía no necesita más que comprobar el tamaño de la misma inspeccionando el campo size.

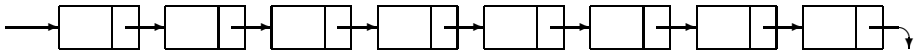
```
fun is_empty(q:queue) ret b:Bool
    b:= (q.size == 0)
end
```

De la misma forma, una cola está llena cuando todas las celdas del arreglo están ocupadas por la cola, es decir, cuando el número de celdas ocupadas por la cola es N.

```
fun is_full(q:queue) ret b:Bool
    b:= (q.size == N)
end
```

Hay algunas variantes para implementar colas usando arreglos circulares. Por ejemplo, se puede reemplazar el campo size por otro entero que indique cuál es la primer celda libre. Sin embargo, esto lleva a desaprovechar una celda del arreglo o a complicar un poco la implementación.

Listas Enlazadas. Por **listas enlazadas** se entiende una manera de implementar listas utilizando tuplas y punteros. Hay una variedad de listas enlazadas, la representación gráfica de la versión más simple de las mismas es la siguiente.



Es parecida a la de cola, con la diferencia de que cada **nodo** se dibuja como una tupla y la flecha que enlaza un nodo con el siguiente nace desde un campo de esa tupla. Los nodos son tuplas y las flechas punteros:

```
type node = tuple
                value: elem
                next: pointer to node
            end
type list = pointer to node
```

Es decir que una lista es en realidad un puntero a un primer nodo, que a su vez contiene un puntero al segundo, éste al tercero, y así siguiendo hasta el último, cuyo puntero es **null** significando que la lista termina allí. Es decir que para acceder al i-ésimo

elemento de la lista, debo recorrerla desde el comienzo siguiendo el recorrido señalado por los punteros. Esto implica que el acceso a ese elemento no es constante, sino $\Theta(i)$. A pesar de ello ofrecen una manera de implementar convenientemente algunos TADs.

Por ejemplo, a continuación se implementa el TAD pila usando listas enlazadas.

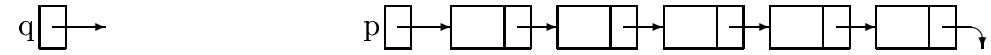
```
type node = tuple
              value: elem
              next: pointer to node
            end
type stack = pointer to node
```

El procedimiento empty inicializa p como la pila vacía. Ésta se implementa con la lista enlazada vacía que consiste de la lista que no tiene ningún nodo, es decir, el puntero al primer nodo de la lista no tiene a quién apuntar. Su valor se establece en **null**.

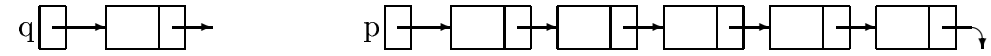
```
proc empty(out p:stack)
    p:= null
end
```

El procemiento push debe alojar un nuevo elemento en la pila. Para ello crea un nuevo nodo (alloc(q)), aloja en ese nodo el elemento a agregar a la pila (q->value:= e), enlaza ese nuevo nodo al resto de la pila (q->next:= p) y finalmente indica que la pila ahora empieza a partir de ese nuevo nodo que se agregó (p:= q).

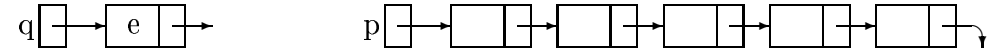
```
proc push(in e:elem,in/out p:stack)
    var q: pointer to node
```



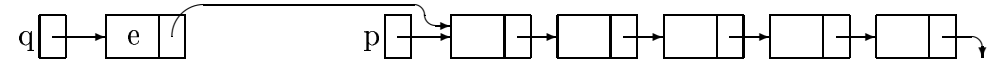
alloc(q)



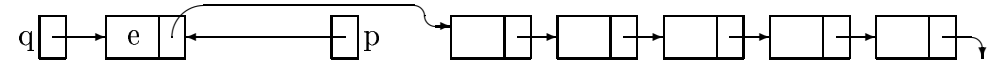
q->value:= e



q->next:= p



p:= q




```
end
```

En limpio queda

```
proc push(in e:elem,in/out p:stack)
    var q: pointer to node
    alloc(q)
    q->value:= e
    q->next:= p
    p:= q
end
```

El valor de las representaciones gráficas que acompañan al código es muy relativo. Sólo sirven para entender lo que está ocurriendo de manera intuitiva. Hacer un tratamiento formal está fuera de los objetivos de este curso. De todas maneras, deben extremarse los cuidados para no incurrir en errores de programación que son muy habituales en el contexto de la programación con punteros.

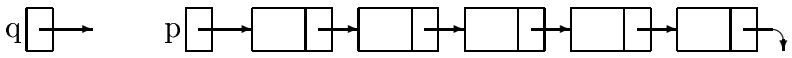
En el ejemplo del procedimiento push además de la representación gráfica que parece indicar que el algoritmo es correcto, habría que asegurarse de que el algoritmo presentado funcione también cuando la pila p a la que se agrega e esté vacía, caso no contemplado en las representaciones gráficas que incluimos en el código. Dejamos al lector la tarea de comprobar que efectivamente se comporta de manera correcta cuando la pila está vacía, es decir, cuando el puntero p es **null**.

La función top no tiene más que devolver el elemento que se encuentra en el nodo apuntado por p.

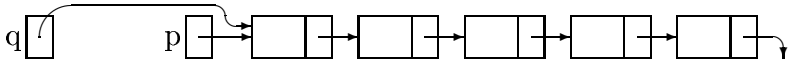
```
fun top(p:stack) ret e:elem          {se aplica a p sólo cuando ¬is_empty(p)}
    e:= p->value
end
```

El procedimiento pop debe liberar el primer nodo de la lista y modificar p de modo que pase a apuntar al nodo siguiente. Observar que el valor que debe adoptar p se encuentra en el primer nodo. Por ello, antes de liberarlo es necesario utilizar la información que en él se encuentra. Por otro lado, si modifico el valor de p, ¿cómo voy a hacer luego para liberar el primer nodo que sólo era accesible gracias al viejo valor de p? La solución es recordar en una variable q el viejo valor de p (q:= p), hacer que p apunte al segundo nodo (p:= p->next) y liberar el primer nodo (free(q)). Observar que una vez liberado, p apunta al primer nodo de la nueva pila.

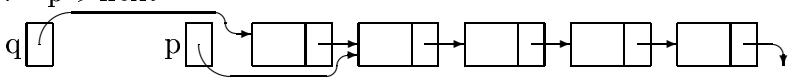
```
proc pop(in/out p:stack)             {se aplica a p sólo cuando ¬is_empty(p)}
    var q: pointer to node
```



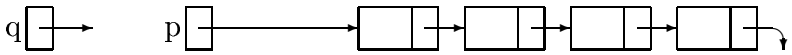

```
    q:= p
```




```
    p:= p->next
```




```
    free(q)
```




```
end
```

En limpio queda

```
proc pop(in/out p:stack)             {se aplica a p sólo cuando ¬is_empty(p)}
    var q: pointer to node
    q:= p
    p:= p->next
    free(q)
end
```

Nuevamente, se deja al lector la tarea de asegurarse de que el procedimiento pop funciona bien cuando la pila p tiene un sólo elemento.

La función is_empty debe comprobar simplemente que la pila recibida corresponda a la lista enlazada vacía que se representa por el puntero **null**.

```
fun is_empty(p:stack) ret b:Bool
    b:= (p == null)
end
```

Como el manejo de la memoria es explícito, es conveniente agregar una operación para destruir una pila. Esta operación recorre la lista enlazada liberando todos los nodos que conforman la pila. Puede definirse utilizando las operaciones proporcionadas por la implementación del TAD pila.

```
proc destroy(in/out p:stack)        {libera todo el espacio de memoria ocupado por p}
    while ¬ is_empty(p) do pop(p) od
end
```

Por último, es fácil comprobar que todas las operaciones definidas resultan constante $\Theta(1)$ con la sola excepción del procedimiento destroy que recorre toda la lista, por lo que su comportamiento es lineal $\Theta(n)$ donde n es el número de elementos de la pila.

Si uno quiere implementar el tipo cola usando listas enlazadas, puede usar la misma representación. La dificultad principal estará en la implementación de enqueue que deberá recorrer toda la lista para llegar al final de la misma, posición en donde debe agregarse el nuevo elemento. Asumiendo entonces que se ha definido

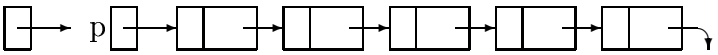
```
type queue = ^node
```

donde node se define igual que para las pilas, se puede definir empty, first, dequeue, is_empty, destroy en forma casi idéntica a como se definió empty, top, pop, is_empty, destroy para las pilas. Sólo cambia la implementación de enqueue, que sería como sigue.

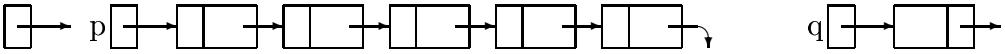
```
proc enqueue(in/out p:queue,in e:elem)
    var q,r: pointer to node
    alloc(q)                                          {se reserva espacio para el nuevo nodo}
    q->value:= e                                      {se aloja allí el elemento e}
    q->next:= null                {ahora el nodo ⋆q debe agregarse al final de la cola}
    if p == null → p:= q                              {si la cola es vacía con esto alcanza}
      p ≠ null →                  {si no es vacía, se inicia la búsqueda de su último nodo}
                  r:= p                 {r realiza la búsqueda a partir del primer nodo}
                  while r->next ≠ null do      {mientras ⋆r no sea el último nodo}
                      r:= r->next              {que r pase a señalar el nodo siguiente}
                  od
                  r->next:= q      {que el siguiente del que era último sea ahora ⋆q}
    fi
end
```

Como en casos anteriores, a continuación decoramos este algoritmo con gráficos que faciliten su comprensión. Observar que en la primera parte del algoritmo la cola puede o no estar vacía, luego del condicional **if** se distinguen esos dos casos, por lo que en una rama la cola es vacía y en la otra no.

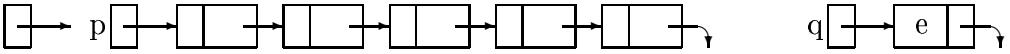
proc enqueue(**in**/**out** p:queue,**in** e:elem)
var q,r: **pointer to** node



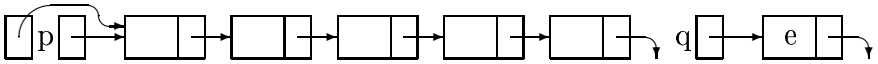
alloc(q)



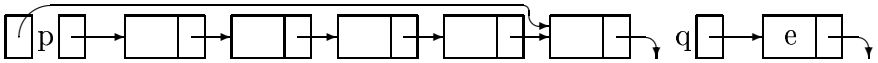
q->value:= e
q->next:= **null**



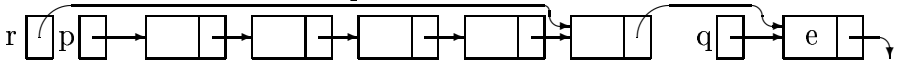
if p == **null** → p:= q {no engañarse con el dibujo, la cola puede ser vacía}
p ≠ **null** → r:= p



while r->next ≠ **null** **do** {mientras ⋆r no sea el último nodo}
r:= r->next {que r pase a señalar el nodo siguiente}
od

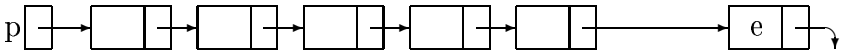


r->next:= q



fi
end

Al finalizar la ejecución de enqueue, las variables locales q y r desaparecen. Queda

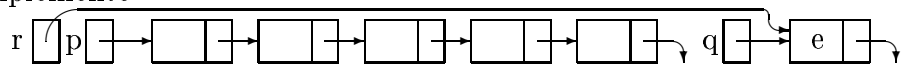


que es la cola con el nuevo elemento agregado al final. El invariante del **while** podría enunciarse diciendo "el último nodo de la cola se encuentra siguiendo las flechas a partir de r".

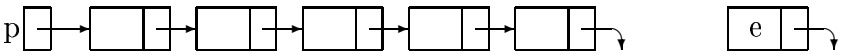
Este algoritmo revela que no es sencillo trabajar con punteros. Exige extremar las precauciones para mantener la integridad de la estructura. Por ejemplo, si bien en la quinta línea la asignación de **null** a q->next puede parecer poco importante, eso es lo que garantiza que una vez terminado el procedimiento, el puntero del último nodo de la cola sea **null**. Por otro lado, la condición del **while**, r->next ≠ **null** requiere que r no sea **null**. Afortunadamente esto es cierto porque inicialmente r es p que a su vez es distinto de **null**, y en el cuerpo del **while** siempre se asigna a r un valor distinto de **null**. Es decir, el invariante implica que r no es **null**. La condición del **while** también requiere que r señale una dirección de memoria que ha sido reservada. Afortunadamente, esto también es consecuencia del invariante porque toda la cola p se asume compuesta de espacios de memoria que han sido oportunamente reservados. Por supuesto que para que esto a su vez sea cierto, uno debe asegurarse de que la implementación de todas las operaciones del TAD cola preserven la integridad de la estructura.

Por último, puede parecer correcto simplificar la condición del **while**, reemplazandolo por **while** r ≠ **null** **do** r:= r->next **od**. Si bien esto también recorre toda la cola, cuando

el **while** termina se tiene r == **null**, es decir que r no señala al último nodo de la cola, que es donde debe insertarse q. La asignación r:= q modificaría r, pero no modificaría la última flecha de la cola p para que señale al nuevo nodo *q. Gráficamente, obtendríamos simplemente



que evidentemente no inserta el nuevo elemento al final. Además, al final del procedimiento, al desaparecer las variables locales q y r, el nuevo nodo no sería accesible jamás. Quedaría



donde lo único accesible es lo que se puede alcanzar siguiendo las flechas partiendo de una variable. En este caso, la única variable es p por lo que el nodo de la derecha no se libera pero se pierde.

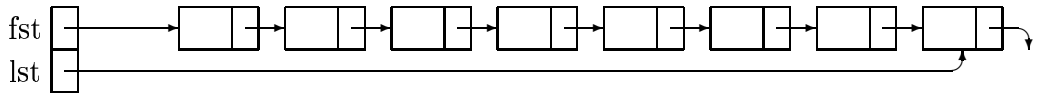
De la misma manera, si usáramos p mismo para recorrer el arreglo en vez de declarar una variable auxiliar r, perderíamos la información que originariamente tenía p, y por lo tanto, perderíamos acceso al comienzo de la lista y, con ello, a toda la lista.

Esta implementación del TAD cola no sólo involucra una recorrida de la cola, que es lo más complicado que hemos visto hasta ahora con punteros, si no que a causa de esa misma recorrida la operación enqueue obtenida es $\Theta(n)$ donde n es la longitud de la cola, es decir, es ineficiente.

Una implementación más eficiente puede obtenerse manteniendo dos punteros en vez de sólo uno; uno al primer nodo de la cola y otro al último:

```
type queue = tuple
               fst: pointer to node
               lst: pointer to node
             end
```

Gráficamente, puede representarse de la siguiente manera



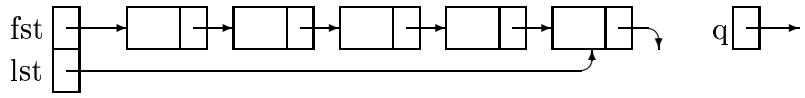
Las operaciones del tipo cola se implementan a continuación:

```
proc empty(out p:queue)
    p.fst:= null
    p.lst:= null
end
```

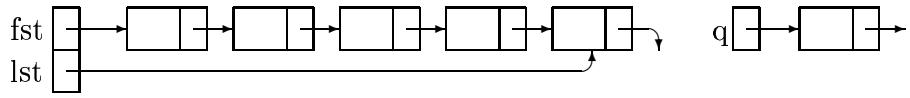
Es decir, la cola vacía estará representada por los dos punteros igualados a **null**. Sólo en un caso más ambos punteros pueden coincidir. En efecto, cuando la cola tenga exactamente un elemento éste será a la vez el primero y último. En ese caso tanto el campo fst como el campo lst de p tendrán la dirección de ese único nodo de la cola.

Para el procedimiento enqueue necesitamos, como en varios de los procedimientos vistos, un puntero auxiliar q para crear el nuevo nodo.

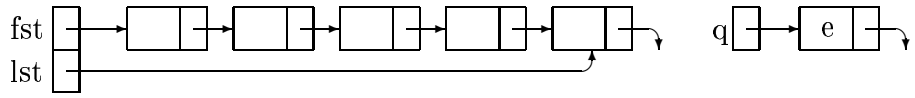
proc enqueue(**in**/**out** p:queue,**in** e:elem)
var q: **pointer to** node



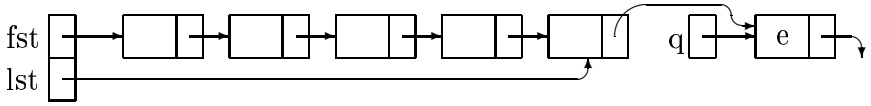
alloc(q)



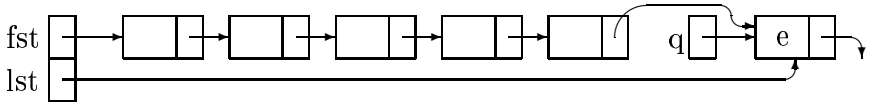
q->value:= e
q->next:= **null**



if p.lst == **null** → p.fst:= q
p.lst:= q {caso enqueue en cola vacía}
p.lst ≠ **null** → p.lst->next:= q



p.lst:= q



fi
end

Pasamos en limpio el procedimiento enqueue borrando los gráficos que hemos usado para describir los cambios de estado. Nos queda:

proc enqueue(**in**/**out** p:queue,**in** e:elem)
var q: **pointer to** node
alloc(q)
q->value:= e
q->next:= **null**
if p.lst == **null** → p.fst:= q
p.lst:= q
p.lst ≠ **null** → p.lst->next:= q
p.lst:= q
fi
end

El primer elemento se encuentra en el nodo señalado por el campo fst de p.

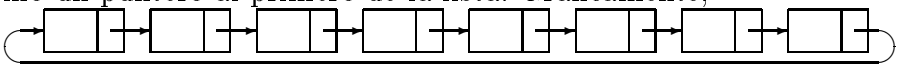
fun first(p:queue) **ret** e:elem {se aplica a p sólo cuando ¬is_empty(p)}
e:= p.fst->value
end

En el procedimiento dequeue se utiliza q para conservar temporariamente la dirección del primer nodo y así poder liberar ese nodo al final del procedimiento, cuando la información que contiene deja de ser necesaria.

```
proc dequeue(in/out p:queue)          {se aplica a p sólo cuando ¬is_empty(p)}
    var q: pointer to node
    q:= p.fst
    if p.fst == p.lst → p.fst:= null                {caso cola con un solo elemento}
                        p.lst:= null
      p.fst ≠ p.lst → p.fst:= p.fst->next
    fi
    free(q)
end
fun is_empty(p:queue) ret b:Bool
   b:= (p.fst == null)
end
proc destroy(in/out p:queue)      {libera todo el espacio de memoria ocupado por p}
    while ¬ is_empty(p) do dequeue(p) od
end
```

En este implementación todas las operaciones resultaron constante $\Theta(1)$ excepto destroy que como debe recorrer toda la cola es lineal $\Theta(n)$ donde n es la cantidad de elementos de la cola.

