EJERCICIOS RESUELTOS PROGRAMACIÓN III

Curso 2008 - 2009

Ejercicios resueltos de programación 3

Tema 3. Notación asintótica.

Alumna: Alicia Sánchez
Centro: UNED-Las Rozas (Madrid)

El *índice* de los ejercicios será el siguiente. Se observa que sólo hay cuestiones de exámenes (de los ejercicios cortos), por lo que se debería hacer hincapié en ellos:

Introducción teórica:

Previo a resolver los ejercicios pondremos un poco de **teoría**, que nos vendrá bien para luego hacer los ejercicios:

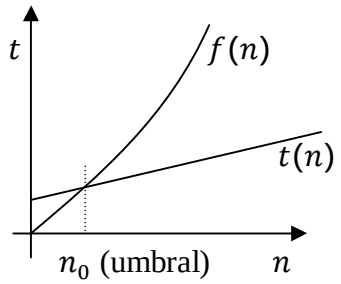
Empezaremos viendo las distintas notaciones, para "el orden de", cota inferior y orden exacto:

- Notación para el orden de (cota superior):

  Es conveniente disponer de un símbolo matemático para representar *el orden de*. Sea $f: \mathbb{N} \to \mathbb{R}^{\geq 0}$ una función arbitraria de los números naturales en los reales no negativos. Le indicará mediante $O(f(n))$ el conjunto de todas las funciones $t: \mathbb{N} \to \mathbb{R}^{\geq 0}$ tales que $t(n) \leq c * f(n)$, para todo $n \geq n_0$ para una constante positiva c y un umbral entero n_0. En otras palabras:

  $$O(f(n)) \equiv \{t: \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists\, c \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall n \geq n_o | t(n) \leq c * f(n)\}$$

  Gráficamente sería:



  siendo:

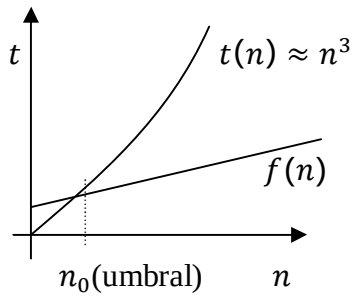
  n_0: Cierto umbral del tamaño del problema.
  $f(n)$: Acota superiormente a la función $t(n)$.

- Notación para la cota inferior:

  Matemáticamente, esto significa que existe una constante real positiva d y un umbral entero n_0 tal que $t(n) \geq d * f(n)$ siempre que $n \geq n_0$.

  $\Omega(f(n)) \equiv \{t: \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists\, d \in \mathbb{R}^+, n_0 \in \mathbb{N}, \forall\, n \geq n_o | t(n) \geq d * f(n)\}$

  Gráficamente sería:



  $t(n) \in \Omega(f(n))$: Cota inferior.

  $f(n) \in O(t(n))$: Cota superior.

- Notación para el orden exacto:

  Diremos que $t(n)$ está en Theta de $f(n)$, o lo que es igual que $t(n)$ está en el orden exacto de $f(n)$ y lo denotamos $t(n) \in \theta(f(n))$, si $t(n)$ pertenece tanto a $O(f(n))$ como a $\Omega(f(n))$.

  La definición formal de θ es:

  $$\theta(f(n)) = O(f(n)) \cap \Omega(f(n)).$$

  Por tanto,

  $$\theta(f(n)) \equiv \{t: \mathbb{N} \to \mathbb{R}^{\geq 0} | \exists\, a, b \in \mathbb{R}^{+}, n_0 \in \mathbb{N}, \forall n \geq n_o | \\ a * f(n) \leq t(n) \leq b * f(n)\}.$$

  Decimos que el conjunto del orden exacto está acotado tanto inferior como superiormente por $f(n)$. Podemos probarlo tanto por la definición como por la regla del límite.

Para demostrar que una función dada no pertenece al orden de otra función $f(n)$ tendremos estas formas:

- Demostración por contradicción: Es la forma más sencilla. Consiste en demostrar la veracidad de una sentencia demostrando que si negación da lugar a una contradicción.
- La regla del umbral generalizado: Implica la existencia de una constante real y positiva c tal que $t(n) \leq c * f(n)$ para todos los $n \geq 1$ (tomaremos n_0 como 1, nos interesa más la definición dada por la regla del umbral sin generalizar).
- La regla del límite: Lo definiremos completamente tras analizar la cota superior y el coste exacto.

