APUNTES PARA ALGORITMOS Y ESTRUCTURAS DE DATOS II

PARTE 3: TÉCNICAS DE DISEÑO DE ALGORITMOS

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

Hemos explicado que los algoritmos de ordenación rápida y ordenación por intercalación son ejemplos típicos de algoritmos 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 cada uno 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 $x_1, x_2, \ldots, x_a$
    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 fun

Otro 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.

En el algoritmo de ordenación por intercalación (merge_sort). 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.

Los algoritmos divide y vencerás son ejemplos de una técnica que se utiliza con frecuencia: la llamada técnica **top-down** (de arriba hacia abajo) de diseño. Esta técnica aborda la resolución de problemas complejos descomponiendolos sucesivamente en problemas más sencillos hasta llegar finalmente a los problemas de solución trivial. En el caso de divide y vencerás, todos los problemas involucrados (el inicial y sus sucesivos sub-problemas) son de idéntica naturaleza. Lo único que se gana al descomponer es reducir el tamaño de los datos.

Ya que se ha mencionado a la técnica **top-down**, es conveniente mencionar también a su opuesta: la técnica **bottom-up** (de abajo hacia arriba) de diseño. Según esta técnica, uno puede abordar la solución del problema planteado construyendo de

manera incremental soluciones a problemas menores que se presume serán útiles para resolver el problema original. La versión iterativa de la ordenación por intercalación es un ejemplo de uso de la técnica **bottom-up**. También lo es el desarrollo de sucesivas librerías de algoritmos de complejidad incremental, cada una de ellas utilizando las librerías anteriores, con la intención de poder resolver con su uso el problema planteado originariamente.

También hemos visto el algoritmo de ordenación rápida (quick_sort), tal vez el más famoso de la técnica divide y vencerás. Se diferencia de la ordenación por intercalación en que descompone el fragmento de arreglo a ordenar de modo de que no sea necesario intercalar las partes ordenadas a posteriori. Por ello, la combinación de las partes ordenadas resulta trivial, pero la descomposición es más compleja y la realiza el procedimiento pivot.

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

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 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: nat, n: nat) ret r: nat                  {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 fun
```

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 algoritmo de multiplicación usual:

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

Por cada dígito de 1593 escribimos un 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 buscada **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 es necesario considerar todas las posibles elecciones, problemas para los que no se poseen criterios satisfactorios de elección que garanticen la construcción de una solución óptima.

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. Como las funciones **solución** y **factible** no tienen en cuenta cuestiones de optimalidad, el éxito del algoritmo en alcanzar una solución óptima, depende del criterio de selección provisto por la función **seleccionar**.

Esquemáticamente, la forma de los algoritmos voraces es la siguiente:

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

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.[12] 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.

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

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 pudiéndose pagar con solamente 3 monedas de 10 centavos.

[12] 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.

Supongamos que tenemos un conjunto de denominaciones para el que este algoritmo obtiene la solución óptima. Para analizar el orden del algoritmo, lo escribimos con un poco más de detalle y lo generalizamos a un conjunto arbitrario n de denominaciones, que podría venir dado por un arreglo ordenado decrecientemente:

```
{Pre: d[1] > d[2] > ... > d[n] }
fun cambio(d:array[1..n] of nat, m: monto) ret S: array[1..n] of nat
    for i:= 1 to n do
        S[i]:= m div d[i]
        m:= m mod d[i]
    od
end fun
```

Como vemos, ordenadas las denominaciones de mayor a menor el criterio de selección se vuelve trivial: el algoritmo intenta utilizar una a una las denominaciones y calcula directamente el número de veces que se utiliza cada una, realizando una división. Realiza una división (en realidad dos si contamos mod) por cada denominación. El orden es entonces $\Theta(n)$. Si fuera necesario ordenar el arreglo de denominaciones, sería $\Theta(n \log n)$.

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 cargar 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 algoritmo que se encuentra más abajo.

Observamos que la primera rama 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 segunda rama del **if** 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.