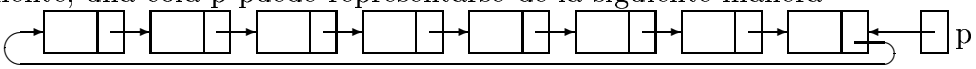
Una implementación alternativa igualmente eficiente es posible usando listas circulares. Éstas son listas enlazadas en las que el último nodo no tiene un puntero con valor nulo **null** sino un puntero al primero de la lista. Gráficamente,



Con esta representación, evidentemente no hace falta mantener un puntero al primero de la cola, alcanza con un puntero al último, aquél puede conseguirse a partir de éste en $\Theta(1)$.

```
type queue = pointer to node
```

Gráficamente, una cola p puede representarse de la siguiente manera



Las operaciones del TAD cola se implementan a continuación. La cola vacía se representa como es habitual con **null**.

```
proc empty(out p:queue)
    p:= null
end
```

Para el procedimiento enqueue volvemos a utilizar un puntero auxiliar, sólo que esta vez luego de utilizarlo para reservar espacio para el nuevo (último) nodo no asignamos **null** al puntero de dicho nodo. Esto se debe a que el último nodo de la cola, en esta representación, debe apuntar al primero en vez de tener el valor **null** como en las versiones anteriores. La novedad radica en "insertar"el nodo en la lista sin romper la circularidad.

```
proc enqueue(in/out p:queue,in e:elem)
    var q: pointer to node
    alloc(q)
    q->value:= e
    if p == null → p:= q                                  {caso enqueue en cola vacía}
                   q->next:= q
      p ≠ null → q->next:= p->next          {que el último nuevo apunte al primero}
                 p->next:= q          {que el último viejo apunte al último nuevo}
                 p:= q                    {que p también apunte al último nuevo}
    fi
end
```

En esta representación, para devolver el primero de la cola es necesario acceder al campo value, no de p que es el último de la cola, sino de p->next que es el primero.

```
fun first(p:queue) ret e:elem          {se aplica a p sólo cuando ¬is_empty(p)}
   e:= p->next->value
end
```

Para eliminar el primero de la cola usamos un puntero auxiliar q que señala el nodo que ha de eliminar, se realiza la eliminación preservando la circularidad de la cola. Es decir, en caso de tener un sólo elemento la cola pasa a ser la cola vacía, y en caso de tener más de un elemento, el puntero del último nodo debe ahora señalar el nodo que hasta entonces era el segundo de la cola.

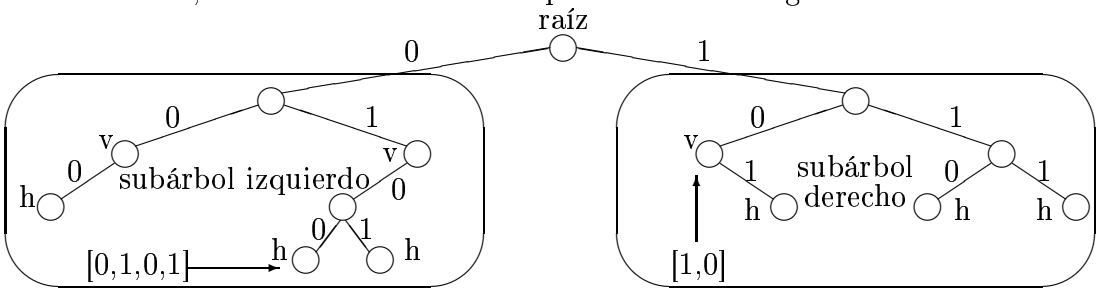
```
proc dequeue(in/out p:queue)           {se aplica a p sólo cuando ¬is_empty(p)}
    var q: pointer to node
    q:= p->next                                                  {q apunta al primero}
    if p == q → p:= null                                {caso cola con un solo elemento}
      p ≠ q → p->next:= q->next  {que el último apunte al que antes era segundo}
    fi
    free(q)
end
```

Las dos operaciones restantes no presentan novedades.

```
fun is_empty(p:queue) ret b:Bool
   b:= (p == null)
end
proc destroy(in/out p:queue)      {libera todo el espacio de memoria ocupado por p}
    while ¬ is_empty(p) do dequeue(p) od
end
```

TAD árbol binario. Ésta es una estructura muy usada en numerosas aplicaciones. Su nombre y demás terminología afín proviene de los árboles de la botánica, pero sobre todo de los árboles genealógicos. A pesar de que la definición puede generalizarse naturalmente, definiremos sólo los árboles binarios. Un árbol binario puede adoptar una de estas dos formas: o bien es un **árbol vacío** o bien es un **nodo** que, además de un elemento, tiene **dos subárboles** (de ahi el vocablo "binario"). Así, un árbol no vacío tiene una **raíz** (el elemento), un **subárbol izquierdo** y un **subárbol derecho**.

Gráficamente, los árboles binarios se representan de la siguiente forma:



En esta representación, los árboles vacíos no se dibujan, sólo se dibujan los nodos. Uno puede fácilmente deducir dónde se encuentran los árboles vacíos ya que cada nodo debe tener exactamente 2 subárboles. Así, los nodos marcados con una "v" tienen un subárbol vacío, y los marcados con una "h" tienen sus 2 subárboles vacíos.

TAD bintree[elem]
constructores
<> : bintree
< _, _, _ > : bintree × elem × bintree → bintree
operaciones
root : bintree → elem {se aplica sólo a un árbol no vacío}
left : bintree → bintree {se aplica sólo a un árbol no vacío}
right : bintree → bintree {se aplica sólo a un árbol no vacío}
is_empty : bintree → bool
ecuaciones
root(<l,e,r>) = e
left(<l,e,r>) = l
right(<l,e,r>) = r
is_empty(<>) = true
is_empty(<l,e,r>) = false

Un nodo con sus dos subárboles vacíos (<<>,e,<>>) se llama **hoja** (en el ejemplo gráfico, son los marcados con una "h") y se denota más brevemente <e>. Dado un nodo n, sus 2 subárboles izquierdo y derecho se denominan usualmente **hijos**, respectivamente **hijo izquierdo** e **hijo derecho** de n. A su vez, n se dice **padre** de sus hijos. Dos árboles son **hermanos** cuando son hijo izquierdo y derecho del mismo padre. Un **camino** es una secuencia $A_1, A_2, \ldots, A_n$ de árboles tales que para cada i, A_{i+1} es hijo de A_i. La **longitud** de este camino es n-1. Además, A_1 se dice **ancestro** o **ascendiente** de A_n y A_n **descendiente** o **subárbol** de A_1. Decimos que siempre hay un camino de longitud 0 de todo árbol a sí mismo. La **altura** de un árbol es la longitud del camino que va desde él hasta su hoja más lejana. La **profundidad** de un subárbol es la longitud del camino que va desde el árbol hasta él. Se llama **nivel** a todo conjunto de subárboles de igual profundidad.

Intuitivamente, si seguimos la representación gráfica, un camino se puede identificar con un recorrido descendente del árbol. Toda esta terminología (salvo por los términos

"izquierdo" y "derecho") puede ser extendido sencillamente a otras definiciones de árboles, como ternarios (cada nodo tiene 3 hijos), finitarios (cada nodo tiene una cantidad finita de hijos), etc.

Resulta conveniente hablar de las posiciones en que la información puede encontrarse en un árbol. En la representación gráfica dada, se decoró con un "0" la arista que une un nodo con su hijo izquierdo, y con un "1" la que lo une con su hijo derecho. Gracias a ello, una posición cualquiera dentro del árbol puede ser señalada dando una secuencia de 0's y 1's. Esta secuencia indica, si uno parte de la raíz, qué subárbol debe elegir (izquierdo o derecho) para llegar hasta la posición indicada. En la representación gráfica se ejemplifican las posiciones indicadas por [0,1,0,1] y por [1,0]. Se define el conjunto de posiciones pos = [{0,1}].

Dado un árbol t y una posición p, definimos $t \downarrow p$ para denotar el descendiente de t que se encuentra en la posición p de t.

$<> \downarrow p = <>$
$< l, e, r > \downarrow [] = < l, e, r >$
$< l, e, r > \downarrow (0 \rhd p) = l \downarrow p$
$< l, e, r > \downarrow (1 \rhd p) = r \downarrow p$

Nótese que si p no está entre las posiciones que aparecen en la representación gráfica de t, entonces $t \downarrow p = <>$. Definimos pos(t) como el conjunto de posiciones que sí aparecen en la representación gráfica: $\text{pos}(t) = \{p \in \text{pos} \mid t \downarrow p \neq <>\}$.

Ahora podemos definir la operación $t.p$ que devuelve el elemento alojado en la posición p de t. Esta operación sólo está definida para $p \in \text{pos}(t)$.

$< l, e, r > .[] = e$
$< l, e, r > .(0 \rhd p) = l.p$
$< l, e, r > .(1 \rhd p) = r.p$

o equivalentemente, se puede definir $t.p = \text{root}(t \downarrow p)$.

Utilizando punteros hay una forma inmediata de implementar árboles binarios, consistente en reemplazar cada arista de la representación gráfica justamente por un puntero.

```
type tnode = tuple
                lft: pointer to tnode
                value: elem
                rgt: pointer to tnode
             end
type bintree = pointer to tnode

fun empty() ret t:bintree
    t:= null
end

fun node(l:bintree,e:elem,r:bintree) ret t:bintree
    alloc(t)
    t->lft:= l
    t->value:= e
    t->rgt:= r
end
```

```
fun root(t:bintree) ret e:elem                {se aplica a t sólo cuando ¬is_empty(t)}
    e:= t->value
end
fun left(t:bintree) ret l:bintree             {se aplica a t sólo cuando ¬is_empty(t)}
    l:= t->lft
end
fun right(t:bintree) ret r:bintree            {se aplica a t sólo cuando ¬is_empty(t)}
    r:= t->rgt
end
fun is_empty(t:bintree) ret b:bool
    b:= (t == null)
end
proc destroy(in/out t:bintree)     {libera todo el espacio de memoria ocupado por t}
     if ¬ is_empty(t) then destroy(left(t))
                           destroy(right(t))
                           free(t)
                           t:= null
     fi
end
```

Esta implementación, sin embargo, no suele cumplir en la práctica con uno de los objetivos principales de las implementaciones de los TADs: que las operaciones que uno necesite luego sobre árboles binarios puedan implementarse eficientemente utilizando exclusivamente las 7 operaciones definidas, sin acceder a la representación concreta que se ha elegido.

Esto se debe a que, si bien la definición de un árbol es respetada por esta implementación, en la práctica las operaciones que usualmente se requieren sobre árboles son distintas. Consisten en recorrerlos, modificar la información que se encuentra en uno de los nodos, borrar alguno de los nodos, reorganizar su estructura, asegurarse de que la estructura del árbol respeta criterios más exigentes que su definición, como por ejemplo la de ser balanceados.[4] Es decir que en la práctica no trabajaremos con implementaciones del TAD árbol binario sino con implementaciones de otras estructuras abstractas que involucran la utilización de árboles binarios.

A continuación veremos justamente otras estructuras que se basan en árboles binarios, son árboles binarios con condiciones suplementarias.

Si T es un tipo, resulta conveniente denotar por $< T >$ al tipo de los árboles binarios cuyos elementos son los de T.

Árboles binarios de búsqueda (ABB). Como su nombre lo indica son árboles binarios cuya información ha sido organizada de manera de facilitar la búsqueda de un elemento cualquiera que pueda estar almacenado en el árbol. Requiere que T esté equipado con un orden total que denotaremos $\leq$. Para que un árbol binario sea un ABB debe cumplirse, para todo nodo del árbol, que todos los elementos alojados en el hijo

[4] Intuitivamente, un árbol binario está balanceado cuando los caminos maximales (que van desde la raíz hasta cada subárbol vacío) son todos de longitud similar.

izquierdo sean menores que el alojado en el propio nodo, y que éste a su vez sea menor que todos los alojados en el hijo derecho. En símbolos,

$$\mathrm{ABB}(t) \stackrel{\mathrm{def}}{=} \forall p \in \mathrm{pos}(t).\forall q \in \mathrm{pos} \begin{cases} (p \lhd 0) +\!\!+ q \in \mathrm{pos}(t) \Rightarrow t.((p \lhd 0) +\!\!+ q) < t.p \\ (p \lhd 1) +\!\!+ q \in \mathrm{pos}(t) \Rightarrow t.p < t.((p \lhd 1) +\!\!+ q) \end{cases}$$

En un ABB la operación principal es la de búsqueda, que en estilo funcional podría definirse:

search : T × <T>→Bool {se aplica a un ABB}
search(e,<>) = false
search(e,<l,e',r>) = **if** e<e' → search(e,l)
e=e' → true
e>e' → search(e,r)
fi

Si el árbol está balanceado esta operación es equivalente a la de búsqueda binaria en un arreglo ordenado, es decir, es $\Theta(\log n)$ donde n es el número de nodos del árbol. Otra operación importante es la de insertar un nuevo elemento en el lugar correspondiente de manera de que el resultado siga siendo un ABB.

insert : T × <T>→<T> {se aplica a un ABB}
insert(e,<>) = <e>
insert(e,<l,e',r>) = **if** e<e' → <insert(e,l),e',r>
e=e' → <l,e',r>
e>e' → <l,e',insert(e,r)>
fi

Si el árbol está balanceado esta operación también es $\Theta(\log n)$. Finalmente definimos la operación borrar que es la más difícil de definir, pues requiere la definición de una función auxiliar que calcule el máximo de un ABB, y otra que lo borre:

max : <T>→ T {se aplica a un ABB no vacío}
max(<l,e,r>) = **if** r=<> → e
r≠<> → max(r)
fi

delete_max : <T>→<T> {se aplica a un ABB no vacío}
delete_max(<l,e,r>) = **if** r=<> → l
r≠<> → <l,e,delete_max(r)>
fi

delete : T × <T>→<T> {se aplica a un ABB}
delete(e,<>) = <>
delete(e,<l,e',r>) = **if** e<e' → <delete(e,l),e',r>
e=e' ∧ l =<> → r
e=e' ∧ l ≠<> → <delete_max(l),max(l),r>
e>e' → <l,e',delete(e,r)>
fi

Todas estas operaciones también son $\Theta(\log n)$ si el árbol está balanceado.

Utilizando la representación de árboles binarios con punteros presentada más arriba, puede implementarse la función de búsqueda de manera recursiva o iterativa. En ambos casos utilizaremos la función next que determina el subárbol siguiente donde continuar la búsqueda. Evidentemente para obtener el nodo siguiente (n) es necesario comparar lo que se ve en el nodo actual (t) con lo que se está buscando (e).

```
fun next(e:elem, t:bintree) ret n: bintree
                              {se aplica sólo a t sólo cuando ¬is_empty(t) y e≠t->value}
    if e < t->value → n:= left(t)
       e > t->value → n:= right(t)
    fi
end
```

La siguiente es una versión recursiva del algoritmo de búsqueda:

```
fun search(e:elem, t:bintree) ret b:bool
    if is_empty(t) → b:= false
       ¬ is_empty(t) ∧ e = root(t) → b:= true
       ¬ is_empty(t) ∧ e ≠ root(t) → b:= search(e,next(t))
    fi
end
```

Otra manera es eliminar la recursión con una implementación iterativa:

```
fun search(e:elem, t:bintree) ret b:bool
    var p: bintree
    p:= t
    while ¬is_empty(p) ∧ e ≠ root(p) do
        p:= next(e,p)
    od
    b:= ¬is_empty(p)
end
```

El procedimiento de inserción puede implementarse también recursiva o iterativamente. La siguiente es una implementación recursiva:

```
proc insert(in e:elem,in/out t:bintree)
     if is_empty(t) → mk_leaf(e,t)
        ¬is_empty(t) ∧ e = root(t) → skip
        ¬is_empty(t) ∧ e < root(t) → insert(e,t->lft))
        ¬is_empty(t) ∧ e > root(t) → insert(e,t->rgt))
     fi
end
```

Hemos utilizado un procedimiento auxiliar mk_leaf para crear una hoja:

```
proc mk_leaf(in e:elem,out p: pointer to node)
      alloc(p)
      p->value:= e
      p->lft:= empty()
      p->rgt:= empty()
end
```

Iterativamente, se realiza una búsqueda del padre del nuevo nodo. Eso explica que el ciclo de búsqueda se ejecute sólo si ¬is_empty(next(e,p)). El primer **if** resuelve el caso en que el árbol es vacío. En caso contrario, se inicia la búsqueda del lugar donde insertar e. La búsqueda termina cuando se encuentra al padre del nodo a insertar. Eso explica que el cuerpo del ciclo se ejecute sólo si ¬is_empty(next(e,p)).

```
proc insert(in e:elem,in/out t:bintree)
    var p: pointer to node
    if is_empty(t) → mk_leaf(e,t)
      ¬is_empty(t) → p:= t
                     while e ≠ root(p) ∧ ¬is_empty(next(e,p)) do
                         p:= next(e,p)
                     od
                     if e = root(p) → skip
                       e < root(p) → mk_leaf(e,p->lft)
                       e > root(p) → mk_leaf(e,p->rgt)
                     fi
    fi
end
```

Observar que se distinguen los casos e < root(p) y e > root(p). ¿Por qué no hacer un solo caso e ≠ root(p) ejecutando mk_leaf(e,next(e,p))? Lo mismo ocurre con la versión recursiva dada más arriba.

TAD cola de prioridades. Es similar a la cola, pero en vez de utilizarse el orden de llegada como criterio para establecer el orden de atención se asigna a cada elemento una prioridad y se atiende cada elemento de acuerdo a su prioridad. La operación first devuelve el de mayor prioridad, la operación dequeue remueve el de mayor prioridad.

TAD pqueue[elem,≤]
constructores
empty : pqueue
enqueue : pqueue × elem → pqueue
operaciones
first : pqueue → elem {se aplica sólo a una cola no vacía}
dequeue : pqueue → pqueue {se aplica sólo a una cola no vacía}
is_empty : pqueue → bool
ecuaciones
enqueue(enqueue(q,e),e') = enqueue(enqueue(q,e'),e)
first(enqueue(empty,e)) = e
e ≥ e' ⇒ first(enqueue(enqueue(q,e'),e)) = first(enqueue(q,e))
{e < e' ⇒ first(enqueue(enqueue(q,e'),e)) = first(enqueue(q,e'))}
dequeue(enqueue(empty,e)) = empty
e ≥ e' ⇒ dequeue(enqueue(enqueue(q,e'),e)) = enqueue(dequeue(enqueue(q,e)),e')
{e < e' ⇒ dequeue(enqueue(enqueue(q,e'),e)) = enqueue(dequeue(enqueue(q,e')),e)}
is_empty(empty) = true
is_empty(enqueue(q,e)) = false

Las ecuaciones encerradas entre llaves son redundantes ya que se derivan de la ecuación anterior utilizando la primer ecuación.

La cola de prioridades se puede implementar de cualquiera de las formas vistas para cola, salvo que, o bien enqueue, o bien dequeue y first serán $\Theta(n)$ donde n es el tamaño de la cola. A continuación veremos que nuevamente los árboles nos proporcionan, una solución logarítimica.

Heap. Al igual que el ABB, el heap es un árbol binario con ciertas condiciones suplementarias respecto a la organización de la información. Las implementaciones del heap proporcionan implementaciones eficaces de las colas de prioridades.

Dado T equipado con un orden total $\leq$, un heap es un árbol binario tal que todos sus nodos alojan el máximo de todo el subárbol que comienza en ese nodo. En símbolos

$$\text{heap}(t) \stackrel{\text{def}}{=} \forall p \in \text{pos}(t).\forall q \in \text{pos}.\ p +\!\!+ q \in \text{pos}(t) \Rightarrow t.(p +\!\!+ q) \leq t.p$$

o equivalentemente, por transitividad de $\leq$,

$$\text{heap}(t) \stackrel{\text{def}}{=} \forall p \in \text{pos}(t).\ \begin{cases} p \lhd 0 \in \text{pos}(t) \Rightarrow t.(p \lhd 0) \leq t.p \\ p \lhd 1 \in \text{pos}(t) \Rightarrow t.(p \lhd 1) \leq t.p \end{cases}$$

Veremos que con un heap se puede implementar una cola de prioridades de manera de que first sea constante y enqueue y dequeue logarítmicas. Gracias a esto, por ejemplo, obtenemos un sencillo y eficiente algoritmo de ordenación, el heapSort.

```
proc heapSort(in/out a:array[1..n] of elem)
    var q: heap
    empty(q)
    for i:= 1 to n do enqueue(q,a[i]) od
    for i:= n downto 1 do a[i]:= first(q)
                          dequeue(q)
                       od
end
```

Este algoritmo consiste simplemente en insertar en el heap uno a uno los elementos del arreglo y luego extraerlos uno a uno volviendo a colocarlos en el arreglo. Usa el hecho de que el heap siempre tendrá al máximo en la raíz (eso explica que el segundo **for** recorra el arreglo de atrás hacia adelante). Como cada inserción y cada extracción son $\Theta(\log n)$, el heapSort es $\Theta(n \log n)$. Obsérvese la simplicidad del heapSort una vez implementado el heap.