La primera y segunda manera no la usaremos por norma general, ya que no nos compensará. En cuanto a la última será la que usemos, de nuevo recordaremos la definición y lo que significa cada resultado:

La regla del límite: Nos permite comparar dos funciones en cuanto a la notación asintótica se refiere. Tendremos que calcular el siguiente límite:

$\lim_{n\to\infty} \frac{f(n)}{g(n)}$.

Se nos darán 3 resultados:

1. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = c \in R \Rightarrow$

$$\Rightarrow \left\{ \begin{array}{lll} f(n) \in O(g(n)) & f(n) \in \Omega(g(n)) & f(n) \in \theta(g(n)) \\ g(n) \in O(f(n)) & g(n) \in \Omega(f(n)) & g(n) \in \theta(f(n)) \end{array} \right\}.$$

Estas funciones se comportan igual. Se diferencian en una constante multiplicativa.

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

$$\Rightarrow \left\{ \begin{array}{lll} f(n) \notin O(g(n)) & f(n) \in \Omega(g(n)) & f(n) \notin \theta(g(n)) \\ g(n) \in O(f(n)) & g(n) \notin \Omega(f(n)) & g(n) \notin \theta(f(n)) \end{array} \right\}.$$

Por muy alta que sea la constante multiplicativa de $g(n)$ nunca superará a $f(n)$.

3. $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0 \Rightarrow$

$$\Rightarrow \left\{ \begin{array}{lll} f(n) \in O(g(n)) & f(n) \notin \Omega(g(n)) & f(n) \notin \theta(g(n)) \\ g(n) \notin O(f(n)) & g(n) \in \Omega(f(n)) & g(n) \notin \theta(f(n)) \end{array} \right\}.$$

$g(n)$ crece más exponencialmente que $f(n)$. Sería su cota superior.

1ª parte. Cuestiones de exámenes:

Febrero 2002 -2ª (ejercicio 2)

Enunciado: Un algoritmo de coste $O(n^2)$ tarda 15 segundos en realizar un determinado procesamiento sobre un ordenador a 450 MHz. ¿Cuánto tiempo se tarda en realizar el mismo procesamiento con el mismo algoritmo en una máquina 3 veces más lenta?

Respuesta: Se nos plantea un problema en el que cambia la velocidad, que equivale a la implementación. En este caso, se divide por tres la velocidad, lo que implica que tarda aún más. Serían 15 seg. * 3 = 45 seg.
Si en esta misma máquina cambiamos el tamaño del problema al doble tardaremos 4 veces más ($(2 * n)^2 = 4 * n^2$), si es triple serian 9 veces más, siguiendo el planteamiento anterior.

Diciembre 2003 (ejercicio 1)

Enunciado: ¿Qué significa que el tiempo de ejecución de un algoritmo está "en el orden exacto de $f(n)$"? Demostrar que $T(n) = 5 * 2^n + n^2$ está en el orden exacto de 2^n.

Respuesta:

Para la primera pregunta tendremos que poner la definición previamente escrita en la teoría o bien en el resumen del tema. No la pondremos, por estar en este mismo documento, en páginas anteriores.

Para la segunda pregunta emplearemos la regla del límite. En este caso, tomaremos $f(n) = 5 * 2^n + n^2$ y $g(n) = 2^n$. Pasamos a resolver el límite, como sigue:

$$\lim_{n\to\infty} \frac{5*2^n+n^2}{2^n}$$

Al resolverla llegamos a una indeterminación, por lo que aplicaremos el **teorema de L'Hôpital** tantas veces como sea necesario hasta llegar a una conclusión coincidiendo con cualquiera de los casos anteriores:

$$\lim_{n\to\infty} \frac{5*2^n+n^2}{2^n} = \lim_{n\to\infty} \frac{5*2^n*log(2)+2*n}{2^n*log(2)} = \lim_{n\to\infty} \frac{5*2^n*log(2)^2+2}{2^n*log(2)^2} =$$
$$\lim_{n\to\infty} \frac{5*2^n*log(2)^3}{2^n*log(2)^3} = 5.$$

Observamos que al final sería una constante numérica, por lo que estamos en el caso 1, concluyendo que $f(n) \in \theta(g(n))$ y, por tanto, queda demostrado.