```
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: nat
    for i:= 1 to n do s[i]:= 0 od
    weight:= 0
    do weight ≠ W → c:= mejor objeto remanente
                    if weight + w[c] ≤ W → s[c]:= 1
                                          weight:= weight + w[c]
                       weight + w[c] > W → s[c]:= (W-weight)/w[c]
                                          weight:= W
                    fi
    od
end fun
```

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 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 $G = (V, A)$ con costos no negativos en las aristas y queremos encontrar el

camino de costo mínimo desde un vértice v hasta cada uno de los demás. Asumiremos que el grafo viene dado por el conjunto de vértices $V = \{1, 2, \ldots, n\}$ 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. Asumimos que $L[j, j] = 0$ para todo vértice j. En caso de no haber ninguna arista de i a j, diremos que $L[i, j] = \infty$. Si quisiéramos aplicar lo que sigue a grafos no dirigidos simplemente agregaríamos la condición $L[i, j] = L[j, i]$ para todo par de vértices i y j.

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

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 v a j. Si pensamos que los costos son longitudes, en D se devolverán las distancias de v a cada uno de los demás vértices.

```
fun Dijkstra(L: array[1..n,1..n] of costo, v: nat) ret D: array[1..n] of costo
    var c: nat
    C:= {1,2,...,n}-{v}
    for j:= 1 to n do D[j]:= L[v,j] od
    do 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
    od
end fun
```

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

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 v a j. Inicialmente, por lo dicho en el párrafo anterior, $D[j]$ debe ser $L[v, j]$. Eso explica la inicialización de D que se realiza en el primer **for**.

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

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

$\forall j \notin C . D[j] =$ costo del camino de costo mínimo de v a j
$\forall j \in C . D[j] =$ costo del camino **especial** de costo mínimo de v 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 ejecución del cuerpo del ciclo un vértice 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 v a c), $D[c]$ es en realidad el costo del camino de costo mínimo de v 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 vértice 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 v a c. ¿Puede haber un camino **no especial** de v a c que cueste menos? Si lo hubiera, dicho camino necesariamente debería tener, por ser **no especial**, algún vértice intermedio **no especial**. Sea w el primer vértice **no especial** que ocurre en ese camino comenzando desde v. Evidentemente el camino **no especial** consta de una primera parte que llega a w. Esa primera parte es un camino **especial** de v a w, por lo que su costo, dice el invariante, debe ser $D[w]$. El costo del camino completo **no especial** de v 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 vértices **no especiales**) $D[c]$, necesariamente debe cumplirse $D[c] \leq D[w]$. Esto demuestra que no puede haber un camino **no especial** de v a c que cueste menos que $D[c]$. Por ello, c puede sin peligro ser considerado un vértice **especial** ya que $D[c]$ contiene el costo del camino de costo mínimo de v a c.

Inmediatamente después de agregar c entre los vértices **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 vértice **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 v a j cuyo último vértice **especial** es c. En efecto, los caminos **especiales** de v a j que pasan por c y cuyo último vértice **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 vértice **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 vértice se vuelve **especial**. Inicialmente v lo es. Por ello, al cabo de n-2 iteraciones, tenemos solamente 1 vértice **no especial**. Llamemos k a ese vértice. El invariante resulta

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

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

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

y por consiguiente

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

Árbol generador de costo mínimo (Minimum Spanning Tree). Sea $G = (V, A)$ un grafo conexo no dirigido con un costo no negativo asociado a cada arista. Se dice que $T \subseteq A$ es un árbol generador (intuitivamente, un tendido) si (V, 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 $|T| = |V| - 1$. En efecto,

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

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

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 $|V|-1$ 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 $|V| - 1$ aristas tendremos un árbol generador.

Estas propiedades puede formularse así:

1. Sea $G = (V, A)$ un grafo sin ciclos. Si $|A| = |V| - 1$ entonces G es conexo.
2. Sea $G = (V, A)$ un grafo conexo. Si $|A| = |V| - 1$ 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 $|T| = |V| - 1$.
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 vértices 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 $G = (V, A)$ conexo con costos no negativos y sea $B \subset V$. 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 $|U \cup \{a\}| = |U| + 1 = |V|$ 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 $W = (U - \{b\}) \cup \{a\}$. El costo de W es menor o igual que el de U. Como $b \notin T$ ya que nadie en T deja B, entonces $T \cup \{a\} \subseteq W$. Luego, $T \cup \{a\}$ es promisorio.

Algoritmo de Prim. El algoritmo de Prim comienza a partir de un vértice k y arma progresivamente una red que conecta a ese vértice con los demás. Inicialmente T es vacío, el vértice k no está conectado con nadie. En cada paso se incorpora una arista a la red T, de modo de que un nuevo vértice resulte conectado a k. En la presentación en pseudo-código que se ve a continuación, C es el conjunto de vértices 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=(V,A) con costos en las aristas, k: nat) ret T: conjunto de aristas
    var c: arista
    C:= V-{k}
    T:= {}
    do |V|-1 times → c:= arista {i,j} de costo mínimo tal que i ∈ C y j ∉ C
                     C:= C-{i}
                     T:= T ∪ {c}
    od
end fun
```