Una implementación muy eficiente de heaps es mediante el uso de arreglos. Dado un arreglo a: **array**[1..n] **of** T, el arreglo puede ser visto como un árbol binario en el que la celda 1 tiene como hijos a las celdas 2 y 3, la 2 tiene como hijos a 4 y 5, la 3, a 6 y 7, en general, la celda n tienen como hijos a las celdas 2n y 2n+1. Como el arreglo es finito, una celda puede alcanzar a tener 2 hijos, sólo uno, o ninguno.

```
type heap = tuple
              elems: array[1..n] of elem
              size: int
            end
```

Las dos funciones que siguen devuelven en j la posición donde debe encontrarse, respectivamente, el hijo izquierdo y el hijo derecho sabiendo que el padre está en la posición i.

```
fun left(i:int) ret j:int
    j:= 2*i
end

fun right(i:int) ret j:int
    j:= 2*i+1
end
```

Análogamente, la siguiente función determina la posición j donde se encuentra el padre de un nodo que está en posición i.

```
fun father(i:int) ret j:int
    j:= i ÷ 2
end
```

Un nodo puede no tener hijos (cuando es una hoja). La siguiente función dice si el nodo que se encuentra en la posición i del heap tiene hijos o no.

```
fun has_children(h:heap, i:int) ret b:bool          {se aplica a 1 ≤ i ≤ h.size}
    b:= (left(i) ≤ h.size)
end
```

Similarmente, un nodo puede no tener padre (cuando es la raíz):

```
fun has_father(i:int) ret b:bool
    b:= (i ≠ 1)
end
```

Si un nodo tiene hijos, es conveniente poder compararlo (e intercambiarlo) con el mayor de sus hijos. Para ello, la siguiente función devuelve en j la posición donde se encuentra el mayor de sus hijos. La función asume que el nodo que se encuentra en la posición i tiene hijos y distingue el caso en que tiene ambos hijos (right(i) ≤ h.size) o sólo uno.

```
fun max_child(h:heap, i:int) ret j:int                {pre: has_children(h,i)}
    if right(i) ≤ h.size ∧ h.elems[left(i)] ≤ h.elems[right(i)] then j:= right(i)
    else j:= left(i)
    fi
end
```

El procedimiento lift, asume que i es un nodo que tiene padre y modifica el arreglo intercambiando lo que se encuentra en la posición i con lo que se encuentra en el nodo padre de i.

```
proc lift(in/out h:heap,in i:int)                  {se aplica a 2 ≤ i ≤ h.size}
     swap(h.elems,i,father(i))
end
```

A continuación, se define la función booleana must_lift que asume que i tiene padre y decide cuándo debe ejecutarse el procedimiento anterior.

```
fun must_lift(h:heap, i:int) ret b:bool                {se aplica a 2 ≤ i ≤ h.size}
    b:= (h.elems[i] >h.elems[father(i)])
end
```

El procedimiento float es el que compara el dato alojado en el nodo i con los alojados en cada uno de los ancestros de i hasta encontrarle la posición correcta. Como a lo sumo recorre un camino en un árbol binario balanceado, es en el peor caso $\Theta(\log n)$.

```
proc float(in/out h:heap,in i:int)                     {se aplica a 1 ≤ i ≤ h.size}
    var c: int
    c:= i
    while has_father(c) ∧ must_lift(h,c) do
        lift(h,c)
        c:= father(c)
    od
end
```

El procedimiento sink es el que compara el dato alojado en el nodo i con los alojados en algunos de sus descendientes hasta encontrarle una posición correcta. En cada paso lo compara con el mayor de los hijos. De esta manera, a lo sumo recorre un camino en un árbol binario balanceado. Por ello, en el peor caso es $\Theta(\log n)$.

```
proc sink(in/out h:heap, in i:int)                     {se aplica a 1 ≤ i ≤ h.size}
    var f: int
    f:= i
    while has_children(h,f) ∧ must_lift(h,max_child(h,f)) do
        f:= max_child(h,f)
        lift(h,f)
    od
end
```

Estas primitivas facilitan enormemente la tarea de implementar una cola de prioridades utilizando un heap. A continuación se presentan los detalles finales para obtener dicha implementación. El tipo pqueue resulta simplemente un sinónimo del tipo heap.

```
type pqueue = heap
```

El procedimiento que inicializa la cola de prioridades no presenta novedades.

```
proc empty(out q:pqueue)
    q.size:= 0
end
```

El procedimiento enqueue, incrementa el campo size del heap (que al ser incrementado señala la primer celda libre del arreglo) aloja en la primer celda libre el nuevo elemento y finalmente, para restrablecer la condición de heap, "hace flotar" el nuevo elemento. La única operación no constante es flotar, por ello enqueue resulta en el peor caso $\Theta(\log n)$.

```
proc enqueue(in/out q:pqueue,in e:elem)        {se aplica a q sólo cuando q.size < n}
    q.size:= q.size+1
    q.elems[q.size]:= e
    float(q,q.size)
end
```

La función first es muy sencilla. En un heap no vacío el máximo se encuentra en la raíz, que está en la posición 1 del arreglo.

```
fun first(q:pqueue) ret e:elem                {se aplica a q sólo cuando q.size > 0}
    e:= q.elems[1]
end
```

El procedimiento dequeue elimina el máximo sobreescribiendo la posición 1 del arreglo con lo que se encuentra en la última posición del heap (la señalada por size), decrementa el campo size dado que ahora va a tener un elemento menos y finalmente, para restablecer la condición de heap, "hunde" el elemento que ahora está en la raíz. La única operación no constante es hundir, por ello dequeue resulta en el peor caso $\Theta(\log n)$.

```
proc dequeue(in/out q:pqueue)                 {se aplica a q sólo cuando q.size > 0}
    q.elems[1]:= q.elems[q.size]
    q.size:= q.size-1
    sink(q,1)
end
```

Por último, la función is_empty no presenta ninguna dificultad.

```
fun is_empty(q:pqueue) ret b:bool
    b:= (q.size == 0)
end
```

Además de ser muy ingeniosa, la implementación del heap en un arreglo permite realizar el heapSort utilizando el mismo arreglo que se pretende ordenar como heap. Así, a medida que se van agregando los elementos al heap, se van utilizando para él justamente aquellas celdas del arreglo cuyos valores ya han sido guardados en el heap, por lo que sus valores no van a perderse a pesar de que esas celdas sean reutilizadas.

```
proc heapSort(in/out a:array[1..n] of elem)
    for i:= 1 to n do
        float(a,i)
    od
    for i:= n downto 1 do
        swap(a,1,i)                                          {a[i]:= first}
        sink(a,i-1,1)
    od
end
```

El segundo **for** del heapSort extrae uno a uno los elementos, con lo que el heap va liberando celdas que son justamente las que vuelve a controlar el arreglo a ordenar. No es necesario registrar en una variable aparte el tamaño del heap ya que el mismo es indicado por la variable i del primer **for**. En el segundo **for**, inmediatamente después del swap, el tamaño del heap es i-1.

Para completar esta versión de heapSort necesitamos adaptar (trivialmente) las siguientes funciones y procedimientos definidas sobre heap.

```
fun has_children(a:array[1..n] of elem, s: int, i:int) ret b:bool      {pre: 1 ≤ i ≤ s}
    b:= (left(i) ≤ s)
end
```

```
fun max_child(a:array[1..n] of elem, s: int, i:int) ret j:int   {pre: has_children(s,i)}
    if right(i) ≤ s ∧ a[left(i)] ≤ a[right(i)] then j:= right(i)
    else j:= left(i)
    fi
end

proc lift(in/out a:array[1..n] of elem,in i:int)
     swap(a,i,father(i))
end

fun must_lift(a:array[1..n] of elem, i:int) ret b:bool
    b:= (a[i] >a[father(i)])
end

proc float(in/out a:array[1..n] of elem,in i:int)
     var c: int
     c:= i
     while has_father(c) ∧ must_lift(a,c) do
          lift(a,c)
          c:= father(c)
     od
end

proc sink(in/out a:array[1..n] of elem, in s:int, in i: int)      {se aplica a 1 ≤ i ≤ s}
     var f: int
     f:= i
     while has_children(s,f) ∧ must_lift(a,max_child(a,s,f)) do
          f:= max_child(a,s,f)
          lift(a,f)
     od
end
```

Los cambios realizados consistieron exclusivamente en adaptar las funciones al hecho de que el heap no está representado por una tupla sino que sus 2 campos están separados. Por un lado se tiene el arreglo a, que es el mismo a ordenar, y por otro el tamaño s que en el programa principal heapSort está dado por el índice i del primer **for**, o i-1 en caso del segundo **for**. Las funciones left, right, father y has_father quedan idénticas.

Por último, presentamos a continuación la versión de heapSort que resultaría si no utilizáramos funciones y procedimientos auxiliares. A los fines pedagógicos la versión anterior es más apropiada, pero la que sigue se parece más a la que suele encontrarse en libros y bibliotecas de programas. Es por eso que decidimos incluirla a continuación.

Se utilizan tres variables locales: f por father (padre), c por child (hijo) y r por right ((hijo) derecho). En el primer **for** se realiza, para cada ejecución del ciclo, la incorporación de un nuevo elemento (a[i]) al heap. El **while** que está dentro de ese primer **for** se encarga de hacer flotar el nuevo elemento. Las divisiones por 2 corresponden a la función father que utilizábamos anteriormente, la condición del **while** a la conjunción de has_father(c) con must_lift(a,c), la operación swap a la llamada al procedimiento lift.

El segundo **for** es más complicado. Luego del swap (que asigna a a[i] su valor definitivo, es decir, deja esa celda ordenada) se realiza el "hundimiento" de la posición 1. Observar que c < i exactamente cuando f tiene hijo(s), y c es i-1 exactamente cuando f tiene 1 hijo, en cuyo caso se cumplirá además r = c. Este artilugio permite tratar de manera uniforme los casos en que f tiene 1 ó 2 hijos. Mientras estemos en uno de esos casos, se compara lo alojado en f con lo alojado en el mayor de sus hijos y se intercambian el padre con el mayor de sus hijos. Nuevamente se asigna a r una expresión no trivial, con la intención de que r sea igual a c en caso de que f tenga un único hijo.

```
proc heapSort(in/out a:array[1..n] of elem)
    var f,c,r: int
    for i:= 1 to n do
        c:= i                                          {comienza enqueue(a[i])}
        f:= i ÷ 2
        while c ≠ 1 ∧ a[c] > a[f] do
            swap(a,c,f)
            c:= f
            f:= f÷2
        od                                             {termina enqueue(a[i])}
    od
    for i:= n downto 1 do
        swap(a,1,i)                                    {a[i]:= first, comienza dequeue}
        f:= 1
        c:= 2
        r:= mín(3,i-1)
        while c < i ∧ (a[f] < máx(a[c],a[r])) do
            if a[c] ≥ a[r] then swap(a,c,f)
                                f:= c
            else swap(a,r,f)
                 f:= r
            fi
            c:= 2*f
            r:= mín(2*f+1,i-1)
        od
    od
end
```

Algoritmos "Divide y Vencerás (o Reinarás) " (Divide and Conquer)

Hemos explicado que el mergeSort es un ejemplo típico de algoritmo del tipo **divide y vencerás**. Las características de estos algoritmos son: uno conoce alguna solución para los casos sencillos; para los complejos uno descubre una manera de **dividir** o **descomponer** el problema en subproblemas de menor tamaño, más aún, el tamaño de los subproblemas es una **fracción** del tamaño del problema original; y también descubre cómo **combinar** las soluciones de los subproblemas para construir una del problema original.

La forma general de los algoritmos divide y vencerás sigue el esquema

fun DC(x) **ret** y
  if x suficientemente pequeño o simple **then** y:= ad_hoc(x)
  else descomponer x en en $s_1, x_2, \ldots, x_a$
    for i:= 1 **to** a **do** y_i:= DC(x_i) **od**
    combinar $y_1, y_2, \ldots, y_a$ para obtener la solución y de x
  fi
end

Un ejemplo lo proporciona la búsqueda binaria. El caso pequeño es cuando se busca en un segmento de arreglo de longitud 0, el caso simple es cuando el elemento buscado se encuentra exactamente en la posición que se mira (med), la descomposición consiste en mirar entre izq y med o entre med y der (o sea, a=1) y la etapa de combinación es trivial.

Otro ejemplo lo da la ordenación por intercalación (mergeSort). El caso pequeño es cuando se ordena un segmento trivial de arreglo (longitud menor o igual a 1), la descomposición consiste en ordenar entre izq y med y entre med y der (a=2) y la etapa de combinación consiste en intercalar los resultados obtenidos por ordenar cada uno de los 2 fragmentos de arreglo.

Ordenación rápida (quickSort). Quizá el algoritmo más famoso de la técnica divide y vencerás lo proporciona otro algoritmo de ordenación: la ordenación rápida (quickSort). Ésta puede verse como una modificación de la ordenación por intercalación a partir de la observación de que intercalar es incómodo, ya que requiere copiar los segmentos a intercalar en espacios de memoria auxiliares. Para evitar intercalar, la idea de la ordenación rápida es descomponer el problema en 2 subproblemas de manera que no requieran intercalación. ¿Cómo? Separando el arreglo en 2 fragmentos de manera 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 **pivot**, es decir, un elemento que se utilizará para separar ambos fragmentos. Aquellos elementos que sean menores o iguales al pivot pertenecerán al primer fragmento, aquéllos que no, al segundo. Llamaremos pivot al procedimiento que realiza dicha fragmentación. El procedimiento quickSort invoca al procedimiento pivot.

proc quickSort (**in**/**out** a: **array**[1..n] **of** elem, **in** izq: **int**, **in** der: **int**)
  var piv: **int**
  if der > izq **then** pivot(a,izq,med,piv)
    {todos los elementos en a[izq,piv] son $\leq$ que a[piv]}
    {$\wedge$ todos los elementos en a(piv,der] son > que a[piv]}
    quickSort(a,izq,piv-1)
    quickSort(a,piv+1,der)
  fi
end

Como puede observarse, la variable piv no ha sido inicializada. Eso es porque su valor no se calcula en el procedimiento quickSort, sino en el procedimiento pivot que pasamos a desarrollar a continuación. Como adelantamos, este procedimiento elige un

pivot que se utiliza para clasificar a los elementos entre izq y der en dos fragmentos: el de aquéllos que son menores o iguales al pivot y el de aquéllos que son mayores a él. Arbitrariamente tomamos el primer elemento (a[izq]) como pivot.

```
proc pivot (in/out a: array[1..n] of elem, in izq: int, in der: int, out piv: int)
    var i,j: int
    piv:= izq
    i:= izq+1
    j:= der
    while i ≤ j do
                {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 pivot en una posición más central}
    piv:= j                                 {señalando la nueva posición del pivot}
end
```

Una vez elegido el pivot (a[izq] o, equivalentemente, a[piv]) se clasifican los restantes elementos según sean menores o iguales, o mayores al pivot. El índice i se utiliza para recorrer desde la posición izq+1 hacia la derecha, avanzando mientras encuentre elementos menores o iguales al pivot. El índice j se utiliza para recorrer desde la posición der hacia la izquierda, retrocediendo mientras encuentre elementos mayores al pivot. 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 pivot y todos los posteriores a j son mayores que el pivot. Para finalizar, se ubica el pivot al final del primer fragmento y se asigna a piv esa nueva posición.

El programa principal que dispara la ordenación rápida es simplemente

```
quickSort(a,1,n)
```

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 quickSort a un arreglo perfectamente ordenado. La primera vez que se ejecute 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. De esta manera 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 quickSort 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. Evidentemente, $t(n) \in \Theta(n^2)$.

Exactamente lo mismo ocurre al aplicar quickSort a un arreglo ordenado al revés. Conclusión: quickSort 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 permite confiar en el caso medio aún si se aplica a un arreglo ordenado. La más sencilla consiste en elegir (pseudo)aleatoriamente el pivot, en vez de tomar siempre el primero del segmento.

Exponenciación. Un ejemplo más sencillo de la técnica divide y vencerás que no deja de ser interesante es un algoritmo que calcula la potencia a^n para un entero a y un natural n. La versión más ingenua de dicho algoritmo es la siguiente:

```
fun expo(a: int, n: int) ret r: int                    {pre: n ≥ 0}
    r:= 1
    for i:= 1 to n do r:= r*a od
end
```

Si bien el costo de una multiplicación varía considerablemente con el tamaño de los números a multiplicar, concentremos por ahora la atención en contar el número de multiplicaciones que realiza este algoritmo en función de n sin preocuparnos por el tamaño de los números a multiplicar. Evidentemente el número de multiplicaciones es n.

¿Se puede hacer algo mejor? Sí, utilizando la técnica de divide y vencerás y observando que para n par $a^n = (a^{n/2})^2$. Esto nos permite escribir la función anterior de otra manera:

```
fun expoDC(a: int, n: int) ret r: int                    {pre: n ≥ 0}
    if n==0 then r:= 1
    else if n es par then r:= expoDC(a,n/2)
                           r:= r*r
         else r:= a*expoDC(a,n-1)
         fi
    fi
end
```

Sea $N(n)$ el número de multiplicaciones realizadas por este algoritmo para el exponente n. Si n es par, $N(n) = N(n/2) + 1$. Ésta es una recurrencia divide y vencerás con $a = 1$, $b = 2$ y $k = 0$. Por ello, $a = b^k$ y $N(n) \in \Theta(\log n)$ para n múltiplo de 2. Se puede comprobar que el mismo resultado se obtiene para los demás n's.

Multiplicación de números grandes. Como afirmamos recién, no es lo mismo multiplicar números pequeños que números grandes. Repasemos con un ejemplo el algoritmode multiplicación usual:

```
    3476
    1593
  -------
   10428
  31284
 17380
 3476
-------
5537268
```

Por cada dígito de 1593 escribimos una término a sumar. Para el cómputo de cada termino, debemos recorrer los dígitos de 3476. Sea n el número de dígitos de los números a multiplicar (asumiendo que los factores tienen aproximadamente el mismo número de dígitos). El cómputo de las n líneas insume tiempo $\Theta(n^2)$, y la suma de los n términos también. Ése es el orden del algoritmo usual de multiplicación.

¿Podemos hacer algo mejor? Intentemos aplicar las ideas de la técnica divide y vencerás. Separemos cada factor en dos mitades: el primero en 34 y 76, y el segundo en 15 y 93:

```
34 * 15  =  510
34 * 93  =  3162
76 * 15  =  1140
76 * 93  =     7068
          ---------
          5537268
```

Esta es una manera correcta de multiplicar dado que $3476 = 10^2 * 34 + 76$ y $1593 = 10^2 * 15 + 93$, luego $3476 * 1593 = 10^4 * (34 * 15) + 10^2 * (34 * 93 + 76 * 15) + 76 * 93$.

En general queremos multiplicar 2 números a y b de n dígitos decimales. Asumimos por simplicidad que n es par. El método que estamos considerando descompone a y b: $a = 10^{n/2}w + x$ y $b = 10^{n/2}y + z$. Luego, $a * b = 10^n wy + 10^{n/2}(wz + xy) + xz$. Es decir, para realizar una multiplicación entre números de longitud decimal n se realizan 4 multiplicaciones entre números de longitud decimal $n/2$, además de 3 sumas de números de

longitud decimal n y multiplicaciones por potencias de 10, que no son verdaderas multiplicaciones sino sólo desplazamientos (shifts) hacia la izquierda, o equivalentemente, el simple agregado de 0's a la derecha. Vale decir que se reduce el problema de realizar una multiplicación entre números de longitud n al de realizar 4 multiplicaciones entre números de longitud $n/2$ más ciertas operaciones (sumas, shifts) que son de orden $\Theta(n)$. Para analizar este algoritmo, consideramos la recurrencia resultante:

$$t(n) = 4t(n/2) + \Theta(n)$$

Como $4 > 2^1$, resulta que $t(n) \in \Theta(n^{\log_2 4}) = \Theta(n^2)$. Es decir, no hemos ganado nada sobre el algoritmo original, el que todos utilizamos en la práctica.