Septiembre 2004 (ejercicio 1)

Enunciado: ¿Qué significa que el tiempo de ejecución de un algoritmo está "en el orden exacto de $f(n)$"? Demostrar que $T(n) = n^3 + 9 * n^2 * \log(n)$ está en el orden exacto de n^3.

Respuesta: La definición del orden exacto la hemos visto previamente. En cuanto a la demostración, se haría igual, empleando el **teorema de L'Hôpital**, por lo que evitamos realizar de nuevo el ejercicio por hacerlo previamente.

Septiembre 2005-reserva (ejercicio 2)

Enunciado: Demostrar formalmente si existe relación de pertenencia entre $f(n)$ y $O(g(n))$ y también entre $g(n)$ y $O(f(n))$ considerando $f(n) = 2^n$ y $g(n) = 2^{2*n}$

Respuesta: Se nos pide que veamos las relaciones entre la función y el orden de otra función dada, por tanto, tras ver la teoría previa, dada en el ejercicio anterior vemos que tenemos que usar la **regla del límite**:

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{2^n}{2^{2*n}}$.

En este caso, no podremos aplicar **L'Hôpital** por ser una exponencial, ya que la derivada quedaría igual:

$\frac{da^n}{dn} = a^n$.

Para resolverlo, tendremos que descomponer una de las dos funciones:

$g(n) = 2^{2*n} = 2^n * 2^n$.

Con esta nueva información pasamos a resolver el límite:

$\lim_{n\to\infty} \frac{2^n}{2^n * 2^n} = \lim_{n\to\infty} \frac{1}{2^n} = 0$.

Estaremos en el caso número 3 de los que vimos previamente, por tanto, deducimos que:

$f(n) \in O(g(n)) \Rightarrow 2^n \in 2^{2*n}$.

$g(n) \notin O(f(n)) \Rightarrow 2^{2*n} \notin 2^n$.

Septiembre 2005 (ejercicio 3)

Enunciado: Sea $T(n) = 4 * n^2 - 3 * n + 2$ el tiempo de ejecución de un algoritmo. Demuestra si es cierta o falsa cada una de las siguientes afirmaciones (0.5 puntos cada una):

a) $T(n) \notin O(n^2 * \log(n))$
b) $T(n) \notin O(n^3)$
c) $T(n) \in \Omega(n * log(n))$
d) $T(n) \in O(n^2)$

Respuesta: Nuevamente usaremos la **regla del límite** y también **L'Hôpital**:

a) $T(n) \notin O(n^2 * \log(n))$

$\lim_{n\to\infty} \frac{4*n^2-3*n+2}{n^2*\log(n)} = \lim_{n\to\infty} \frac{8*n-3}{2*n*log(n)+n^2*\frac{1}{n}} = \lim_{n\to\infty} \frac{8}{2*log(n)+3} = 0$.

Pertenece al orden indicado, por tanto, la afirmación es **falsa**.

b) $T(n) \notin O(n^3)$

$\lim_{n\to\infty} \frac{4*n^2-3*n+2}{n^3} = \lim_{n\to\infty} \frac{8*n+3}{3*n^2} = \lim_{n\to\infty} \frac{8}{6*n} = 0$.

Por lo que $T(n) \in O(n^3)$, por lo que la afirmación es **falsa**.

c) $T(n) \in \Omega(n * log(n))$

$\lim_{n\to\infty} \frac{4*n^2-3*n+2}{n*log(n)} = \lim_{n\to\infty} \frac{8*n+3}{log(n)+n*\frac{1}{n}} = \lim_{n\to\infty} \frac{8}{\frac{1}{n}} = \infty.$

Con lo que es **cierto**.

d) $T(n) \in O(n^2)$

$\lim_{n\to\infty} \frac{4*n^2-3*n+2}{n^2} = \lim_{n\to\infty} \frac{8*n-3}{2*n} = \lim_{n\to\infty} \frac{8}{2} = 4 \in R^+.$

En este caso, al ser $T(n) \in \theta(n^2)$ se cumple también que $T(n) \in O(n^2)$, por lo que es **cierto**.