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 vértice 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 vértice 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=(V,A) con costos en las aristas) ret T: conjunto de aristas
    var i,j: vértice; u,v: componente conexa; c: arista
    C:= A
    T:= {}
    do |T| < |V| - 1 → c:= arista {i, j} de C de costo mínimo
                       C:= C-{c}
                       u:= find(i)
                       v:= find(j)
                       if u ≠ v → T:= T ∪ {c}
                                   union(u,v)
                       fi
    od
end fun
```

La función find devuelve la componente conexa de un vértice, 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 auxiliares.

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, cuando 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 ≡ 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 trep = array[1..n] of nat
```

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 eficientemente 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: trep)
    for i:= 1 to n do rep[i]:= i od
end proc
```

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: trep, i: nat) ret r: nat
   r:= rep[i]
end fun
```

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: trep, in i,j: nat)          {i ≠ j ∧ i = rep[i] ∧ j = rep[j]}
    for k:= 1 to n do
        if rep[k]=i → rep[k]:= j fi
    od
end proc
```

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: trep, i: nat) ret b: bool
   b:= (rep[i] = i)
end fun
```

Practicamos una mejora en el procedimiento union:

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

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 k quién es el representante de k, dice quién fue alguna vez el representante de k. Si dice que el representante de k fue i, y dice que el de i fue j, y resulta que is_rep(rep,j) es verdadero, entonces el representante de k ahora es j. Como vemos, al buscar el representante de un elemento puede ser necesario recorrer varias posiciones de la tabla hasta encontrar el representante actual.

```
fun find(rep: trep, i: nat) ret r: nat
    var j: nat
    j:= i;
    do ¬ is_rep(rep,j) → j:= rep[j] od
    r:= j
end fun
```

En realidad, ahora la tabla puede verse como una foresta, es decir, un conjunto de árboles donde cada elemento es un vértice 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.

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

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. Se puede observar que, a pesar de este efecto secundario, find conserva algunas características de las funciones. Por ejemplo, la condición find(rep,i) = find(rep,i) será siempre verdadera. Puede que la segunda llamada se ejecute más rápidamente gracias a los efectos secundarios causados por la primera llamada, pero el resultado será el mismo.

Los demás algoritmos (init, is_rep y union) quedan como están. Este cambio torna eficiente al find. No puede ser frecuentemente lineal, ya que cada llamada allana el camino de las siguientes.

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

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 agregaría 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: trep, in i,j: nat)     {i ≠ j∧ is_rep(rep,i) ∧ is_rep(rep,j)}
    if nr[i] ≤ nr[j] → nr[j]:= nr[i]+nr[j]
                       rep[i]:= j
       nr[i] > nr[j] → nr[i]:= nr[i]+nr[j]
                       rep[j]:= i
    fi
end proc
```

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 $-rep[i]$.

Para implementar esta idea necesitamos redefinir el tipo trep, que era arreglo de números naturales:

```
type trep = array[1..n] of int
```

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

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

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: trep, i: nat) ret b: bool
   b:= (rep[i] <0)
end fun
```