Sin embargo, éste es fácil de mejorar. Llamemos $p = wy$, $q = xz$ y $r = (w+x)(y+z)$. Con sólo 3 multiplicaciones entre números de longitud $n/2$ (o $n/2+1$ en el último caso) obtenemos p, q y r. Observemos además que r resulta igual a $r = wy + wz + xy + xz = p + wz + xy + q$. Por ello, $a * b = 10^n p + 10^{n/2}(r - p - q) + q$, es decir, 3 multiplicaciones y varias sumas y restas. La recurrencia asociada a este algoritmo es:

$$t(n) = 3t(n/2) + \Theta(n)$$

Como aún tenemos $3 > 2^1$, resulta que $t(n) \in \Theta(n^{\log_2 3})$. Esto es $t(n) \in \mathcal{O}(n^{1,6})$ y $t(n) \in \Omega(n^{1,5})$. Es decir, este algoritmo es significativamente mejor que el que utilizamos en la práctica.

Algoritmos Voraces (o Glotones, Golosos) (Greedy)

Una técnica muy sencilla de resolución de problemas (que, lamentablemente, no siempre da resultado) es la de los algoritmos voraces. Se trata de algoritmos que resuelven problemas de **optimización**. Es decir, tenemos un problema que queremos resolver de manera **óptima**: el mejor camino que une dos ciudades, el valor máximo alcanzable seleccionando ciertos productos, el costo mínimo para proveer un cierto servicio, el menor número de billetes para pagar un cierto importe, etc. Los algoritmos voraces intentan construir la solución óptima búscada **paso a paso**, **eligiendo** en cada paso la **componente** de la solución que **parece** más apropiada.

Estos algoritmos nunca revisan una **elección** ya realizada, confían en haber elegido bien las componentes anteriores. Por ello, no siempre dan resultado, existen problemas para los cuales a veces conviene realizar un paso aparentemente menos apropiado pero que a la larga resulta en una solución mejor (de la misma forma que en ajedrez puede convenir sacrificar una pieza para obtener una ventaja varias jugadas después).

Sin embargo, para ciertos problemas interesantes, los algoritmos voraces obtienen soluciones óptimas. Cuando eso ocurre, no sólo resultan algoritmos sencillos, sino que además son muy eficientes.

Forma general de los algoritmos voraces. Las características comunes de los algoritmos voraces son las siguientes:

- se tiene un problema a resolver de manera **óptima**, para ello se cuenta con un conjunto de **candidatos** a ser parte de la solución,

- a medida que el algoritmo se ejecuta, los candidatos pueden clasificarse en 3: aquéllos que áun no han sido considerados, aquéllos que han sido **incorporados** a la solución que se intenta construir, y aquéllos que han sido **descartados**,
- existe una función que chequea si los candidatos incorporados ya forman una **solución** del problema, sin preocuparse por si la misma es o no óptima,
- una segunda función que comprueba si un conjunto de candidatos es **factible** de crecer hacia una solución (nuevamente sin preocuparse por cuestiones de optimalidad),
- finalmente, una tercer función que **selecciona** de entre los candidatos aún no considerados, cuál es el más promisorio.

Los algoritmos voraces proceden de la siguiente manera: inicialmente ningún candidato ha sido considerado, es decir, ni incorporado ni descartado. En cada paso se utiliza la función de **selección** para elegir cuál candidato considerar. Se utiliza la función **factible** para evaluar si el candidato considerado se incorpora a la solución o no. Se utiliza la función **solución** para comprobar si se ha llegado a una solución o si el proceso de construcción debe continuar.

Más esquemáticamente tenemos:

```
fun greedy(C) ret S          {C: conjunto de candidatos, S: solución a construir}
    S:= {}
    while S no es solución do
        c:= seleccionar de C
        C:= C-{c}
        if S∪{c} es factible then S:= S∪{c} fi
    od
end
```

Problema de las monedas. Un ejemplo de la vida cotidiana lo proporciona el siguiente problema. Asumimos que tenemos una cantidad infinita de monedas de cada una de las siguientes denominaciones: 1 peso, 50 centavos, 25 centavos, 10 centavos, 5 centavos y 1 centavo. Se debe determinar un algoritmo que dado un importe encuentre la manera de pagar dicho importe exacto con la menor cantidad de monedas posible.

Siendo un problema tomado de la vida cotidiana, todos hemos tenido que resolverlo (quizá inconscientemente) alguna vez. La idea que utilizamos en la práctica es tomar tantas monedas de 1 peso como sea posible sin pasarnos del monto, luego agregar tantas de 50 centavos como sea posible sin pasarnos del resto del monto, luego agregar tantas de 25 centavos como sea posible, luego agregar tantas de 10 centavos como sea posible, luego de 5 centavos y finalmente de 1 centavo.[5] Más brevemente, lo que hacemos es ir utilizando sucecivamente monedas de la mayor denominación posible sin sobrepasar el monto total. Esto determina un algoritmo voraz, que puede escribirse utilizando pseudo-código de la siguiente manera. La variable s contabiliza la suma de las monedas ya incorporadas al manojo S de monedas que se utilizarán para pagar el monto.

[5]Para estas denominaciones, no precisaremos más de 1 moneda de 50 centavos, 1 de 25 centavos, 2 de 10 centavos, 1 de 5 centavos y 4 de 1 centavo. En cambio podemos precisar numerosas monedas de 1 peso, dependiendo del importe total a pagar.

```
fun cambio(n: monto) ret S: conjunto de monedas
    var c,s: monto
    C:= {100, 50, 25, 10, 5, 1}
    S:= {}
    s:= 0
    while s ≠ n do
        c:= mayor elemento de C tal que s+c ≤ n
        S:= S∪{una moneda de denominación c}
        s:= s+c
    od
end
```

Este algoritmo encuentra, para cualquier monto, la manera óptima de pagarlo, es decir, la manera de pagarlo con el menor número de monedas posibles. Pero ¿qué ocurre si cambiamos las denominaciones posibles? Supongamos que no existen monedas de 5 centavos. Suponiendo entonces que C no contiene al 5, ¿sigue obteniendo este algoritmo la solución óptima? Es fácil comprobar que al aplicar esta función a un importe de 30 centavos, obtenemos como resultado {una moneda de 25 centavos y 5 de 1 centavo} que no es la solución óptima ya que en vez de estas 6 monedas podríamos utilizar solamente 3 monedas de 10 centavos.

Supongamos un conjunto C para el que este algoritmo obtiene la solución óptima. Con C fijo, este algoritmo es lineal en el monto a pagar. Si tenemos un número m de denominaciones inicialmente ordenadas deberemos recorrer esas denominaciones manteniendo la linealidad en el monto. El algoritmo resulta lineal en $n * m$. Si las monedas no están inicialmente ordenadas habría que agregar la tarea de ordenarlas. Esto resulta en un algoritmo lineal en $n * m * \log m$. Otra alternativa sería mantener un heap con las denominaciones. Esto también resulta en el peor caso en un algoritmo lineal en $n * m * \log m$.

Problema de la mochila. Tenemos una mochila de capacidad W. Tenemos n objetos de valor $v_1, v_2, \ldots, v_n$ y peso $w_1, w_2, \ldots, w_n$. Se quiere encontrar la mejor selección de objetos para llevar en la mochila. Por mejor selección se entiende aquélla que totaliza el mayor valor posible sin que su peso exceda la capacidad W de la mochila. Para que el problema no sea trivial, asumimos que la suma de los pesos de los n objetos excede la capacidad de la mochila, obligándonos entonces a seleccionar cuáles cargar en ella.

Este problema es complejo, no se conocen soluciones eficientes para él. Para encontrar una solución voraz, simplificamos significativamente el problema de la siguiente manera. Asumiremos que los objetos pueden fraccionarse. La ventaja de esto es que permite llenar completamente la mochila sin dejar capacidad de la mochila sin aprovechar.

Con esta simplificación, podemos decir que el problema a resolver consiste en determinar números reales $s_1, s_2, \ldots, s_n$ entre 0 y 1 que indican qué fragmento de cada objeto se incluyen en la mochila. Así, $s_i = 0$ indica que el i-ésimo objeto no se incluyó en la mochila, $s_i = 1$ indica que todo el i-ésimo objeto se incluyó, $s_i = 3/4$ indica las tres cuartas partes de dicho objeto se incluyeron, etc. Es decir, el problema consiste en determinar $s_1, s_2, \ldots, s_n$ entre 0 y 1 tales que $\sum_{i=1}^{n} s_i v_i$ sea máximo y $\sum_{i=1}^{n} s_i w_i < W$.

Para encontrar una solución voraz a este problema debemos decidir con qué criterio elegir cada objeto a incorporar. Una vez determinado dicho criterio, se trata de ir incorporando uno a uno los elementos elegidos con ese criterio sin superar la capacidad de la mochila. Cuando querramos incorporar uno que ya "no cabe", lo fragmentamos e incorporamos la porción que sí cabe. Esto da lugar al siguiente algoritmo expresado en pseudo-código.

```
fun mochila(v: array[1..n] of valor, w: array[1..n] of peso, W: peso)
                                                    ret s: array[1..n] of real
    var weight: peso
        c: int
    for i:= 1 to n do s[i]:= 0 od
    weight:= 0
    while weight ≠ W do
        c:= mejor objeto remanente
        if weight + w[c] ≤ then s[c]:= 1
                                weight:= weight + w[c]
        else s[c]:= (W-weight)/w[c]
             weight:= W
        fi
    od
end
```

Observamos que la rama **then** del **if** corresponde al caso en que el objeto elegido c "cabe" en la mochila. En ese caso, weight contabiliza el peso que ya tendrá la mochila al incorporar c y el arreglo s registrará que el objeto c está en la mochila. La rama **else** corresponde al caso en que c "no cabe" entero en la mochila. En ese caso se incluirá en ella el máximo fragmento que quepa. Evidentemente el peso que tendrá la mochila al incorporar el mayor fragmento posible será el máximo que la mochila puede alcanzar: W. Eso explica la segunda asignación. La primera puede obtenerse viendo que queremos que s[c], la fracción a agregar del objeto c, debe llenar el resto de la mochila, es decir, debe satisfacer weight + s[c]*w[c] = W. Despejando s[c] obtenemos la primera asignación.

Aún queda por determinar el criterio para elegir el "mejor objeto remanente." Uno de los primeros que surgen es incorporar el objeto remanente de mayor valor. La idea no está tan mal, sin embargo puede que incorporemos objetos de mucho valor, pero que pesan demasiado privándonos de incorporar luego otros que quizá tenga un poco menos de valor pero pesen significativamente menos que los de mayor valor. Otro criterio es, incorporar el objeto remanente de menor peso. Esta idea padece de similares defectos que la anterior. Debemos combinar las ideas "mayor valor" con "menor peso". La nueva idea es asegurarse de que cada vez que agregamos un objeto a la mochila estamos "sacando el máximo jugo" posible del peso que ese objeto ocupa. Es decir, incorporamos objetos cuya relación valor/peso es la mayor posible. Por último, observemos que los objetos remanentes son aquéllos objetos i tales que s[i] es nulo. El algoritmo resultante se obtiene reemplazando la línea que dice

c:= mejor objeto remanente

por

c:= objeto tal que s[c]==0 y v[c]/w[c] es máximo

El análisis de este algoritmo es muy similar al de la moneda. Sólo hay que tener en cuenta que al ordenar (o armar un heap) con los objetos, el criterio de comparación debe ser entre las fracciones valor/peso de cada objeto.

Por último, regresemos al punto en que dijimos que para resolver el problema de la mochila con un algoritmo voraz debíamos permitir fragmentar objetos. ¿Por qué? ¿Por qué no alcanza con, en caso de que el objeto a insertar "no quepa", intentar con el siguiente, etc. hasta encontrar uno que sí "quepa" o dejar una porción de la mochila sin ocupar?

Problema del camino de costo mínimo: Algoritmo de Dijkstra. Tenemos un grafo dirigido $G = (N, A)$ con costos no negativos en las aristas y queremos encontrar el camino de costo mínimo desde un nodo hasta cada uno de los demás. Asumiremos que el grafo viene dado por el conjunto de nodos $N = \{1, 2, \ldots, n\}$ y los costos por una matriz L : **array**[1..n,1..n] **of** costo, que en $L[i, j]$ mantiene el costo de la arista que va de i a j. En caso de no haber ninguna arista de i a j, diremos que $L[i, j] = \infty$. Asumimos $L[j, j] = 0$. Si quisiéramos aplicar lo que sigue a grafos no dirigidos simplemente agregaríamos la condición $L[i, j] = L[j, i]$ para todo par de nodos i y j.

A continuación damos una versión simplificada del algoritmo. En vez de hallar el camino de costo mínimo desde un nodo hasta cada uno de los demás, halla sólo el costo de dicho camino. Es decir, halla el costo del camino de costo mínimo desde un nodo hasta cada uno de los demás.

El resultado estará dado por un arreglo D: **array**[1..n] **of** costo, en $D[j]$ devolverá el costo del camino de costo mínimo que va de i a j. Si pensamos que los costos son longitudes, en D se devolverán las distancias de i a cada uno de los demás nodos.

```
fun Dijkstra(L: array[1..n,1..n] of costo, i: int) ret D: array[1..n] of costo
    var c: int
    C:= {1,2,...,n}-{i}
    for j:= 1 to n do D[j]:= L[i,j] od
    repeat n-2 times
        c:= elemento de C que minimice D[c]
        C:= C-{c}
        for j in C do D[j]:= mín(D[j],D[c]+L[c,j]) od
    end
end
```

Llamamos **nodos especiales** a los que no pertenecen a C. Inicialmente el único nodo especial es i. Un **camino especial** es un camino cuyos nodos son especiales salvo quizá el último. Inicialmente, los caminos especiales son el camino vacío (que va de i a i y tiene costo $L[i, i] == 0$) y las aristas que van de i a j que tienen costo $L[i, j]$.

La idea del algoritmo es que en todo momento, D mantiene en cada posición j, el costo del camino **especial** de costo mínimo que va de i a j. Inicialmente, por lo dicho en el párrafo anterior, $D[j]$ debe ser $L[i, j]$. Eso explica la inicialización de D que se realiza en el primer **for**.

Veremos que cuando un nodo c es especial, ya se conoce el costo del camino de costo mínimo que va de i a c, y es el que está dado en ese momento por $D[c]$. Dijimos que inicialmente el nodo i es el único especial, efectivamente el valor inicial de $D[i]$, es decir, 0, es el costo del camino de costo mínimo para ir desde i a i.

Lo dicho en los dos últimos párrafos puede expresarse en el siguiente invariante:

$$\forall j \notin C.D[j] = \text{costo del camino de costo mínimo de } i \text{ a } j$$
$$\forall j \in C.D[j] = \text{costo del camino } \textbf{especial} \text{ de costo mínimo de } i \text{ a } j$$

Por ello, para entender el algoritmo es importante prestar atención a la palabra **especial**. Cuando conocemos el costo del camino **especial** de costo mínimo no necesariamente hemos obtenido lo que buscamos, buscamos el costo del camino de costo mínimo, el mínimo de todos, especial o no.

Como puede verse, el algoritmo de Dijkstra elimina en cada ciclo un nodo c de C. Para que se mantenga el invariante es imprescindible saber que para ese c (que pertenecía a C y por lo tanto por el invariante $D[c]$ era el costo del camino **especial** de costo mínimo de i a c), $D[c]$ es en realidad el costo del camino de costo mínimo de i a c.

¿Cómo podemos asegurarnos de eso? El algoritmo elige $c \in C$ de modo de que $D[c]$ sea el mínimo. Es decir, elige un nodo c que aún **no es especial** y tal que $D[c]$ es mínimo. Sabemos, por el invariante, que $D[c]$ es el costo del camino **especial** de costo mínimo de i a c. ¿Puede haber un camino **no especial** de i a c que cueste menos? Si lo hubiera, dicho camino necesariamente debería tener, por ser **no especial**, algún nodo intermedio **no especial**. Sea w el primer nodo **no especial** que ocurre en ese camino comenzando desde i. Evidentemente el camino **no especial** consta de una primer parte que llega a w. Esa primer parte es un camino **especial** de i a w, por lo que su costo, dice el invariante, debe ser $D[w]$. El costo del camino completo **no especial** de i a c que pasa por w costará al menos $D[w]$ ya que ése es apenas el costo de una parte del mismo. Sin embargo, como c fue elegido como el que minimiza (entre los nodos **no especiales**) $D[c]$, necesariamente debe cumplirse $D[c] \leq D[w]$. Esto demuestra que no puede haber un camino **no especial** de i a c que cueste menos que $D[c]$. Por ello, c puede sin peligro ser considerado un nodo **especial** ya que ya $D[c]$ contiene el costo del camino de costo mínimo de i a c.

Inmediatamente después de agregar c entre los nodos **especiales**, es decir, inmediatamente después de eliminarlo de C, surgen nuevos caminos **especiales** ya que ahora se permite que los mismos pasen también por el nuevo nodo **especial** c. Eso obliga a actualizar $D[j]$ para los j **no especiales** de modo de que siga satisfaciendo el invariante. Efectivamente, ahora un camino **especial** a j puede pasar por c. Sólo hace falta considerar caminos **especiales** de i a j cuyo último nodo **especial** es c. En efecto, los caminos **especiales** de i a j que pasan por c y cuyo último nodo **especial** es k no ganan nada por pasar por c ya que c está antes de k en esos caminos y entonces el costo del tramo hasta k, siendo k **especial**, sigue siendo como mínimo $D[k]$, es decir, en el mejor de los casos lo mismo que se tenía sin pasar por c.

Consideremos entonces solamente los caminos **especiales** a j que tienen a c como último nodo **especial**. El costo de un tal camino de costo mínimo está dado por $D[c] + L[c, j]$, la suma entre el costo del camino de costo mínimo para llegar hasta c ($D[c]$) más el costo de la arista que va de c a j ($L[c, j]$). Este costo debe compararse con el

que ya se tenía, el que sólo contemplaba los caminos **especiales** antes de que c fuera **especial**. Ese valor es $D[j]$. El mínimo de los dos es el nuevo valor para $D[j]$. Eso explica el segundo **for**.

Por último, puede observarse que en cada ejecución del ciclo un nuevo nodo se vuelve **especial**. Inicialmente i lo es. Por ello, al cabo de n-2 iteraciones, tenemos solamente 1 nodo **no especial**. Sea k ese nodo. El invariante resulta

$$\forall j \neq k.D[j] = \text{costo del camino de costo mínimo de } i \text{ a } j$$
$$D[k] = \text{costo del camino } \textbf{especial} \text{ de costo mínimo de } i \text{ a } k$$

pero siendo k el único nodo **no especial** todos los caminos de i a k (que no tengan ciclos en los que k esté involucrado) son **especiales**. Por ello, se tiene

$$D[k] = \text{costo del camino de costo mínimo de } i \text{ a } k$$

y por consiguiente

$$\forall j.D[j] = \text{costo del camino de costo mínimo de } i \text{ a } j$$

Árbol generador de costo mínimo (Minimum Spanning Tree). Sea $G = (N, A)$ un grafo conexo no dirigido con un costo no negativo asociado a cada arista. Se dice que $T \subseteq A$ es un árbol generador si (N, T) es conexo y no contiene ciclos. Su costo es la suma de los costos de sus aristas. Se busca T tal que su costo sea mínimo.

El problema de encontrar un árbol generador de costo mínimo tiene numerosas aplicaciones en la vida real. Cada vez que se quiera realizar un tendido (eléctrico, telefónico, etc) se quieren unir distintas localidades de modo que requiera el menor costo en instalaciones (por ejemplo, cables) posible. Se trata de realizar el tendido siguiendo la traza de un árbol generador de costo mínimo.

Propiedades. Sabemos que $|T| = |N| - 1$. En efecto,

1. Todo grafo $G = (N, A)$ conexo satisface $|A| \geq |N| - 1$.
2. Todo grafo $G = (N, A)$ sin ciclos satisface $|A| \leq |N| - 1$.

Ambas propiedades se demuestran fácilmente por inducción en $|N|$.