Febrero 2008-1ª (ejercicio 2)

Enunciado: ¿Cuáles de las siguientes respuestas son verdaderas y cuáles falsas? Demuestra tus respuestas.

a) $n^2 \in O(n^3)$
b) $n^2 \in \Omega(n^3)$
c) $4 * n^3 - 3 * n + 2 \in \Omega(n * log(n))$
d) $n! \in \theta((2 * n + 1)!)$

Respuesta: Como en el ejercicio anterior emplearemos la misma técnica para resolverlo.

a) $n^2 \in O(n^3)$

Consideramos $f(n) = n^2$ y $g(n) = n^3$ y aplicamos el teorema del límite como sigue:

$\lim_{n\to\infty} \frac{n^2}{n^3} = \lim_{n\to\infty} \frac{2*n}{3*n^2} = \lim_{n\to\infty} \frac{2}{6*n} = 0.$

Esto significa que n^3 crece más rápidamente que n^2, por tanto, es **cierto**.

b) $n^2 \in \Omega(n^3)$

En este caso, veríamos que con el resultado anterior, podremos afirmar que es **falso.**

c) $4 * n^3 - 3 * n + 2 \in \Omega(n * log(n))$

$\lim_{n\to\infty} \frac{4*n^3-3*n+2}{n*log(n)} = \lim_{n\to\infty} \frac{8*n-3}{log(n)+n*\frac{1}{n}} = \lim_{n\to\infty} \frac{8}{\frac{1}{n}} = \infty.$

Es **cierto**, por ser $f(n) \in \Omega(n * log(n))$.

d) $n! \in \big((2 * n + 1)!\big)$

En este caso, no es posible usar **L'Hôpital** por ser un factorial, por lo que usaremos otro método. Lo resolveremos así:

$$\lim_{n\to\infty} \frac{n!}{(2 * n + 1)!} = \lim_{n\to\infty} \frac{(n) * (n-1) * \ldots * 1}{(2 * n + 1) * (2 * n) * (2 * n - 1) * \ldots * (n) * (n-1) * \ldots * 1} = 0$$

Vemos que al desarrollar el factorial podemos eliminar muchos valores. En este caso, el resultado es 0, ya que $g(n)$ crecerá más rápidamente que $f(n)$.
Como resultado, es **falsa**, por ser $f(n) \notin \Omega\big(g(n)\big)$ y, por tanto, $f(n) \notin \theta\big(g(n)\big)$.

Ejercicios resueltos de programación 3

Tema 4. Análisis algorítmico.

Alumna: Alicia Sánchez
Centro: UNED-Las Rozas (Madrid)

El *índice* de los ejercicios será el siguiente. Se observa que solo hay cuestiones de exámenes (de los ejercicios cortos), por lo que se debería hacer hincapié en ellos:

Introducción teórica:

Previo a resolver los ejercicios pondremos un poco de **teoría**, ya que estos ejercicios por norma general se resuelven de la misma manera. Si se pidiera algo distinto se explicará la teoría en el propio ejercicio.

- Reducción por sustracción:

  La ecuación de la recurrencia es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 0 \leq n < b \\ a * T(n-b) + c * n^k & \text{si } n \geq b \end{cases}$$

  La resolución de la ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta\left(a^{n \; div \; b}\right) & \text{si } a > 1 \end{cases}$$

- Reducción por división:

  La ecuación de la recurrencia es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

  La resolución de la ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

siendo:

a: Número de llamadas recursivas.

b: Reducción del problema en cada llamada.

$\boldsymbol{c * n^k}$: Todas aquellas operaciones que hacen falta además de las de recursividad.

1ª parte. Cuestiones de exámenes:

Diciembre 2002 (ejercicio 2)

Enunciado: calcular la ecuación de recurrencia y hallar el coste del siguiente algoritmo:

```
proc P (n) {
    var i, j: enteros
    j ← 1
    si n ≤ 1 entonces terminar
    si no {
        para i ← 1 hasta 7 hacer P (n DIV 2)
        para i ← 1 hasta 4 * n^3 hacer j ← j + 1
    }
```

Respuesta: Este ejercicio está hecho por una alumna, por lo que la solución dada no se asegura que este correcta.

Se nos pide que hallemos la **ecuación de recurrencia**, que puede ser la siguiente, siguiendo los dos bucles:

$$T(n) = 7 * t\ (n\ DIV\ 2) + \theta(n^3)$$

Siguiendo esta ecuación de recurrencia observamos que estamos ante una **recursión por división**. Veremos las siguientes variables:

a: Número de llamadas recursivas = 7
b: Reducción del problema en cada llamada recursiva = 2
$\boldsymbol{c * n^k}$: Todas aquellas operaciones que hacen falta además de la recursividad. Tendremos que siguiendo la ecuación de recurrencia previamente escrita, nos queda:
$c * n^k = n^3 \Rightarrow k = 3$

Por tanto, siguiendo esta fórmula, nos quedará:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Calculamos $a = b^k$, que sustituyendo tenemos $7 < 2^3$, siendo, por tanto, el primer caso. El coste será $\boldsymbol{\theta(n^3)}$, lo que significa que lo determinarán las llamadas externas a la recursividad.