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: trep, in i,j: nat)        {i ≠ j∧ is_rep(rep,i) ∧ is_rep(rep,j)}
    if rep[i] ≥ rep[j] → rep[j]:= rep[i]+rep[j]
                         rep[i]:= j
       rep[i] < rep[j] → rep[i]:= rep[i]+rep[j]
                         rep[j]:= i
    fi
end proc
```

Backtracking

Cuando se obtiene un algoritmo voraz que resuelve un problema, en general es una solución eficiente. Para ello, es necesario determinar un buen criterio que permita seleccionar convenientemente un nuevo candidato a integrar la solución óptima.

Eso no siempre es posible. En el extremo opuesto, a veces nos encontramos sin ningún criterio para seleccionar candidatos. Lo que se puede hacer en ese caso es intentar todas las maneras posibles de armar soluciones a partir de los candidatos existentes. Esto se llama **backtracking** porque equivale a tomar una decisión y luego volver hacia atrás y cambiarla por otra.

Evidentemente esto, en general, es ineficiente. Por ello sólo se recomienda cuando no se nos ocurre nada mejor. Así y todo, es conveniente ordenar la recorrida del espacio de búsqueda de soluciones de manera de no repetir soluciones, ya que ello volvería aún más ineficiente al algoritmo.

Problema de la moneda. Ya vimos en la página 86 que el algoritmo voraz para el problema de la moneda no siempre encuentra la solución óptima, depende de cuáles sean las denominaciones de las monedas. Utilizando backtracking 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.

Aplicar backtracking significa considerar todas las maneras posibles de pagar el monto k, y elegir entre ellas la que involucre el menor número de monedas. Observar la diferencia con la técnica voraz que directamente construía la menor solución.

Para ello, dados i y j tales que $0 \leq i \leq n$ y $0 \leq j \leq k$ se definirá $m(i, j)$ 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 exactamente 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 $1 + \infty = \infty$ y que ∞ es el elemento neutro de la operación mínimo. Una forma de definir recursivamente $m(i, j)$ es a través de las siguientes ecuaciones:

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

En efecto, cuando $j = 0$ la manera de pagar exactamente el monto 0 es sin utilizar ninguna moneda, es decir, 0 monedas. Cuando $j > 0$ 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 tercera 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 consideran 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 mín$(m(i-1,j), 1+m(i,j-d_i))$.

Puede observarse que la ejecución recursiva de este algoritmo consiste en explorar todas las posibilidades y elegir la que minimiza el número de monedas. En efecto, en la cuarta ecuación surgen dos posibilidades: ambas son exploradas y se elige la que lleva a menor solución. En cada una de las tres primeras ecuaciones, en cambio, hay una única posibilidad.

Este algoritmo puede escribirse fácilmente en pseudo-código:

```
fun cambio(d:array[1..n] of nat, i,j: nat) ret r: nat
    if j=0 then r:= 0
    else if i = 0 then r:= ∞
         else if d[i] >j then r:= cambio(i-1,j)
              else r:= min(cambio(i-1,j),1+cambio(i,j-d[i]))
              fi
         fi
    fi
end fun
```

La llamada principal es cambio(d,n,k). Existen varias otras soluciones que utililzan backtracking para resolver este problema. Por ejemplo, la misma función m(i,j) definida más arriba podría definirse con las siguientes ecuaciones:

$$m(i,j) = \begin{cases} 0 & j = 0 \\ 1 + \text{mín}(\{m(x, j-d_x) | 1 \leq x \leq i \wedge d_x < j\}) & j > 0 \end{cases}$$

Problema de la mochila. En la página 87 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 de la capacidad restante.

Ahora utilizaremos backtracking para resolver 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.

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 para el caso del problema de la moneda, esto puede escribirse fácilmente como un algoritmo recursivo en pseudo-código.

Problema del camino de costo mínimo. El algoritmo de Dijkstra (página 88) para obtener, en un grafo dirigido, (el costo de) un camino de costo mínimo desde un vértice a cada uno de los demás fue otro de los ejemplos vistos del uso de la técnica voraz. A continuación veremos un algoritmo que utiliza backtracking para calcular (el costo de) los caminos de costo mínimo para todo vértice origen y vértice destino.

Al igual que en la página 88, tenemos un grafo dirigido $G = (V, A)$ con costos no negativos en las aristas. Asumimos que el grafo viene dado por el conjunto de vértices $V = \{1, 2, \ldots, n\}$ 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 vértices 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 vértices 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 vértice 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 vértices $1, 2, \ldots, k$. Como $k > 0$ esta lista de vértices 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 $m_{k-1}(i, k) + m_{k-1}(k, j)$.

Programación Dinámica

Muchos problemas pueden resolverse utilizando backtracking. En general la solución es ineficiente. Si se repasan las soluciones dadas en la sección anterior, se puede observar que todas ellas tienen alguna ecuación con más de una llamada recursiva. Esto da lugar a algoritmos exponenciales. Veremos cómo la técnica de **programación dinámica** puede ayudar a evitar este problema.

Dicha técnica 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 secuencia de 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 secuencia y dado $n \in \mathbb{N}$ ¿Cuántas veces se calcula f_1 para calcular f_n? Set $t(n)$ =número de veces que se calcula 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 materia, cuando estudiamos resolución de recurrencias, $t(n)$ 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 implementar más eficientemente la función de Fibonacci? Una manera que ya conocemos es utilizando directamente la expresión que se obtuvo al resolver la recurrencia. 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 nat** 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: nat) ret r: nat
    var f: array[0..max(n,1)] of nat
    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 fun
```