Podemos ver que para construir un árbol generador podemos partir del conjunto vacío de aristas e incorporar gradualmente aristas (cuidando de no introducir ciclos). Cuando hayamos juntado $|N| - 1$ aristas tendremos un árbol generador. Otra alternativa sería partir del conjunto A de aristas y quitarle gradualmente aristas (cuidando de no volverlo inconexo). Cuando sólo resten $|N| - 1$ aristas tendremos un árbol generador.

Estas propiedades puede formularse así:

1. Sea $G = (N, A)$ un grafo sin ciclos. Si $|A| = |N| - 1$ entonces G es conexo.
2. Sea $G = (N, A)$ un grafo conexo. Si $|A| = |N| - 1$ entonces G no tiene ciclos.

Veremos enseguida 2 algoritmos voraces para encontrar un árbol generador de costo mínimo. Ambos siguen el primero de estos enfoques: parten de un conjunto vacío de aristas T e incorporan gradualmente aristas sin introducir ciclos. De todas maneras no alcanza con hallar un árbol generador, se trata de hallar uno de costo mínimo.

Ambos algoritmos tienen las siguientes características en común:

candidatos: El conjunto de candidatos es el conjunto de aristas A.
solución: Se encuentra una solución cuando se tiene que $|T| = |N| - 1$.

factible: Se dice que T es factible si no tiene ciclos.
selección: Los 2 algoritmos que veremos se diferencian justamente en el criterio de selección de una nueva arista para incorporar a T
promisorio: Se dice que T es **promisorio** cuando puede extenderse con 0 ó más aristas de modo de convertirse en un árbol generador de costo mínimo.
dejar: Se dice que una arista **deja** un conjunto de nodos cuando exactamente uno de sus extremos está en ese conjunto (y por consiguiente, el otro no).

El siguiente lema determina un criterio general para poder agregar una arista a un conjunto promisorio T de modo de que el resultado siga siendo promisorio.

Lema: Sea $G = (N, A)$ conexo con costos no negativos y sea $B \subset N$. Sea $T \subseteq A$ promisorio tal que ningún $t \in T$ deja B. Sea a la arista de menor costo que deja B. Entonces, $T \cup \{a\}$ es promisorio.

Demostración: Sea U un árbol generador de costo mínimo tal que $T \subseteq U$. Si $a \in U$, evidentemente $T \cup \{a\}$ es promisorio.

Si $a \notin U$ entonces $|U \cup \{a\}| = |U| + 1 = |N|$ tiene un ciclo del que a forma parte. Como a deja B, debe haber $b \neq a$ en ese ciclo que también deja B. El costo de a es menor o igual que el de b. Sea $V = (U - \{b\}) \cup \{a\}$. El costo de V es menor o igual que el de U. Como $b \notin T$ ya que nadie en T deja B, entonces $T \cup \{a\} \subseteq V$. Luego, $T \cup \{a\}$ es promisorio.

Algoritmo de Prim. El algoritmo de Prim comienza a partir de un nodo k y arma progresivamente una red que conecta a ese nodo con los demás. Inicialmente T es vacío, el nodo k no está conectado con nadie. En cada paso se incorpora una arista a la red T, de modo de que un nuevo nodo resulte conectado a k. En la presentación en pseudo-código que se ve a continuación, C es el conjunto de nodos aún no conectados a k, cada nueva arista es la de costo mínimo que deja C. Por el lema, este criterio es apropiado para encontrar un árbol generador de costo mínimo.

```
fun Prim(G=(N,A) con costos en las aristas, k: int) ret T: conjunto de aristas
    var c: arista
    B:= N-{k}
    T:= {}
    repeat n-1 times
        c:= arista {i, j} de costo mínimo tal que i ∈ B y j ∉ B
        B:= B-{i}
        T:= T ∪ {c}
    end
end
```

Algoritmo de Kruskal. El algoritmo de Kruskal también comienza a partir de un conjunto de aristas T vacío, pero en vez de comenzar desde un nodo k, como Prim, puede incorporar aristas de cualquier zona del grafo. En realidad, en un primer vistazo no se fija dónde está la arista, sólo toma la arista de menor costo que queda por considerar. Si dicha arista puede incorporarse (según el lema visto más arriba), entonces la incorpora a T, si no la descarta. Al ir incorporando sucesivamente aristas que pueden pertenecer

a diferentes zonas del grafo, en realidad lo que ocurre puede explicarse en términos de componentes conexas como sigue.

Incialmente existen n componentes conexas, una para cada nodo ya que T es vacío. Al incorporar una nueva arista, habrá 2 componentes conexas que quedarán unidas en una sola componente conexa. El criterio para decidir si una arista se agrega o no a T es si sus extremos pertenecen a 2 componentes conexas diferentes o no. Nuevamente éste es un criterio apropiado según el lema, ya que si pertenecen a diferentes componentes conexas, evidentemente la arista deja una de esas componentes. Como las aristas son tratadas en orden creciente de sus costos, es la mínima que lo deja.

```
fun Kruskal(G=(N,A) con costos en las aristas, i: int) ret T: conjunto de aristas
    var i,j: nodo
        u,v: componente conexa
        c: arista
    C:= A
    T:= {}
    repeat n-1 times
        c:= arista {i, j} de C de costo mínimo
        C:= C-{c}
        u:= find(i)
        v:= find(j)
        if u ≠ v then T:= T ∪ {c}
                      union(u,v)
        fi
    end
end
```

La función find devuelve la componente conexa de un nodo, el procedimiento union realiza la unión de 2 componentes conexas. No es obvio cómo implementar estos algoritmos auxiliares. Hacerlo de manera eficiente requiere cierto ingenio. A continuación hacemos un paréntesis para regresar a "estructuras de datos" y explicar una manera muy eficiente de implementar estos algoritmos.

Estructuras de datos: el problema Union-Find. El problema Union-Find, es el problema de cómo mantener un conjunto finito de elementos distribuidos en distintas componentes. Las operaciones que se quieren realizar son tres:

init: inicializar diciendo que cada elemento está en una componente integrada exclusivamente por ese elemento,

find: encontrar la componente en que se encuentra un elemento determinado,

union: unir dos componentes para que pasen a formar una sola que tendrá la unión de los elementos que había en ambas componentes.

De sólo manipularse por estas tres operaciones, las componentes serán siempre disjuntas y siempre tendremos que la unión de todas ellas dará el conjunto de todos los elementos. Una componente corresponde a una clase de equivalencia donde la relación de equivalencia sería "a $\equiv$ b sii a y b pertenecen a la misma componente."

Pero ¿cómo implementar una componente? Observando la analogía con clases de equivalencia, podemos pensar que una componente estará dada por un representante de esa componente. Esto permite implementarlas a través de una tabla que indica para cada elemento cuál es el representante de (la componente de) dicho elemento.

Dado que asumimos una cantidad finita de elementos, los denotamos con números de 1 a n. La tabla que indica cuál es el representante de cada elemento será entonces un arreglo indexado por esos números:

type arreglo = **array**[1..n] **of int**

En lo que sigue, se enumeran varias formas de administrar una tabla para implementar las 3 operaciones mencionadas de manera eficiente. Es importante destacar que en la práctica se realiza sólo una operación init y numerosas operaciones find y union. La pregunta es cómo hacer estas dos últimas operaciones lo más eficiente posible.

Primer intento. En este primer intento procuramos simplemente dar alguna solución que sea fácil de entender. Las optimizaciones vendrán en los intentos posteriores. En éste, las operaciones init y union resultarán lineales y find constante.

En la posición i de la tabla diremos quién es el representante de (la componente de) i. Inicialmente cada elemento i pertenece a una componente que tiene sólo a i como elemento, luego i mismo debe ser el representante de dicha componente:

```
proc init(out rep: arreglo)
    for i:= 1 to n do rep[i]:= i od
end
```

Cuando quiero averiguar cuál es la componente de i, debo consultar la tabla. Como dijimos más arriba, una componente estará dada por un representante de esa componente. Para encontrar entonces el representante de la componente de i alcanza con consultar la tabla rep.

```
fun find(rep: arreglo, i: int) ret r: int
   r:= rep[i]
end
```

Por último, al unir dos componentes dadas por sus representantes i y j, se registra en la tabla rep que todos aquéllos elementos que estaban siendo representados por i serán de ahora en más representados por j. Podríamos haber tomado la decisión opuesta, por supuesto, sin que esto produzca ninguna diferencia relevante.

```
proc union(in/out rep: arreglo, in i,j: int)        {i ≠ j ∧ i = rep[i] ∧ j = rep[j]}
    for k:= 1 to n do
        if rep[k]==i then rep[k]:= j fi
    od
end
```

Observar que uno puede darse cuenta de si i es representante consultando la tabla. En efecto, i es representante sii $i = rep[i]$. Justamente la precondición del procedimiento union establece que dicho procedimiento se aplica sólo a i y j que son representantes (de componentes distintas).

Segundo intento. Disconformes con un procedimiento union lineal, proponemos acá una mejora que lo hace constante. Lamentablemente la mejora vuelve a la función find lineal en el peor caso. De todas formas se trata de una mejora ya que find no necesariamente será lineal, sólo se comportará así en el peor caso.

El procedimiento init queda como en el primer intento. Será conveniente definir una función auxiliar que plasma lo que enunciamos anteriormente: i es representante sii $i = rep[i]$.

```
fun is_rep(rep: arreglo, i: int) ret b: bool
    b:= (rep[i] == i)
end
```

Practicamos una mejora en el procedimiento union:

```
proc union(in/out rep: arreglo, in i,j: int)          {i ≠ j∧ is_rep(i) ∧ is_rep(j)}
     rep[i]:= j
end
```

Como vemos sólo modificamos la posición del arreglo correspondiente a i, que deja de ser representante ya que luego de la asignación $rep[i] \neq i$. Si existieran otros elementos, por ejemplo k, que eran representados por i, en la tabla se sigue cumpliendo $rep[k] = i$. Sin embargo, dado que la componente representada por i y por j se han unido, y sólo j sigue siendo representante, el representante de k debe ser ahora j. No asentamos esto en la tabla porque, justamente, requeriría recorrer toda la tabla buscando tales k desembocando nuevamente en un algoritmo lineal.

Como consecuencia de nuestra decisión de optimizar el procedimiento union, la tabla rep no dice necesariamente en la posición i quién es el representante de i, dice quién fue alguna vez el representante de i. Si dice que el representante de i fue k, y dice que el de k fue j, y resulta que is_rep(j) es verdadero, entonces el representante de i ahora es j. Como vemos, al buscar el representante de un elemento puede ser necesario recorrer varias posiciones de la tabla hasta encontrar el actual representante.

```
fun find(rep: arreglo, i: int) ret r: int
    var j: int
    j:= i;
    while ¬ is_rep(j) do j:= rep[j] od
    r:= j
end
```

En realidad, ahora la tabla puede verse como una foresta, es decir, un conjunto de árboles donde cada elemento es un nodo de que indica quién es su padre. Cada representante es la raíz de uno de los árboles. La función find recorre una rama desde i hasta la raíz de su árbol. Hay una correspondencia entre los árboles y las componentes.

Tercer intento. En este tercer intento, se procura acortar los caminos que debe recorrer la función find para ir desde un elemento hasta la raíz del árbol correspondiente a dicha componente. Una idea muy sencilla es que dicha función, una vez encontrado el representante, vuelva a recorrer el camino actualizando ahora la tabla para que quede registrado quién es el actual representante de cada uno de los elementos de dicho camino.

Así, la siguiente vez que se requiera averiguar el representante del mismo elemento (o de cualquiera de los que estaban en el camino actualizado) se evitará repetir su recorrido.

Como la función find tiene un "efecto secundario", dado que modifica el arreglo rep, señalamos esto agregando la notación **in/out** a dicho parámetro. Hacemos esto de manera excepcional, una verdadera función no debería modificar un parámetro.

```
fun find(in/out rep: arreglo, i: int) ret r: int
    var j,k: int
    j:= i
    while ¬ is_rep(j) do j:= rep[j] od
    r:= j
    j:= i
    while ¬ is_rep(j) do
        k:= rep[j]
        rep[j]:= r
        j:= k
    od
end
```

Los demás algoritmos no requieren modificación alguna.

Cuarto intento. Además de acortar los caminos a recorrer por la función find, se procurará ahora evitar que el procedimiento union permita su crecimiento. Para ello, se puede utilizar una tabla auxiliar que dirá cuántos elementos tiene cada componente. Por ejemplo, el procedimiento init debe indicar que cada componente tiene exactamente un elemento.

```
proc init(out rep: arreglo, out nr: arreglo)
    for i:= 1 to n do
        rep[i]:= i
        nr[i]:= 1
    od
end
```

Esta información es utilizada en el procedimiento union para decidir cuál de los dos representantes, i y j, sigue siéndolo y cuál no. Supongamos que i represente a más elementos que j. En ese caso es más probable encontrar caminos largos en el árbol cuya raíz es i que en el árbol cuya raíz es j. Si i pasara ahora a ser representado por j, dichos caminos que ya eran largos pasarían a ser más largos todavía, ya que se les agrega un paso más para llegar hasta j. Por ello, i debe pasar a ser representado por j sólo cuando i represente a lo sumo el mismo número de elementos que j:

```
proc union(in/out rep, nr: arreglo, in i,j: int)        {i ≠ j∧ is_rep(i) ∧ is_rep(j)}
    if nr[i] ≤ nr[j] then nr[j]:= nr[i]+nr[j]
                          rep[i]:= j
    else nr[i]:= nr[i]+nr[j]
         rep[j]:= i
    fi
end
```

Como se ve, cuando i represente más elementos que j es j quien pasa a ser representado por i. El procedimiento union además registra el número de elementos que pasa a tener la componente que acaba de obtenerse como unión de las dos dadas.

Las funciones is_rep y find quedan intactas.

Con estas mejoras, en la práctica los algoritmos find y union se comportan casi como si fueran constantes. Queda sin embargo una última mejora tendiente a eliminar la utilización del arreglo nr recién introducido.

Quinto intento. Se puede observar que del arreglo nr, solamente se utilizan los datos en el procedimiento union. Por ello, sólo importan los valores que dicho arreglo tiene en las posiciones correspondientes a representantes. Eso sugiere la posibilidad de señalar en el mismo arreglo rep el dato que hasta recién alojabamos en nr. Para ello, en vez de convenir que i será representante cuando $i = rep[i]$, en esos caso alojaremos en rep[i] un valor que señale el número de elementos representados por i. Observemos que si i representa k elementos no podemos simplemente poner $rep[i] = k$ ya que parecerá que i está siendo representado por k. Pero podemos aprovechar que hasta ahora sólo hay valores positivos en la tabla rep, para poner $rep[i] = -k$. De esta manera, no hay forma de equivocarse: si $rep[i] < 0$ es porque i es un representante y en ese caso el número de elementos representados por i es $-rep[i]$.

Por ejemplo, el procedimiento init debe establecer que todos son representantes de componentes que tienen un elemento cada una.

```
proc init(out rep: arreglo)
    for i:= 1 to n do rep[i]:= -1 od
    end
```

La función is_rep averigua si i es representante o no. Como dijimos, lo será sii en su lugar de la tabla se encuentra un número negativo (cuyo valor absoluto dice cuántos elementos i representa).

```
fun is_rep(rep: arreglo, i: int) ret b: bool
   b:= (rep[i] <0)
end
```

La función find queda intacta. El procedimiento union es similar salvo que las dos modificaciones se hacen sobre el mismo arreglo rep. Observar que el $\leq$ de la condición del **if** cambió por un $\geq$. Eso se debe a que en vez de comparar números positivos ahora estamos comparando números negativos. El menor de ellos corresponde al de mayor valor absoluto, por ello, al que representa más elementos.

```
proc union(in/out rep: arreglo, in i,j: int)        {i ≠ j∧ is_rep(i) ∧ is_rep(j)}
    if rep[i] ≥ rep[j] then rep[j]:= rep[i]+rep[j]
                            rep[i]:= j
    else rep[i]:= rep[i]+rep[j]
         rep[j]:= i
    fi
end
```

Programación Dinámica

Existen numerosos problemas que se resuelven con relativa facilidad a través de un algoritmo definido recursivamente, pero cuya solución tiene un comportamiento exponencial o difícil de prever. La técnica de **programación dinámica** permitirá dar una versión iterativa de dichos algoritmos mediante la construcción de una tabla que mantiene los valores que calcula el algoritmo. La ventaja de mantener dicha tabla es que evita que el algoritmo necesite ejecutarse varias veces para los mismos valores. Un ejemplo de esto lo proporciona la definición usual de la serie de Fibonacci.

Fibonacci. Recordemos la definición de la serie Fibonacci:

$$f_n = \begin{cases} n & n \leq 1 \\ f_{n-1} + f_{n-2} & n > 1 \end{cases}$$

Interpretando estas ecuaciones como una definición recursiva de los términos de la serie y dado $n \in \mathbb{N}$ ¿Cuántas veces se calcula f_1 para calcular f_n? Set $t(n)$ =número de veces que se ccalcula f_1 para calcular f_n, puede verse fácilmente que tenemos

$$t(n) = \begin{cases} n & n \leq 1 \\ t(n-1) + t(n-2) & n > 1 \end{cases}$$

Efectivamente, para calcular f_0 no se calcula f_1 ni siquiera una vez, para calcular f_1 se calcula exactamente una vez. Para calcular f_n, f_1 se calcula tantas veces como haga falta para calcular f_{n-1} $(t(n-1))$ más tantas veces como haga falta para calcular f_{n-2} $(t(n-2))$.

Es decir, que $t(n) = f_n$. Por lo visto en la página 18, $t(n)$ es exponencial. De ello se deduce que el comportamiento de la definición recursiva del n-ésimo término de la serie de Fibonacci presentada es exponencial.

¿Se puede hacer algo mejor? Una manera que ya conocemos es utilizando directamente la expresión que se obtuvo al resolver la recurrencia en la página 18. Para ilustrar la técnica de programación dinámica, presentamos a continuación una solución lineal que se basa en una idea mucho más simple: la construcción de una tabla con los valores ya calculados y la formulación de una solución iterativa.

En esta solución, f será un arreglo de tipo **array**[0..n] **of int** donde se almacenarán los resultados antes denotados f_0, f_1, ..., f_n. Como adelantamos, dichos valores se calculan desde 0 hasta n, en ese orden.

```
fun fib(n: int) ret r: int
    var f: array[0..max(n,1)] of int
    f[0]:= 0
    f[1]:= 1
    for i:= 2 to n do f[i]:= f[i-1] + f[i-2] od
    r:= f[n]
end
```

Hemos construido una definición iterativa lineal a partir de una recursiva exponencial. También suele decirse que la definición recursiva sigue un diseño "top-down" ya que se define la función sobre un argumento cualquiera descomponiendo en problemas menores, en este ejemplo, al aplicarla a argumentos menores. La definición iterativa sigue un

diseño "bottom-up" porque se calculan los valores desde los más sencillos hasta los más complejos.

Problema de la moneda. Vimos en la página 60 que el algoritmo voraz para el problema de la moneda no siempre encuentra la solución óptima, depende de cuál sean las denominaciones de las monedas. Utilizando programación dinámica es posible encontrar una solución más general, que dé la solución óptima cualquiera sean las denominaciones.