Septiembre 2003 (ejercicio 1) (igual al ejercicio 1 de Septiembre 2007-reserva y ejercicio 2 de Septiembre 2008-reserva)

Enunciado: Hallar formalmente el coste de los siguientes algoritmos siendo $h(n,r,k) \in O(n)$.

```
Procedimiento uno (n,k: entero)
VAR i, r: entero;
    COMIENZO
        SI n < 2 ENTONCES DEVOLVER 1;
    SI NO
        COMIENZO
            r ← uno(n DIV 2, k − 1);
            r ← r + uno(n DIV 2, k + 1);
            r ← r * uno(n DIV 2, k + 2);
            DEVOLVER r;
        FIN
    FIN
```

```
procedimiento dos (n,k: entero)
VAR i, r: entero;
    COMIENZO
        SI n < 2 ENTONCES DEVOLVER 1;
    SI NO
        COMIENZO
            r ← dos(n DIV 2, k − 1);
            r ← r + dos(n DIV 2, k + 1);
            PARA i ← 1 HASTA n/2 HACER
                COMIENZO
                    r ← h(n, r, i);
                    r ← r + h(n, r − 1, i)
                FIN
            r ← r + dos(n DIV 2, k + 2);
            DEVOLVER r;
        FIN
    FIN
```

Respuesta: Este ejercicio lo veremos en más ocasiones y se hace de la misma manera. Pasamos a ver las distintas funciones:

Función uno: Resolvemos la **reducción por división** como vimos previamente:
La ecuación de la recurrencia es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Como es habitual en estos tipos de ejercicios las variables que tenemos son:

a: Número de llamadas recursivas = 3
b: Reducción del problema en cada llamada = 2
$\boldsymbol{c * n^k}$: Coste de las llamadas externas a la recursividad. Las llamadas extras son constantes, por lo que $c * n^k = 1 \Rightarrow k = 0$.

Por tanto, estaremos en el caso tercero $a > b^k$, siendo, por tanto, el coste $\boldsymbol{\theta\left(n^{\log_2 3}\right)}$.

Función dos: Como previamente hemos dicho será también una recursión con **reducción por división**, con estos datos:

a: Número de llamadas recursivas = 3
b: Reducción del problema en cada llamada = 2
$\boldsymbol{c * n^k}$: Coste de las llamadas externas a la recursividad. Por existir un bucle "para" con coste lineal y la función $h(n, r, k)$ con coste línea también, tal y como vimos en el ejercicio anterior, tendremos que el coste de las llamadas externas es n^2. Por tanto, $c * n^k = n^2 \Rightarrow k = 2$.

El coste será el correspondiente con el caso primero $a < b^k$, $\boldsymbol{\theta(n^2)}$, siendo estas llamadas externas más costosas que las recursivas, como antes, por lo que determinarán el coste total.

Febrero 2004 -2ª (ejercicio 1)

Enunciado: ¿En qué orden está el tiempo de ejecución del siguiente algoritmo? Justifica tu respuesta.

```
procedimiento h (n, i, j)
    si n > 0 entonces
        h (n − 1, i, 6 − i − j);
        escribir i " → " j;
        h (n − 1, 6 − i − j, j);
    fsi
```

Respuesta: Tendremos la siguiente ecuación de recurrencia que resuelve este algoritmo:

$T(n) = 2 * T(n-1) + 1$

Como es habitual en estos tipos de ejercicios deducimos las siguientes variables:

a: Número de llamadas recursivas = 2
b: Reducción del problema en cada llamada recursiva = 1
$\boldsymbol{c * n^k}$: Todas aquellas operaciones que hacen falta además de la recursividad. Tendremos que $k = 0$, por ser el tiempo extra constante.

Aplicando la recurrencia de **reducción por división** siguiendo esto:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta\left(a^{n \, div \, b}\right) & \text{si } a > 1 \end{cases}$$

Tendremos que $a = 2 > 1$, por tanto, aplicando la tercera fórmula, el coste del algoritmo es:

$T(n) \in \theta\left(a^{n \, div \, b}\right) = \theta(2^n)$.