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 98 las siguientes ecuaciones para solucionar el problema de la moneda usando backtracking:

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

La posible reiteración de cálculos no es tan evidente como en el caso de Fibonacci, pero puede darse. Por ejemplo, para j suficientemente grande, suponiendo las denominaciones $d_{i-1} = 5$ y $d_i = 10$, puede observarse que evaluar $m(i,j)$ lleva a evaluar dos veces $m(i-1, j-10)$ (señalado por el subrayado):

$$\begin{aligned} m(i,j) &= \text{mín}(m(i-1,j), 1 + m(i,j-10)) \\ m(i,j-10) &= \text{mín}(\underline{m(i-1,j-10)}, 1 + m(i,j-20)) \\ m(i-1,j) &= \text{mín}(m(i-2,j), 1 + m(i-1,j-5)) \\ m(i-1,j-5) &= \text{mín}(m(i-2,j-5), 1 + \underline{m(i-1,j-10)}) \end{aligned}$$

Esta duplicación lo vuelve exponencial ya que $m(i-1, j-10)$ tiene a su vez duplicaciones, etc.

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 nat, k: nat) ret r: nat
    var m: array[0..n,0..k] of nat
    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 fun
```

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 presentaremos 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.

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.

```
fun cambio(d:array[1..n] of nat, k: nat) ret nr: array[0..n] of nat
    var m: array[0..n,0..k] of nat
        r,s: nat
    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 fun
```

Problema de la mochila. Lo mismo ocurre con el problema de la mochila. La versión recursiva de la página 99:

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

se traduce a una versión iterativa basada en una matriz:

```
fun mochila(v:array[1..n] of valor, w:array[1..n] of nat, W: nat) 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 fun
```

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

```
fun mochila(v:array[1..n] of valor, w:array[1..n] of nat, W: nat)
                                                    ret nr: array[1..n] of bool
    var m: array[0..n,0..W] of valor
        r,s: nat
    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 fun
```

Problema del camino de costo mínimo: Algoritmo de Floyd. En la página 100 vimos las siguientes ecuaciones para solucionar el problema del camino de costo mínimo usando backtracking:

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

Nuevamente podríamos representar m por una matriz 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 fun
```

Hemos utilizado un arreglo tridimensional m pensando en que m[k,i,j] es la representación 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,i,k] y m[k-1,k,j]). 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. Justo esa columna (y fila) no cambian en el paso k. En efecto, si i=k, m[k-1,k,j] = m[k-1,k,k]+m[k-1,k,j] y, si j=k, también m[k-1,i,k] = m[k-1,i,k]+m[k-1,k,k]. Entonces, como el cálculo de m[k,i,j] depende solamente de la propia celda m[k-1,i,j] y de valores de la columna y fila k, que no cambian, resulta posible utilizar una sola matriz (de dimensión 2) y actualizar cada celda. 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 fun
```

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 vértice 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 fun
```