Sean $d_1, d_2, \ldots, d_n$ las denominaciones de las monedas (todas mayores que 0), asumimos que se disponen de suficientes monedas de cada denominación. Sea k el monto a pagar. Queremos calcular el menor número de monedas necesarias para pagar de manera exacta el monto k usando monedas con las denominaciones mencionadas.[6]

Para aplicar la técnica de programación dinámica, intentamos hallar primero una definición recursiva. Para $0 \leq i \leq n$ y $0 \leq j \leq k$ sea $m(i,j)$ el menor número de monedas necesarias para pagar de manera exacta el monto j usando **a lo sumo** monedas de denominación $d_1, d_2, \ldots, d_i$. Por convención, si para un cierto i y un cierto j no hay manera de pagar exactamete el monto j con monedas de denominación $d_1, d_2, \ldots, d_i$, diremos que $m(i,j) = \infty$. También abusaremos de la notación asumiendo que $1 + \infty = \infty$. Recordemos que ∞ es el elemento neutro de la operación mínimo. Se puede ver que se cumple:

$$m(i,j) = \begin{cases} 0 & j = 0 \\ \infty & j > 0 \wedge i = 0 \\ m(i-1,j) & d_i > j > 0 \wedge i > 0 \\ \text{mín}(m(i-1,j), 1 + m(i, j-d_i)) & j \geq d_i > 0 \wedge i > 0 \end{cases}$$

En efecto, cuando $j = 0$ la manera de pagar de manera exacta el monto 0 es sin utilizar ninguna moneda, es decir, 0 monedas. Cuando $j \neq 0$ y $i = 0$, no disponemos de ninguna moneda y sin embargo debemos pagar de manera exacta un monto $j > 0$. Esto es imposible, por ello el valor indicado es, como habíamos convenido, ∞. En la tercer ecuación disponemos de monedas de denominación $d_1, d_2, \ldots, d_i$ para pagar el monto j. Al ser $d_i > j$ no podemos utilizar monedas de denominación d_i. Por ello, es como si no dispusiéramos de esas monedas. El menor número de monedas necesarias para pagar de manera exacta el monto j con monedas de todas esas denominaciones será entonces igual al menor número de monedas necesarias para pagar de manera exacta dicho monto con monedas de denominación $d_1, d_2, \ldots, d_{i-1}$, que se denota $m(i-1,j)$. Por último, en la cuarta ecuación al ser $d_i \leq j$ sí se pueden utilizar monedas de denominación d_i. Pero ¿conviene usarlas?. Se evalúan las dos posibilidades: si no las usamos obtenemos, como recién, $m(i-1,j)$. Si las usamos analicemos lo que ocurre al usar una de ellas: deberemos contabilizar que ya estamos utilizando 1 moneda y deberemos contar además el menor número de monedas que hará falta para pagar de manera exacta el saldo, que será $j - d_i$. Es decir, si las usamos, el menor número de monedas que necesitaremos será $1 + m(i, j-d_i)$. Volvemos a la pregunta: ¿conviene usarlas? Depende de cuál de las dos posibilidades dá un número menor. Eso explica $\text{mín}(m(i-1,j), 1 + m(i, j-d_i))$.

[6] En realidad se quiere conocer además cuáles son las monedas a elegir. Sin embargo, nos ocuparemos primero sólo por averiguar el número de monedas y luego veremos cómo utilizar la información calculada para determinar también cuáles son las monedas a elegir.

Nuevamente tenemos una definición recursiva de un algoritmo para resolver el problema de la moneda que contiene una ecuación, como en el caso de Fibonacci, con 2 llamadas recursivas. Esto puede volver al algoritmo exponencial. Para resolver esto utilizamos la técnica de programación dinámica representando m por una tabla, en este caso una matriz:

```
fun cambio(d:array[1..n] of int, k: int) ret r: int
    var m: array[0..n,0..k)] of int
    for i:= 0 to n do m[i,0]:= 0 od
    for j:= 1 to k do m[0,j]:= ∞ od
    for i:= 1 to n do
        for j:= 1 to k do
            if d[i] >j then m[i,j]:= m[i-1,j] else m[i,j]:= min(m[i-1,j],1+m[i,j-d[i]]) fi
        od
    od
    r:= m[n,k]
end
```

La solución así obtenida tiene orden $n * k$ ya que realiza un número constante de comparaciones, sumas y asignaciones por cada celda de la matriz.

A continuación presentamos una versión del algoritmo que devuelve un arreglo nr que contiene en la posición 0 el número total de monedas necesarias y en la posición i, para $1 \leq i \leq n$ el número de monedas de denominación d_i necesarias.

```
fun cambio(d:array[1..n] of int, k: int) ret nr: array[0..n] of int
    var m: array[0..n,0..k)] of int
        r,s: int
    for i:= 0 to n do m[i,0]:= 0 od
    for j:= 1 to k do m[0,j]:= ∞ od
    for i:= 1 to n do
        for j:= 1 to k do
            if d[i] >j then m[i,j]:= m[i-1,j] else m[i,j]:= min(m[i-1,j],1+m[i,j-d[i]]) fi
        od
    od
    for i:= 0 to n do nr[i]:= 0 od
    nr[0]:= m[n,k]
    if m[n,k] ≠ ∞ then
      r:= n
      s:= k
      while m[r,s] > 0 do
          if m[r,s] = m[r-1,s] then r:= r-1
          else nr[r]:= nr[r]+1
               s:= s-d[r]
          fi
      od
    fi
end
```

Para calcular los valores del arreglo nr, es decir, para averiguar cuáles son las monedas necesarias para pagar el importe k de manera óptima, este algoritmo rastrea el origen del número que se encuentra en la celda [n,k] de la matriz. Si es igual al que se encuentra en la celda [n-1,k] significa que no es necesario utilizar monedas de denominación d_n. Para saber cuáles son las monedas, entonces, correspondería rastrear el origen del número que se encuentra en la celda [n-1,k]. Si en cambio el contenido de la celda [n,k] es diferente al número que se encuentra en la celda [n-1,k], es necesario usar al menos una moneda de denominación d_n para pagar k de manera óptima. Para saber cuáles son las demás monedas continuamos rastreando el origen del número que se encuentra en la celda [n,k-d[n]]. Se termina cuando nos toca rastrear una celda que tiene valor 0.

Problema de la mochila. En la página 61 vimos cómo solucionar una versión simplificada del problema de la mochila mediante un algoritmo voraz. La simplificación consistía en permitir fragmentar objetos. Con esta simplificación el problema se resolvía encontrando un criterio para elegir el mejor objeto a introducir en la mochila, fragmentandolo si se excedía la capacidad.

Ahora vemos el problema de la mochila sin esa simplificación. Sea W la capacidad de la mochila y sean $v_1, v_2, \ldots, v_n$ y $w_1, w_2, \ldots, w_n$ los valores y los pesos de los n objetos disponibles, todos ellos números positivos. Se debe encontrar el máximo valor alcanzable al seleccionar los objetos a cargar en la mochila, sin exceder su capacidad.[7]

Para $0 \leq i \leq n$ y $0 \leq j \leq W$ definimos $m(i,j)$ como el máximo valor alcanzable sin superar el peso j seleccionando entre los primeros i objetos. Se puede ver que se cumple:

$$m(i,j) = \begin{cases} 0 & j = 0 \\ 0 & j > 0 \wedge i = 0 \\ m(i-1,j) & w_i > j > 0 \wedge i > 0 \\ \text{máx}(m(i-1,j), v_i + m(i-1, j-w_i)) & j \geq w_i > 0 \wedge i > 0 \end{cases}$$

Como en el caso del problema de la moneda esto da lugar al siguiente algoritmo iterativo:

```
fun mochila(v:array[1..n] of valor, w:array[1..n] of int, W: int) ret r: valor
    var m: array[0..n,0..W)] of valor
    for i:= 0 to n do m[i,0]:= 0 od
    for j:= 1 to W do m[0,j]:= 0 od
    for i:= 1 to n do
        for j:= 1 to W do
            if w[i] >j then m[i,j]:= m[i-1,j] else m[i,j]:= max(m[i-1,j],v[i]+m[i-1,j-w[i]]) fi
        od
    od
    r:= m[n,W]
end
```

Si además queremos informar cuáles son los objetos a seleccionar, rastreamos el origen del contenido de la celda [n,W].

[7] Como en el caso del problema de la moneda, posponemos el problema de informar cuáles son los objetos a seleccionar para cargar la mochila de manera óptima.

```
fun mochila(v:array[1..n] of valor, w:array[1..n] of int, W: int)
                                                  ret nr: array[1..n] of bool
    var m: array[0..n,0..W)] of valor
        r,s: int
    for i:= 0 to n do m[i,0]:= 0 od
    for j:= 1 to W do m[0,j]:= 0 od
    for i:= 1 to n do
       for j:= 1 to W do
          if w[i] >j then m[i,j]:= m[i-1,j] else m[i,j]:= max(m[i-1,j],v[i]+m[i-1,j-w[i]]) fi
       od
    od
    r:= n
    s:= W
    while m[r,s] > 0 do
        if m[r,s] = m[r-1,s] then nr[r]:= false
        else nr[r]:= true
             s:= s-w[r]
        fi
        r:= r-1
    od
end
```

Problema del camino de costo mínimo: Algoritmo de Floyd. El algoritmo de Dijkstra (página 63) para obtener, en un grafo dirigido, (el costo de) un camino de costo mínimo desde un nodo a cada uno de los demás era otro ejemplo fue otro de los ejemplos vistos del uso de la técnica voraz. A continuación veremos un algoritmo que calcula (el costo de) los caminos de costo mínimo para todo nodo origen y nodo destino. Este algoritmo se basa en la técnica de programación dinámica.

Al igual que en la página 63, tenemos un grafo dirigido $G = (N, A)$ con costos no negativos en las aristas. Asumimos que el grafo viene dado por el conjunto de nodos $N = \{1, 2, \ldots, n\}$ y los costos por una matriz L : **array**[1..n,1..n] **of** costo, que en $L[i, j]$ mantiene el costo de la arista que va de i a j. En caso de no haber ninguna arista de i a j, diremos que $L[i, j] = \infty$. Asumimos $L[j, j] = 0$. Si quisiéramos aplicar lo que sigue a grafos no dirigidos simplemente agregaríamos la condición $L[i, j] = L[j, i]$ para todo par de nodos i y j.

Sea $m_k(i, j)$ el costo del camino de costo mínimo que va de i a j pasando a lo sumo por los nodos intermedios $1, 2, \ldots, k$. Se puede ver que se cumple:

$$m_k(i,j) = \begin{cases} L[i,j] & k = 0 \\ \text{mín}(m_{k-1}(i,j), m_{k-1}(i,k) + m_{k-1}(k,j)) & k > 0 \end{cases}$$

En efecto, si $k = 0$, $m_k(i, j)$ debe ser el costo del camino de costo mínimo que va de i a j sin pasar por ningún nodo intermedio. El único camino estaría conformado por la arista que va de i a j cuyo costo está dado por $L[i, j]$. Si $k > 0$ entonces se busca el costo del camino de costo mínimo que va de i a j pasando a lo sumo por los nodos $1, 2, \ldots, k$. Como $k > 0$ esta lista de nodos no es vacía: al menos k está en dicha lista.

Por ello puede que el camino de costo mínimo pase por k. Pero ¿conviene que pase por k? Analizamos las dos posibilidades: si no pasa por k, el costo del camino de menor costo que pasa a lo sumo por $1, 2, \ldots, k$ pero no pasa por k es el costo del camino de menor costo que pasa a lo sumo por $1, 2, \ldots, k-1$. Dicho costo está dado por $m_{k-1}(i, j)$. La otra posibilidad es que sí pase por k. En este caso el camino de costo mínimo de i a j está formado por el camino de costo mínimo de i a k seguido por el camino de costo mínimo de k a j. Ninguno de estos tramos pasará por k, dado que en ese caso tendríamos un ciclo y no se trataría de un camino de costo mínimo. Por ello, el costo de los tramos está dado por $m_{k-1}(i, k)$ y $m_{k-1}(k, j)$ respectivamente y el costo del camino entero por la suma de dichos números. Recordemos la pregunta: ¿conviene pasar por k? Depende de cuál de los dos casos da un valor menor. Por ello se toma el mínimo entre $m_{k-1}(i, j)$ y $m_{k-1}(i, k) + m_{k-1}(k, j)$.

De interpretarse recursivamente, la segunda ecuación representa un caso con 3 llamadas recursivas. Como vimos en los ejemplos anteriores, se puede transformar en una versión iterativa:

```
fun Floyd(L:array[1..n,1..n] of costo) ret r: array[1..n,1..n] of costo
    var m: array[1..n,1..n,1..n)] of costo
    copiar L a m[0,*,*]
    for k:= 1 to n do
        for i:= 1 to n do
            for j:= 1 to n do
                m[k,i,j]:= min(m[k-1,i,j],m[k-1,i,k]+m[k-1,k,j])
            od
        od
    od
    copiar m[n,*,*] a r
end
```

Hemos utilizado un arreglo tridimensional m pensando en que m[k,i,j] es la repreentación de $m_k(i, j)$. Veremos a continuación que es suficiente con un arreglo bidimensional, es decir, una matriz.

Observemos primero que para cada k, podemos ver a m[k,*,*] como una matriz. El algoritmo presentado calcula sucesivamente m[1,*,*], m[2,*,*], ..., m[n,*,*]. Para calcular m[k,i,j] lo único que necesita es el valor que había en la misma celda de la matriz anterior (m[k-1,i,j]) y los valores de la columna k y de la fila k de la matriz anterior (m[k-1,*,k] y m[k-1,k,*]). Es decir, que para calcular m[k,i,j] necesitamos el valor enterior en dicha celda y los valores anteriores en las columna y fila k. Esto indica que sería posible utilizar una sola matriz (de dimensión 2) y actualizar cada celda. El único problema es ¿qué ocurre si, por el orden de evaluación, se actualizan algunas celdas de la columna k o fila k antes de actualizarse la posición [i,j]? Esto en realidad no es un problema ya que por más que se actualicen las celdas de la columna k o de la fila k, sus valores no cambian. Efectivamente, podemos ver que la asignación m[k,i,j]:= min(m[k-1,i,j],m[k-1,i,k]+m[k-1,k,j]) para las celdas de la fila k se vuelve m[k,k,j]:= min(m[k-1,k,j],m[k-1,k,k]+m[k-1,k,j]) que es equivalente (ya que m[*,k,k] = 0) a m[k,k,j]:= min(m[k-1,k,j],m[k-1,k,j]) lo que evidentemente dice que la posición [k,j] de la matriz m[k,*,*] queda igual a la de

la matriz anterior. Lo mismo se observa para la columna k. Por ello, el algoritmo puede transformarse en uno que sólo utiliza un arreglo bidimensional m:

```
fun Floyd(L:array[1..n,1..n] of costo) ret m: array[1..n,1..n] of costo
    copiar L a m
    for k:= 1 to n do
        for i:= 1 to n do
            for j:= 1 to n do
                m[i,j]:= min(m[i,j],m[i,k]+m[k,j])
            od
        od
    od
end
```

Si además de obtener la información sobre el costo del camino se quieren recuperar los caminos mismos, conviene agregar una matriz interm que registrará en la posición [i,j] un nodo intermedio del camino óptimo para ir de i a j:

```
fun Floyd(L:array[1..n,1..n] of costo) ret interm: array[1..n,1..n] of int
    var m: array[1..n,1..n] of costo
    inicializar interm con todas las celdas en 0
    copiar L a m
    for k:= 1 to n do
        for i:= 1 to n do
            for j:= 1 to n do
                if m[i,j] > m[i,k]+m[k,j] then m[i,j]:= m[i,k]+m[k,j]
                                               interm[i,j]:= k
                fi
            od
        od
    od
end
```

Funciones con memoria. Las definiciones recursivas que hemos planteado en los ejemplos anteriores pueden tener la desventaja de comportarse en forma exponencial. Las versiones iterativas correspondientes que se basaron en la idea de programación dinámica eliminan este riesgo planteando un algoritmo muy predecible en cuanto a comportamiento: se limitan a recorrer todas las celdas de un arreglo y realizar para cada una de ellas una operación de comlejidad predecible. De todas formas, puede que cuando se plantea un algoritmo que computa toda una tabla, se lleguen a calcular posiciones de la tabla cuyos valores son innecesarios.

Para evitar esto presentamos a continuación una técnica mixta (entre recursiva y con tablas) que pretende quedarse con lo mejor de ambos mundos: no calcular valores más de una vez y no calcular valores innecesarios. La idea es dar una definición recursiva que mantiene además una tabla con los valores ya calculados.

Por ejemplo, para el problema de la moneda podríamos hacer lo siguiente. Definimos **type** memoria = **array**[0..n,0..k] **of int** y damos la siguiente función recursiva que asume que la memoria está inicializada con -1 en cada celda.

```
fun cambio_mem(d:array[1..n] of int, in/out m: memoria, i,j: int) ret r: int
    var v,w: int
    if m[i,j] = -1 then
      if j = 0 then m[i,j]:= 0
      else if i = 0 then m[i,j]:= ∞
           else if j <d[i] then v:= cambio_mem(d,m,i-1,j)
                                m[i,j]:= v
                else v:= cambio_mem(d,m,i-1,j)
                     w:= cambio_mem(d,m,i,j-d[i])
                     m[i,j]:= min(v,1+w)
                fi
           fi
      fi
    fi
    r:= m[i,j]
end
```

La función principal inicializa la memoria y llama a cambio_mem:

```
fun cambio(d:array[1..n] of int, k: int) ret r: int
    var m: memoria
    for i:= 0 to n do
       for j:= 0 to k do
          m[i,j]:= -1
       od
    od
    r:= cambio_mem(d,m,n,k)
end
```

Como puede verse, no está claro cuál es la ganancia de proceder de esta manera: de todas formas estamos recorriendo laçtoda la matriz para inicializarla.

Para resolver esto utilizaremos lo que se llama inicialización virtual.

Inicialización virtual. Definiremos el tipo memoria de otra manera. Primero observemos cómo quisiéramos poder utilizarla:

- inicializarlo: queremos inicializar la memoria de manera eficiente.
- calculado: queremos poder consultar si una posición ya fue calculada.
- asignar: cuando calculamos un valor aún no asignado a la celda de memoria correspondiente, queremos poder asignarlo a dicha celda.
- valor: cuando a una celda se le ha asignado un valor, queremos poder consultarlo.

Con estas primitivas, podríamos reescribir la solución al problema de la moneda utilizando una notación un poco más abstracta para la inicialización, asignación y las funciones calculado y valor:

```
fun cambio_mem(d:array[1..n] of int, in/out m: memoria, i,j: int) ret r: int
    var v,w: int
    if ¬calculado(m,i,j) then
      if j = 0 then asignar(m,i,j,0)
      else if i = 0 then asignar(m,i,j,∞)
           else if j <d[i] then v:= cambio_mem(d,m,i-1,j)
                                asignar(m,i,j,v)
                else v:= cambio_mem(d,m,i-1,j)
                     w:= cambio_mem(d,m,i,j-d[i])
                     asignar(m,i,j,min(v,1+w))
                fi
           fi
      fi
    fi
    r:= valor(m,i,j)
end
fun cambio(d:array[1..n] of int, k: int) ret r: int
    var m: memoria
    inicializar(m)
    r:= cambio_mem(d,m,n,k)
end
```

Resta diseñar una estructura de datos capaz de manejar estas 4 operaciones en tiempo constante. Inicialización virtual proporciona una tal estructura a costa de una mayor utilización de espacio. Para ilustrar como funciona, supongamos que en vez de una memoria bidimensional, como en el caso de la mochila, quisieramos una memoria unidimensional. En ese caso, definiríamos

```
type memoria = tuple
                 mem: array[1..N] of elem
                 ord: array[1..N] of int
                 sec: array[1..N] of int
                 cont: int
               end
```

El campo mem se utiliza para almacenar los elementos a cada celda, el campo sec registra en cada celda i la posición de mem que fue la i-ésima en inicializarse, el campo ord es el inverso de sec, el campo cont dice cuántas celdas de mem han sido inicializadas. Las cuatro operaciones se definirían como sigue:

```
proc inicializar(out m: memoria)
    m.cont:= 0
end
fun calculado(m: memoria, i: int) ret b: bool
    b:= (1 ≤ m.ord[i] ≤ m.cont ∧ i=m.sec[m.ord[i]])
end
```

```
proc asignar(in/out m: memoria, i: int, e: elem)                {¬calculado(m,i)}
    m.mem[i]:= e
    m.cont:= m.cont+1
    m.ord[i]:= m.cont
    m.sec[m.cont]:= i
end
fun valor(m: memoria, i: int) ret e: elem                       {¬calculado(m,i)}
   e:= m.mem[i]
end
```

En el caso que nos ocupa para el problema de la moneda, se podría proceder de una de las dos siguientes maneras: o bien se hace aritmética con las posiciones de modo de transformar posiciones que señalan celdas de una matriz en posiciones que señalan celdas de un vector que representa a la matriz, o bien se modifica la representación de la memoria de modo de que mem sea una matriz en vez de un vector. En el primer caso tendríamos $N = (n+1) * (k+1)$ y $f(i,j) = i * (k+1) + j$. El procedimiento inicializar queda como antes. Los algoritmos calculado, asignar y valor necesitan la función f para transformar posiciones matriciales en posiciones vectoriales.

```
fun calculado(m: memoria, i,j: int) ret b: bool
   var x: int
   x:= f(i,j)
   b:= (1 ≤ m.ord[x] ≤ m.cont ∧ x=m.sec[m.ord[x]])
end
proc asignar(in/out m: memoria, i,j: int, e: elem)              {¬calculado(m,i)}
    var x: int
    x:= f(i,j)
    m.mem[x]:= e
    m.cont:= m.cont+1
    m.ord[x]:= m.cont
    m.sec[m.cont]:= x
end
fun valor(m: memoria, i,j: int) ret e: elem                     {¬calculado(m,i)}
   var x: int
   x:= f(i,j)
   e:= m.mem[x]
end
```

Otra posibilidad es reemplazar la definición de memoria por la siguiente:

```
type memoria = tuple
                   mem: array[0..n,0..k] of elem
                   ord: array[0..n,0..k] of int
                   sec_l: array[1..(n+1)*(k+1)] of int
                   sec_c: array[1..(n+1)*(k+1)] of int
                   cont: int
               end
```

El procedimiento inicializar queda sin cambios.

```
fun calculado(m: memoria, i,j: int) ret b: bool
    b:= (1 ≤ m.ord[i,j] ≤ m.cont ∧ i=m.sec_l[m.ord[i,j]] ∧ j=m.sec_c[m.ord[i,j]])
end
proc asignar(in/out m: memoria, i,j: int, e: elem)                {¬calculado(m,i)}
      m.mem[i,j]:= e
      m.cont:= m.cont+1
      m.ord[i,j]:= m.cont
      m.sec_l[m.cont]:= i
      m.sec_c[m.cont]:= j
end
fun valor(m: memoria, i,j: int) ret e: elem                       {¬calculado(m,i)}
    e:= m.mem[i,j]
end
```

Recorrida de grafos y backtracking

Los algoritmos voraces construyen una solución a un problema dado a través de una secuencia incremental de soluciones parciales: paso a paso, se selecciona una nueva componente de la solución. Lo seleccionado en cada paso jamás vuelve a revisarse. Por ello, el éxito del algoritmo depende del criterio de selección.