Febrero 2005 -2ª (ejercicio 1)

Enunciado: Demuestra cuál es el orden exacto de complejidad en función de la variable n:

```
procedimiento lona (n: entero, j: entero)
    para i desde 1 hasta n div 4 hacer
        j := j + 1;
    fpara
    si n > 1 entonces
        lona(n − 2, j);
        escribir "j";
        lona(n − 2, n − j);
    fsi
fprocedimiento
```

Respuesta: En este caso, tendremos unas llamadas recursivas que corresponden con una **recursión por sustracción**. Se necesitará averiguar, por tanto, qué variable determina nuestro problema, pero nos lo dicen en el enunciado, que es *n*. Nuestra ecuación de recurrencia será:

$$T(n) = \begin{cases} c * n^k & \text{si } 0 \leq n < b \\ a * T(n-b) + c * n^k & \text{si } n \geq b \end{cases}$$

Los datos que tendremos a partir del algoritmo son:

a : Número de llamadas recursivas = 2

b: Reducción del subproblema en cada llamada = 2

$\boldsymbol{c * n^k}$: Las operaciones externas a las llamadas recursivas. En este caso, tendremos el bucle:

```
para i desde 1 hasta n div 4 hacer
    j := j + 1;
fpara
```

Al tener un bucle desde 1 hasta $n\ div\ 4$ de una operación elemental $j := j + 1$ tendremos que están en el orden de $O(n)$, entonces para averiguar el valor de k tendremos:

$c * n^k = n \Rightarrow k = 1$.

Aplicando la siguiente fórmula, tendremos:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta\left(a^{n\ div\ b}\right) & \text{si } a > 1 \end{cases}$$

Estaremos en este caso en la tercera opción, por lo que aplicando lo anterior tendremos:

$T(n) \in \theta\left(2^{n\ div\ 2}\right)$

Septiembre 2005-reserva (ejercicio 3)

Enunciado: Analizar y hallar el coste de los algoritmos siguientes (Considerar de orden $O(n^2)$ la función $h(n,r,k)$):

```
Procedimiento a (n,k: entero)
VAR i, r: entero;
    COMIENZO
        SI n < 3 ENTONCES DEVOLVER (1);
    SI NO
        COMIENZO
            r ← a(n DIV 2, k − 1);
            r ← r + a(n DIV 2, k + 1);
            PARA i ← 1 HASTA n^2 HACER
                COMIENZO
                    r ← r + k;
                FIN
            DEVOLVER (r);
        FIN
    FIN
```

```
procedimiento b (n,k: entero)
VAR i, r: entero;
    COMIENZO
        SI n < 4 ENTONCES DEVOLVER (1);
    SI NO
        COMIENZO
        r ← b(n DIV 2, k + 1);
        PARA i ← 1 HASTA n/2 HACER
            COMIENZO
                r ← h(n, r, i);
                r ← r + h(n, r − 1, i)
            FIN
        r ← r + b(n DIV 2, k + 2);
        DEVOLVER (r);
    FIN
FIN
```

Respuesta:

Este ejercicio se haría de igual manera a los demás, por lo que realmente es copiarlo y pegarlo modificando los datos nuevos que se nos den.

Función "a": Nos damos cuenta que *n* es el que indica el tamaño del problema, que determinará nuestro problema. Por lo anteriormente expuesto tendremos esta ecuación de **recursión por división** con la que resolveremos este algoritmo:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

Por este motivo, tendremos estas variables, que deduciremos del algoritmo:

a: Número de llamadas recursivas = 2

b: Reducción del problema en cada llamada = 2, que corresponde con la división de nuestro tamaño del problema (n).

$\boldsymbol{c * n^k}$: Coste de las llamadas externas a la recursividad. Para ello observamos que existe este bucle:

```
PARA i ← 1 HASTA n^2 HACER
    COMIENZO
        r ← r + k;
    FIN
```

} $O(n^2)$

Explicamos lo que hemos puesto más arriba. Tendremos, por tanto, un bucle desde 1 hasta n^2, por lo que el coste es $O(n^2)$, siendo, por tanto, $k = 2$.