Nos encaminamos ahora a una estrategia de construcción incremental de una solución para cuando no disponemos de un buen criterio de selección. Eso obliga a tener que revisar selecciones realizadas por comprobarse más adelante que no llevan a la solución. La técnica que veremos se llama backtracking (en castellano, "retroceso" o "volver sobre sus pasos," en referencia precisamente a retroceder para revisar selecciones anteriores.

A pesar de aplicarse a problemas de diferente naturaleza (no necesariamente de grafos) backtracking puede describirse como un algoritmo que recorre un cierto grafo, muchas veces implícito en el problema. Por ello, comenzamos esta sección con algunos algoritmos de recorrida de grafos: árboles binarios, árboles finitarios, grafos dirigidos y no dirigidos.

Recorriendo árboles binarios. Recorrer un grafo significa dar un algoritmo para visitar, o procesar, todos los nodos de un grafo. Nos interesan por ahora sólo las distintas estrategias para recorrer grafos, por ello nos conformamos con que visitar o procesar sea una actividad simbólica.

En el caso de árboles binarios, y dada la definición que hemos dado de ellos en la página 43, naturalmente pensamos en un algoritmo que se aplicará o bien a un árbol vacío, en cuyo caso no hay ningún nodo por visitar, o bien en un nodo que tiene un elemento, un subárbol izquierdo y un subárbol derecho. Recorrer este árbol consiste en

- visitar este mismo nodo, por ejemplo, procesando el elemento que se encuentra en él,
- visitar los nodos del subárbol izquierdo, es decir, recorrer el subárbol izquierdo, y
- recorrer el subárbol derecho

Realizadas estas tres acciones el árbol en cuestión habrá sido recorrido. Se desprenden naturalmente 3 estrategias para recorrerlo:

pre-orden: Primero se visita el nodo, y luego se recorren los subárboles izquierdo y derecho.

in-orden: Primero se recorre el subárbol izquierdo, luego se visita nodo, y finalmente se recorre el subárbol derecho.

pos-orden: Primero se recorren los subárboles izquierdo y derecho y finalmante se visita el nodo.

Otras 3 estrategias naturales se obtienen recorriendo de derecha a izquierda:

pre-orden, der-izq: Primero se visita el nodo, y luego se recorren los subárboles derecho y izquierdo.

in-orden, der-izq: Primero se recorre el subárbol derecho, luego se visita nodo, y finalmente se recorre el subárbol izquierdo.

pos-orden, der-izq: Primero se recorren los subárboles derecho y izquierdo y finalmante se visita el nodo.

Para ver un ejemplo, pensemos en un algoritmo que devuelve una lista con todos los nodos elementos del árbol (respetando las repeticiones). Para ello es necesario recorrer todo el árbol buscando los elemento para devolver en la lista. Evidentemente, alcanza con visitar cada nodo una vez. Hay al menos 6 maneras de hacerlo según cuál de las 6 estrategias enumeradas se utilice. Por ejemplo, en la recorrida pre-orden el algoritmo se puede escribir:

$\text{pre\_orden}(<>) = [\,]$
$\text{pre\_orden}(< l, e, r >) = e \triangleright \text{pre\_orden}(l) \mathbin{++} \text{pre\_orden}(r)$

En la recorrida in-orden, se obtieneel siguiente algoritmo

$\text{in\_orden}(<>) = [\,]$
$\text{in\_orden}(< l, e, r >) = \text{in\_orden}(l) \mathbin{++} (e \triangleright \text{in\_orden}(r))$

Por último, en la recorrida pos-orden se obtiene:

$\text{pos\_orden}(<>) = [\,]$
$\text{pos\_orden}(< l, e, r >) = \text{pos\_orden}(l) \mathbin{++} \text{pos\_orden}(r) \triangleleft e$

Es importante notar que si utilizamos el algoritmo in_orden en un ABB obtenemos una lista ordenada de menor a mayor. Efectivamente, para todo nodo del árbol, in_orden lista primero todos los elementos del subárbol izquierdo (que por tratarse de un ABB son todos los menores al elemento del nodo), luego el elemento del nodo, y finalmente los elementos del subárbol derecho (que por tratarse de un ABB son todos los mayores al elemento del nodo).

Si dado un ABB quisiéramos listar los elementos de mayor a menor podríamos utilizar el algoritmo in_orden_der_izq:

$\text{in\_orden\_der\_izq}(<>) = [\,]$
$\text{in\_orden\_der\_izq}(< l, e, r >) = \text{in\_orden\_der\_izq}(r) \mathbin{++} (e \triangleright \text{in\_orden\_der\_izq}(l))$

Observar que los 6 algoritmos sirven para recorrer cualquier árbol binario. Sólo hemos mencionado una propiedad adicional que se cumple en el caso en que alguno de los algoritmos in-orden se aplica a un ABB.

Recorriendo árboles finitarios. Un árbol finitario es similar a un árbol binario, sólo que en vez de tener 2 subárboles cada nodo tiene una cantidad (cualquiera) finita de subárboles. Podemos pensar en un grafo $G = (N,\text{root},\text{children})$ donde root es la raíz del árbol y children es la función que aplicada a un nodo del árbol devuelve una lista de los nodos que son sus hijos. El orden en que esa lista enumera los hijos es irrelevante. Sí es importante que si se aplica dos veces children al mismo nodo, en ambos casos listará los hijos en el mismo orden.

Para el caso de árboles finitarios, las 3 primeras estrategias vistas anteriormente se reducen a 2. En efecto, la que deja de ser una estrategia natural es la in_orden. Habiendo una cantidad arbitraria de hijos, ¿entre cuáles hijos visitaríamos al nodo en sí?. Las otras dos siguen siendo fácil de enunciar:

pre-orden: Primero se visita el nodo, y luego se recorren los hijos en el orden que son enumerados por children.

pos-orden: Primero se recorren los hijos en el orden que son enumerados por children y finalmante se visita el nodo.

Estas 2 estrategias abarcan las versiones de izquierda a derecha y de derecha a izquierda (y muchas más) dado que hemos abstraído el orden en que children enumera los hijos.

A continuación escribimos un algoritmo que visita todos los nodos de un árbol finitario en pre-orden:

```
fun pre_orden(G=(N,root,children)) ret mark: marcas
    inicializar(mark)
    pre_recorrer(G, mark, root)
end

proc pre_recorrer(in G, in/out mark: marcas, in n: N)
     visitar(mark,n)
     for m ∈ children(n) do pre_recorrer(mark, m) od
end
```

Si queremos que el algoritmo le asigne a cada nodo un número que indique en qué orden los fue visitando, definimos

```
type marcas = tuple
                 ord: array[N] of int
                 cont: int
              end

proc inicializar(out mark: marcas)
     mark.cont:= 0
end

proc visitar(in/out mark: marcas, in n: N)
     mark.cont:= mark.cont+1
     mark.ord[n]:= mark.cont
end
```

Observar que no es necesario inicializar las celdas del arreglo ya que, tratándose de un árbol y dado que se comienza por la raíz cada nodo será visitado exactamente una vez y en ese momento la celda correspondiente inicializada.

A diferencia del anterior, el siguiente algoritmo recorre el árbol en pos-orden:

```
fun pos_orden(G=(N,root,children)) ret mark: marcas
    inicializar(mark)
    pos_recorrer(G, mark, root)
end
proc pos_recorrer(in G, in/out mark: marcas, in n: N)
     for m ∈ children(n) do pre_recorrer(mark, m) od
     visitar(mark,n)
end
```

A continuación, definimos para $n, m \in N$ la siguiente relación: $n \preceq m$ sii existen $p, q, r \in N$ tales que p y q aparecen en children(r) en ese orden, p es ancestro de n, y q es ancestro de m. Esta relación es antisimétrica, es decir, $n \preceq m \wedge n \succeq m \Rightarrow n = m$.

Se puede demostrar que si pre_mark = pre_orden(G) y pos_mark = pos_orden(G) entonces para todo $n, m \in N$ se cumple:

$$\begin{array}{rcl} \text{pre\_mark}[n] \leq \text{pre\_mark}[m] & \iff & n \preceq m \vee n \text{ es ancestro de } m \\ \text{pos\_mark}[n] \geq \text{pos\_mark}[m] & \iff & n \succeq m \vee n \text{ es ancestro de } m \end{array}$$

Precondicionamiento. Estas observaciones nos permiten dar un ejemplo de una técnica que suele llamarse precondicionamiento. Se trata de elaborar los datos de determinada manera con el fin de facilitar luego su utilización. Por ejemplo, si tenemos n números naturales $m_1, m_2, \ldots, m_n$ y queremos averiguar para una gran cantidad de números inicialmente desconocidos $k_1, k_2, \ldots, k_M$ si cada uno de ellos es divisible por $m_1, m_2, \ldots,$ y m_n o no. En vez de testear para cada $1 \leq i \leq M$ si k_i es divisible por m_1, luego por m_2, etc. podemos preprocesar los datos de la siguiente forma. Calculamos el mínimo común múltiplo m de $m_1, m_2, \ldots, m_n$ y luego simplemente testeamos para cada $1 \leq i \leq M$ si k_i es divisible por m. Al calcular ese mínimo común múltiplo estamos "precondicionando", creando condiciones favorables para realizar las cuentas siguientes.

Otro ejemplo se puede obtener para árboles finitarios. Si dado un árbol finitario quiero averiguar para una gran cantidad de pares de nodos inicialmente desconocidos $(n_1, m_1), \ldots, (n_M, m_M)$ si n_i es ancestro de m_i de manera eficiente, conviene calcular los arreglos pre_mark y pos_mark. Por las observaciones realizadas 2 párrafos más arriba, n_i será ancestro de m_i sii pre_mark[n] $\preceq$ pre_mark[m] y pos_mark[n] $\succeq$ pos_mark[m].

```
fun ancestro(pre_mark, pos_mark: marcas, n, m: int) ret b: bool
    b:= (pre_mark[n] ≤ pre_mark[m] ∧ pos_mark[n] ≥ pos_mark[m])
end
```

Búsqueda en profundidad (DFS). Las dos estrategias vistas para recorrer árboles finitarios tienen algo en común: ambas corresponden a realizar recorridas "en profundidad," es decir, antes de visitar el segundo hijo de cada nodo se visitan todos los descendientes del primer hijo. Esto es lo que se llama DFS (Depth-First Search), búsqueda en profundidad. Esta estrategia de búsqueda es muy útil en la práctica incluso para grafos que no sean necesariamente árboles. A continuación vemos cómo extender la estrategia a grafos arbitrarios.

Uno se enfrenta esencialmente con dos problemas al intentar extender los algoritmos de recorrida a grafos que no sean necesariamente árboles: la posibilidad de que no sean

conexos y la posible existencia de ciclos. Si el grafo no es conexo, no podremos recorrer todo el grafo tan solo partiendo de un nodo. Si el grafo tiene ciclos, nuestra recorrida en profundidad puede llevarnos a un nodo ya visitado e incluso, si no se tiene precaución, a una recorrida que no termine.

Para cuidar estos aspectos modificamos ligeramente nuestras primitivas para manipular marcas de modo de poder averiguar en cualquier momento si un nodo fue visitado. Redefinimos el procedimiento inicializar y definimos una nueva función booleana visitado.

```
proc inicializar(out mark: marcas)
      mark.cont:= 0
      for n ∈ N do mark.ord[n]:= 0 od
end

fun visitado(mark: marcas, n: N) ret b: bool
    b:= (mark.ord[n] ≠ 0)
end
```

Con estas definiciones es posible definir la búsqueda en profundidad. Dicha búsqueda se asemeja a la que uno podría realizar de encontrarse encerrado en un laberinto teniendo la posibilidad de marcar con piedritas los lugares recorridos en la búsqueda de la salida. La búsqueda partiría del lugar donde nos encontramos dejando caer periódicamente piedritas de forma que queden marcados los lugares ya recorridos. Cuando llegamos a una división del camino elegimos la primera de la derecha y continuamos, siempre marcando el camino elegido. Cuando llegamos a un lugar marcado, damos marcha atrás hasta encontrar la última división en la que nos queden alternativas sin marcar. Intentamos la primer alternativa de la derecha que aún no fue marcada y volvemos a avanzar marcando.

En pseudo-código este algoritmo de búsqueda se puede escribir:

```
fun dfsearch(G=(N,neighbours)) ret mark: marcas
    inicializar(mark)
    for n ∈ N do
        if ¬visitado(mark,n) then dfs(G, mark, n) fi
    od
end

proc dfs(in G, in/out mark: marcas, in n: N)
      visitar(mark,n)
      for m ∈ neighbours(n) do
          if ¬visitado(mark,m) then dfs(G, mark, m) fi
      od
end
```

El procedimiento recursivo dfs es muy similar al procedimiento pre_recorrer visto para árboles salvo que en vez de children(n) usa neighbours(n) (un nombre más apropiado para denominar la lista de todos los nodos vecinos al nodo n teniendo en cuenta que G no es necesariamente un árbol) y que antes de realizar una llamada recursiva se

asegura de que el vecino en cuestión no haya sido visitado. Esto evita visitar más de una vez un mismo nodo, y con ello que la recorrida se pierda en un ciclo del grafo.

El procedimiento dfsearch también es similar al procedimiento pre_orden que vimos para árboles salvo que en vez de iniciar la recorrida sólo a partir de la raíz la inicia dentro de un **for**. Es decir, la inicia tantas veces como sea necesario a partir de diferentes nodos. Esto soluciona el problema que podría presentarse al recorrer grafos no conexos: se iniciaría la búsqueda una vez por componente conexa. Observar que sólo se inicia la búsqueda a partir de un nodo si el mismo no ha sido ya visitado en una búsqueda iniciada en un nodo anterior.

Por último, es importante destacar que este algoritmo se comporta de forma adecuada tanto para grafos dirigidos como para grafos no dirigidos. En efecto, la única diferencia entre un caso y el otro estaría dada por la función neighbours que en el caso de grafos no dirigidos debe satisfacer la condición adicional m $\in$ neighbours(n) sii n $\in$ neighbours(m).

¿Cómo hacer una versión análoga de dfs que corresponda a pos_recorrer en vez de a pre_recorrer? Hay que tener cuidado con los ciclos.

Algoritmo iterativo. Es posible dar una versión iterativa del procedimiento dfs de manera similar a como hicimos en su momento con el algoritmo de ordenación por intercalación: utilizando una pila. En este caso la pila almacenará el camino que va desde el nodo donde se inició el procedimiento dfs hasta el nodo que actualmente se está visitando, nodo que estará en el tope de la pila. Si este nodo tiene algún vecino sin visitar, se lo visita y agrega a la pila. Si no, se lo borra de la misma.

```
proc dfs(in G, in/out mark: marcas, in n: N)
    var p: stack of N
    empty(p)
    visitar(mark,n)
    push(n,p)
    while ¬is_empty(p) do
        if existe m ∈ neighbours(top(p)) tal que ¬visitado(m) then
          visitar(mark,m)
          push(m,p)
        else pop(p)
        fi
    od
end
```

Búsqueda a lo ancho (BFS). Como expresamos en la sección anterior, la búsqueda en profundidad es aquélla en que para cada nodo n se visita al i-ésimo vecino de n sólo después de que se hayan visitado todos los nodos alcanzables desde los vecinos anteriores a i. La posibilidad contraria, es decir, aquélla en que para cada nodo n se visite al i-ésimo vecino de n antes de que se visiten los nodos no-vecinos alcanzables desde los demás vecinos, se denomina BFS (Breadth-First Search) búsqueda a lo ancho.

A diferencia de DFS, no hay una definición recursiva natural del algoritmo BFS. De todas maneras, es muy fácil modificar la versión iterativa de DFS para que realice búsqueda a lo ancho: basta con reemplazar la pila por una cola.

```
proc bfs(in G, in/out mark: marcas, in n: N)
    var q: queue of N
    empty(q)
    visitar(mark,n)
    enqueue(q,n)
    while ¬is_empty(q) do
        if existe m ∈ neighbours(first(q)) tal que ¬visitado(m) then
          visitar(mark,m)
          enqueue(q,m)
        else dequeue(q)
        fi
    od
end
```

El algoritmo principal bfsearch es idéntico a dfsearch, sólo invoca a bfs en vez de invocar a dfs.

```
fun bfsearch(G=(N,neighbours)) ret mark: marcas
    inicializar(mark)
    for n ∈ N do
        if ¬visitado(mark,n) then bfs(G, mark, n) fi
    od
end
```

Backtracking. Backtracking es una técnica de programación que se aplica a problemas que requieren la construcción de una solución de manera incremental, paso a paso, donde en cada paso hay una cantidad finita de posibilidades y no se sabe cuál o cuáles llevan a la solución o a las soluciones. Eso significa que después de varios pasos puede que se demuestre que no se esté arribando a la solución, por lo que algunas decisiones anteriores deben revisarse y reemplazarse por otras. Backtracking es un mecanismo adecuado que contempla intentar todas las combinaciones posibles de manera de que si una solución existe la misma será encontrada. Backtracking corresponde a realizar una búsqueda en profundidad en un grafo implícito. Es decir, un grafo que no está presente en el enunciado del problema, pero que puede explicitarse si uno lo desea expresando cuáles son en cada paso las diferentes posibilidades que podrían llevar hacia la solución.

Problema de la mochila. El primer problema que intentaremos resolver utilizando backtracking es el de la mochila. Recordemos que tenemos n objetos de valor $v_1, \ldots, v_n$ y peso $w_1, \ldots, w_n$ y una mochila de capacidad W. Se quiere obtener el máximo valor posible sin exceder la capacidad de la mochila.

Una manera de analizar este problema es pensando que en el paso i se decide si se incorpora o no el objeto i. Explicitando el grafo, tendríamos

$$N = \{p \in \{0,1\}^* \mid |p| \leq n \wedge w(p) \leq W\}$$
$$w([x_1, \ldots, x_k]) = \sum_{i=1}^{k} x_i * w_i$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{0,1\}.q = p \lhd b)$$

Los nodos son secuencias de 0's y 1's. El nodo $[0, 0, 1, 1]$ corresponde al caso en que, de los primeros 4 objetos, se ha decidido guardar en la mochila solamente el tercero y el cuarto. Los demás objetos, desde el quinto al n-ésimo, aún no han sido considerados. Una arista de p a q consiste en tomar una decisión (0 ó 1) respecto al siguiente objeto. De $[0, 0, 1, 1]$ sólo hay dos aristas: una a $[0, 0, 1, 1, 0]$ y otra a $[0, 0, 1, 1, 1]$.

En el algoritmo este grafo sólo es implícito. El siguiente algoritmo determina el máximo valor alcanzable sin exceder el peso utilizando backtracking:

```
fun mochila(v:array[1..n] of valor, w:array[1..n] of peso, j: peso, i: int) ret r: valor
    {calcula el máximo valor alcanzable con los objetos i, ..., n sin exceder el peso j}
  if i > n then r:= 0
  else if w[i] >j then r:= mochila(v,w,j,i+1)
       else r:= máx(mochila(v,w,j,i+1),v[i]+mochila(v,w,j-w[i],i+1))
       fi
  fi
end
```

La función principal simplemente llama a la anterior con j igual a W e i igual a 1.

Otra posibilidad que da lugar a un grafo implícito más pequeño es pensar que a cada paso se decide incorporar cualquier objeto (que no supere el peso restante). El grafo se puede explicitar como sigue

$$N = \{p \subseteq \{1, \ldots, n\} \mid |p| \leq n \wedge w(p) \leq W\}$$
$$w(p) = \sum_{k \in p} w_k$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{1, \ldots, n\}.q = p \lhd b \wedge \forall k \in p.k < b)$$

Los nodos son ahora conjuntos de objetos. Cada uno corresponde al caso en que dichos objetos hayan sido guardados en la mochila. Para no considerar más de una vez los mismos casos, sólo se pueden agregar objetos posteriores a los que ya se agregaron. Esto se refleja en la condición $\forall k \in p.k < b$ que dice justamente que el objeto b que se agrega es posterior a todos los que ya estaban.

Nuevamente este grafo da lugar a un algoritmo que utiliza backtracking:

```
fun mochila(v:array[1..n] of valor, w:array[1..n] of peso, j: peso, i: int) ret r: valor
    {calcula el máximo valor alcanzable con los objetos i, ..., n sin exceder el peso j}
  r:= 0
  for k:= i to n do
      if w[k] ≤ j then r:= máx(r,v[k]+mochila(v,w,j-w[k],k+1)) fi
  od
end
```

Como en el caso anterior, la función principal es

```
fun mochila_ppal(v:array[1..n] of valor, w:array[1..n] of peso, W: peso) ret r: valor
    {calcula el máximo valor alcanzable con todos los objetos sin exceder el peso W}
  r:= mochila(v,w,W,1)
end
```

Ocho reinas. Otro problema interesante para resolver usando la técnica del backtracking es el de calcular el número de maneras diferentes de ubicar 8 reinas (o damas) en un

tablero de ajedrez de manera de que ninguna de ellas amenace a ninguna de las demás. Recordemos que una reina amenaza todas las casillas que se encuentran en la misma fila, columna o diagonal que ella. A escala más pequeña puede formularse el problema con 4 reinas en un tablero de 4 filas por 4 columnas. No es difícil encontrar una solución al problema manualmente.

Primer intento. Lo primero que podríamos hacer es revisar todas las maneras posibles de ubicar 8 reinas en un tablero de ajedrez y controlar para cada una de esas distribuciones si hay una reina que ataque a otra. Para ubicar la primer reina tenemos 64 celdas posibles, para la segunda 63, etc. Si somos un poco más cuidadosos podemos evitar permutaciones (da lo mismo si la reina que está en una celda fue la primera o la quinta que acomodamos). Para ello cada reina será ubicada en casillas posteriores a las reinas que ya se ubicaron. Esto da lugar al algoritmo

```
fun ocho_reinas_1() ret r: int
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    var sol: list of int
    r:= 0
    for i1:= 1 to 57 do
        for i2:= i1+1 to 58 do
            for i3:= i2+1 to 59 do
                for i4:= i3+1 to 60 do
                    for i5:= i4+1 to 61 do
                        for i6:= i5+1 to 62 do
                            for i7:= i6+1 to 63 do
                                for i8:= i7+1 to 64 do
                                    sol:= [i1,i2,i3,i4,i5,i6,i7,i8]
                                    if solucion_1(sol) then r:= r+1 fi
                                od
                            od
                        od
                    od
                od
            od
        od
    od
end
```

donde asumimos que solucion_1 se encarga de verificar si la lista con las posiciones de las 8 reinas es o no una solución.

Para la primera reina hay 57 posibilidades en vez de 64 ya que debe dejar como mínimo 7 casillas libres al final para las siguientes 7 reinas.

Para hacer explícito el grafo tomamos

$$N = \{[p_1, p_2, \ldots, p_n] \in \{1, \ldots, 64\}^* \mid n \leq 8 \wedge p_1 < p_2 < \ldots < p_n \leq 56 + n\}$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{1, \ldots, 64\}.q = p \lhd b)$$

Segundo intento. Si bien ya tenemos una solución, evidentemente la misma considera demasiadas posibilidades que fácilmente podrían descartarse. Por ejemplo, si la primer reina y la segunda se colocan en la misma fila, no importa donde se coloquen las demás, no obtendremos una solución. Sin embargo el algoritmo anterior considera todas las (muchísimas) formas posibles de colocar las restantes 6 reinas.

A continuación presentamos un algoritmo mejor, que considera sólo las distintas maneras de colocar 8 reinas en filas diferentes, condición necesaria para obtener una solución. La primer reina irá a la primer fila, la segunda reina a la segunda fila, etc. Por ello, para cada reina se determina sólo un número entre 1 y 8, el de la columna que le corresponde en la fila que ya tiene asignada.

```
fun ocho_reinas_2() ret r: int
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    var sol: list of int
    r:= 0
    for j1:= 1 to 8 do
        for j2:= 1 to 8 do
            for j3:= 1 to 8 do
                for j4:= 1 to 8 do
                    for j5:= 1 to 8 do
                        for j6:= 1 to 8 do
                            for j7:= 1 to 8 do
                                for j8:= 1 to 8 do
                                    sol:= [j1,j2,j3,j4,j5,j6,j7,j8]
                                    if solucion_2(sol) then r:= r+1 fi
                                od
                            od
                        od
                    od
                od
            od
        od
    od
end
```

Para hacer explícito el grafo tomamos

$$N = \{p \in \{1, \ldots, 8\}^* \mid |p| \leq 8\}$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{1, \ldots, 8\}.q = p \lhd b)$$

Antes de pasar al tercer intento conviene presentar una versión recursiva de este algoritmo:

```
fun ocho_reinas_2() ret r: int
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    r:= 0
    or_2([ ], r)
end
```

```
proc or_2(in sol: list of int, in/out r: int)
                                        {calcula el número de maneras de extender sol}
                                      {hasta ubicar en total 8 reinas sin que se amenacen}
    if |sol| = 8 then
      if solucion_2(sol) then r:= r+1 fi
    else for j:= 1 to 8 do
            or_2(sol ◁ j, r)
         od
    fi
end
```

El algoritmo recursivo es fácilmente modificable para resolver el mismo problema para una cantidad arbitraria de reinas (y tableros suficientemente grandes).

Observar que la función solucion_2 es diferente a la solucion_1 del primer intento, ya que ésta recibe listas de números entre 1 y 8 mientras que aquélla recibía listas de números entre 1 y 64.

Tercer intento. Así como observarmos que para encontrar una solución dos reinas no pueden ir en la misma fila y logramos mejorar el algoritmo para tener eso en cuenta, a continuación lo mejoraremos para tener en cuenta que, análogamente, dos reinas no pueden ir en la misma columna. Por suerte la lista sol contiene exactamente las columnas que ya han sido ocupadas.

```
fun ocho_reinas_3() ret r: int
                {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
  r:= 0
  or_3([ ], r)
end

proc or_3(in sol: list of int, in/out r: int)
                                        {calcula el número de maneras de extender sol}
                                      {hasta ubicar en total 8 reinas sin que se amenacen}
    if |sol| = 8 then
      if solucion_3(sol) then r:= r+1 fi
    else for j:= 1 to 8 do
            if j ∉ sol then or_3(sol ◁ j, r) fi
         od
    fi
end
```

En este caso, solucion_3 es idéntico a solucion_2.

Haciendo explícito el grafo tenemos

$$N = \{p \in \{1, \ldots, 8\}^* \mid |p| \leq 8 \wedge p \text{ sin repeticiones}\}$$
$$\forall p, q \in N.(p \to q \iff \exists b \in \{1, \ldots, 8\}.q = p \lhd b)$$

Cuarto intento. Por último, para reducir aún más el espacio de búsqueda podemos considerar, para cada nueva reina, sólo aquéllas diagonales que aún no han sido ocupadas. Llamaremos bajadas y subidas respectivamente a las listas que llevan la cuenta de las diagonales que bajan de izquierda a derecha y las que suben de izquierda a derecha. Asumimos que tenemos dos funciones bajada(i,j) y subida(i,j) que dicen a qué diagonal (en bajada y en subida respectivamente) pertenece la casilla (i,j).

```
fun ocho_reinas_4() ret r: int
                    {calcula el número de maneras de ubicar 8 reinas sin que se amenacen}
    r:= 0
    or_3([ ], [ ], [ ], r)
end
proc or_3(in sol, bajadas, subidas: list of int, in/out r: int)
                                              {calcula el número de maneras de extender sol}
                                          {hasta ubicar en total 8 reinas sin que se amenacen}
                                      {bajadas y subidas son las diagonales ya amenazadas}
    if |sol| = 8 then r:= r+1 fi
    else i:= |sol|+1
        for j:= 1 to 8 do
            if j ∉ sol ∧ bajada(i,j) ∉ bajadas ∧ subida(i,j) ∉ subidas
            then or_3(sol ◁ j, bajadas ◁ bajada(i,j), subidas ◁ subida(i,j), r)
            fi
        od
    fi
end
```

Observar que dadas las restricciones consideradas al agregar una nueva reina, una vez que se han ubicado las 8 no es necesario usar una función auxiliar solucion_4 ya que efectivamente ninguna reina atacará a otra.

Por último, dado que todas las casillas (i,j) que comparten una misma bajada (y sólo ellas) dan idéntico valor a la expresión i-j y todas las casillas que comparten una misma subida (y sólo ellas) dan idéntico valor a la expresión i+j, definimos

```
fun bajada(i,j:int) ret r: int
    r:= i-j
end
fun subida(i,j:int) ret r: int
    r:= i+j
end
```

Haciendo el grafo explícito tenemos

$$N = \{[p_1, \ldots, p_n] \in \{1, \ldots, 8\}^* \mid n \leq 8 \wedge$$
$$\wedge (i \neq j \Rightarrow p_i \neq p_j) \wedge (i \neq j \Rightarrow p_i - i \neq p_j - j) \wedge (i \neq j \Rightarrow p_i + i \neq p_j + j)\}$$
$$\forall p, q \in N.(p \rightarrow q \Longleftrightarrow \exists b \in \{1, \ldots, 8\}.q = p \lhd b)$$

Branch & bound. Otra técnica de programación importante es la de ramificación y cota (o ramificación y poda). Sus características principales pueden enunciarse a continuación:

- explora un grafo implícito (en esto se parece a backtracking),
- se aplica a problemas de optimización (en esto se parece a la técnica voraz),
- en cada paso se calculan cotas sobre posibles soluciones siguiendo la rama actual del grafo,
- si las cotas muestran que la rama actual lleva a soluciones peores que la encontrada hasta ahora, se descarta (poda de ramas),
- las cotas también pueden usarse para ver cuál rama es más promisoria

A continuación exhibimos un ejemplo de la técnica branch & bound.

1.0.1. Problema de la asignación de tareas. Tenemos n tareas y n operarios, a cada uno de ellos queremos asignarle una tarea. Para cada operario cada tarea tiene un costo. Por ejemplo, según la formación del operario, la tarea puede llevarle mayor o menor tiempo. Asumimos que el costo de la tarea para cada operario viene dado por una tabla. Por ejemplo, si tenemos 4 operarios a, b, c y d y 4 tareas 1, 2, 3 y 4, la siguiente tabla indica cuánto cuesta que cada operario realice cada tarea:

	1	2	3	4
a	11	12	18	40
b	14	15	13	22
c	11	17	19	23
d	17	14	20	28

Queremos encontrar la manera de asignar tareas a operarios que minimice el costo total. Para ello, la técnica de branch & bound determina primero una asignación cualquiera que sirva de cota superior. En el ejemplo, si a se le asigna la tarea 1, a b la 2, a c la 3 y a d la 4, el costo es 11+15+19+28=73. Sabemos entonces que el costo de la asignación de costo mínimo no superará 73. Veremos que este dato servirá para podar el árbol de búsqueda.

Para cada rama de dicho árbol, estimaremos también el costo de la mejor solución que podríamos, en principio, encontrar en dicha rama. Originariamente (cuando estamos en la raíz del árbol de búsqueda) no hemos hecho ninguna asignación. Podemos pensar que, en principio, la mejor solución que podríamos encontrar sería asignando cada tarea a quien le cuete menos. En el ejemplo, la tarea uno cuesta como mínimo 11, la dos 12, la tres 13 y la cuatro 22. Sabemos entonces que no tenemos esperanzas de encontrar una solución que cueste menos de 11+12+13+22=58. Puede que ese costo mínimo sea inalcanzable, es decir, que toda solución cueste más. En particular, para obtener ese costo acabamos de sumar el 13 con el 22 que se obtienen de asignarle al operario b dos tareas, la 3 y la 4. Sabemos que sólo se le podrá asignar 1. De todas maneras, el número obtenido (58) sirve como una estimación.

Tenemos así las dos cotas, pero recordemos que la cota mayor, 73, es para todo el árbol (se usa para podar ramas) y la inferior es una por rama.

A partir de la raíz del árbol, podemos considerar 4 posibilidades según al operario a se le asigne la tarea 1, 2, 3 ó 4:

$$58 \begin{cases} a:1 & 60 \\ a:2 & 58 \\ a:3 & 65 \\ a:4 & 78* \end{cases}$$

A cada rama le hemos asociado el costo mínimo que podríamos obtener en principio en esa rama. Así, una vez que la tarea 1 fue asignada al operario a, lo mejor que podría ocurrir es que a las tareas 2, 3 y 4 las realice el operario a quien le cuesta menos, sin contar al operario a que ya está ocupado con la tarea 1 (cuyo costo es 11): 11+14+13+22=60. De la misma manera, cuando al operario a se le asigna la tarea 2, lo mejor que se podría sería 11+12+13+22=58, cuando al operario a se la signa la tarea 3, 18+11+14+22=65, y cuando al operario a se le asigna la tarea 4, 40+11+14+13=78. En este último caso vemos que la manera más económica de distribuir las tareas sería, en principio, 78, es decir, más caro que 73 que es la cota superior que teníamos. Por ello, no es necesario explorar esta última rama del árbol de búsqueda. Señalamos entonces que esta rama se poda con el símbolo "$*$".

Las otras 3 ramas deben explorarse. Lo más conveniente podría ser explorar la segunda rama, ya que el menor costo obtenible, en principio, es de sólo 58. Es decir, parece haber mayores chances de obtener la solución óptima siguiendo esa rama.

$$
58\begin{cases}
a:1 & 60 & & & \\
a:2 & 58 & \begin{cases} a:2,b:1 & 68* & \\ a:2,b:3 & 59 & \begin{cases} a:2,b:3,c:1,d:4 & 64 \\ a:2,b:3,c:4,d:1 & 65* \end{cases} \\ a:2,b:4 & 64* & \end{cases} & & \\
a:3 & 65* & & & \\
a:4 & 78* & & &
\end{cases}
$$

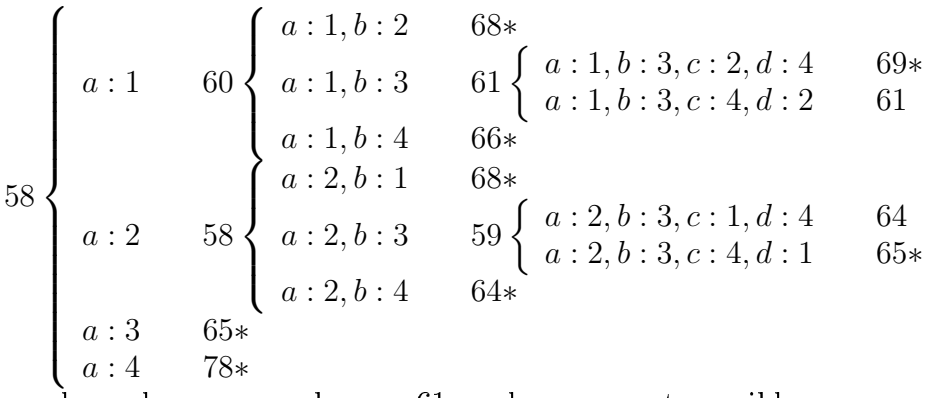
Las cotas inferiores correspondientes a cada nueva rama se obtuvieron realizando las sumas 14+12+19+23=68, 11+12+13+23=59 y 11+12+19+22=64. No podamos ninguna rama ya que en todos los casos el número obtenido es mayor a la cota superior.

Nuevamente elegimos la rama más promisoria, en este caso la de cota inferior 59. Quedan 2 posibilidades según se asigna a c la tarea 1 ó 4 y la restante a d. Las mismas dan un costo total de 64 y 65. Obtuvimos ahora una solución de costo 64 que es una cota superior mejor que la que teníamos antes (73) a la solución óptima. Inmediatamente se podan todas aquéllas ramas que sabemos que no pueden llevarnos a soluciones mejores (las marcamos nuevamente con "$*$"). A partir de ahora, 64 es la cota superior.

La única que queda sin marcar es la de cota inferior 60. La ramificación que se puede realizar ahora es

$$
58\begin{cases}
a:1 & 60 & \begin{cases} a:1,b:2 & 68* & \\ a:1,b:3 & 61 & \\ a:1,b:4 & 66* & \end{cases} & & \\
a:2 & 58 & \begin{cases} a:2,b:1 & 68* & \\ a:2,b:3 & 59 & \begin{cases} a:2,b:3,c:1,d:4 & 64 \\ a:2,b:3,c:4,d:1 & 65* \end{cases} \\ a:2,b:4 & 64* & \end{cases} & & \\
a:3 & 65* & & & \\
a:4 & 78* & & &
\end{cases}
$$

La única rama que queda por explorar es la de cota inferior 61. Las opciones son si c hace la tarea 2 y d la 4 o c la 4 y d la 2:



con lo cual se comprueba que 61 es el menor costo posible.

Para escribirlo como algoritmo podemos utilizar una cola de prioridades de manera de seleccionar cada vez la rama más promisoria.

```
fun asignacion(c: array[1..n,1..n] of costo) ret r: list of int
    var p: pqueue of list of int
    empty(p)
    enqueue(c,p,[ ])
    while |first(c,p)| < n do
        for i:= 1 to n do
            if i ∉ first(c,p) then enqueue(c,p,first(c,p) ◁ i) fi
        od
        dequeue(c,p)
    od
    r:= first(c,p)
end
```

Observar que las operaciones de la cola tienen el parámetro c con los costos de las realizaciones de las tareas para poder calcular la cota inferior que se utilizará como prioridad. Si la cola de prioridades es implementada con un heap, la operación first no necesitará el parámetro c. El cálculo de la prioridad de cada lista se realiza con la función:

```
fun estimacion(c: array[1..n,1..n] of costo, t: list of int) ret r : costo
    var m: int
    r:= 0
    for i:= 1 to |t| do r:= r + c[i,t.i] od
    for j:= 1 to n do
        if j ∉ t then
          m:= ∞
          for i:= |t|+1 to n do
              m:= mín(m,c[i,j])
          od
          r:= r + m
        fi
    od
end
```