La resolución de la **recursión por división** será:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Con esta información, ya estamos dispuestos a resolver la recurrencia, entonces, tendremos que saber si $a < b^k$, $a = b^k$ ó $a > b^k$. Por lo que, como antes hicimos

$2 < 2^2$.

Estamos en el **caso primero**, por lo que el coste del algoritmo es $\boldsymbol{\theta(n^2)}$, lo que significa que realmente el que determina el coste total es el de las llamadas externas a la recursividad, por tener un coste elevado, como hemos visto en otros ejercicios anteriores.

Función "b": Nuestra variable que determina el coste es *n* igualmente. A continuación, vemos que es una **recursión por división**, con la siguiente ecuación de recurrencia:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

De nuevo, tenemos que averiguar cuáles son las variables:

a: Número de llamadas recursivas = 2.
b: Reducción del problema en cada llamada recursiva = 2.
$\boldsymbol{c * n^k}$: Coste de las operaciones externas a la recursividad. Seguimos el mismo planteamiento que antes y vemos que el bucle "para" es el siguiente:

PARA $i \leftarrow$ 1 HASTA n/2 HACER
    COMIENZO
        $r \leftarrow h(n, r, i)$;
        $r \leftarrow r + h(n, r-1, i)$;
    FIN

$O(n^2)$

$O(n * n^2) = O(n^3)$

En este caso, tendríamos k = 3, razonando de igual manera que la función *a*.
Pasamos a calcular el coste de la recurrencia con esta fórmula (aunque esté en la misma página, repito que hay que machacarla mucho)

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Tendremos, por tanto, que $2 < 2^3$, siendo el primer caso y llegando a la conclusión del coste $\boldsymbol{\theta(n^3)}$.

Septiembre 2006-reserva (ejercicio 1)

Enunciado: Hallar el coste de los siguientes algoritmos siendo $h(n,r,k) \in O(n^2)$.

```
Procedimiento uno (n,k: entero)
VAR i, r: entero;
    COMIENZO
        SI n < 6 ENTONCES DEVOLVER 1;
    SI NO
        COMIENZO
            r ← uno(n DIV 6, k − 1);
            r ← r + uno(n DIV 6, k + 1);
            r ← r * uno(n DIV 6, k + 2);
            PARA i ← 1 HASTA 2 * n HACER
                COMIENZO
                    r ← h(n,r,i) + h(n,r − 1,i);
                FIN
            DEVOLVER r;
        FIN
    FIN
```

```
procedimiento dos (n,k: entero)
VAR i, r: entero;
    COMIENZO
        SI n < 8 ENTONCES DEVOLVER 1;
    SI NO
        COMIENZO
            r ← dos(n DIV 8, k − 1);
            r ← r + dos(n DIV 8, k + 1);
            PARA i ← 1 HASTA n/2 HACER
                COMIENZO
                    r ← h(n,r,i);
                    r ← r + h(n,r − 1,i)
                FIN
            r ← r + dos(n DIV 8, k + 2);
            DEVOLVER r;
        FIN
    FIN
```

Respuesta:

Función uno: Tendremos una recursividad que hay que resolver, tal y como vimos en el ejercicio anterior. Nos damos cuenta que n es el que indica el tamaño del problema, por lo que lo que determine nuestro problema. Por lo anteriormente expuesto tendremos esta fórmula de **recursión por división** con la que resolveremos este algoritmo:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

Por este motivo, tendremos estas variables, que deduciremos de dicho algoritmo:

a: Número de llamadas recursivas = 3

b: Reducción del problema en cada llamada = 6, que corresponde con la división de nuestro tamaño del problema (n).

$\boldsymbol{c * n^k}$: Coste de las llamadas externas a la recursividad. Para ello observamos que existe este bucle:

```
PARA i ← 1 HASTA 2 * n HACER
    COMIENZO
        r ← h(n,r,i) + h(n,r − 1,i);
    FIN
```

$O(n^2)$; $O(n * n^2) = O(n^3)$

Explicamos lo que hemos puesto más arriba. Tendremos, por tanto, un bucle desde 1 hasta $2 * n$, por lo que, evidentemente el coste del mismo es $O(n)$. En el enunciado se nos da que $h(n,r,k) \in O(n^2)$, por lo que el coste de las llamadas externas es $O(n^3)$.

Resolviendo $c * n^k = n^3$, vemos que $k = 3$.

La fórmula para hallar el coste de la recursión por división será:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Con esta información, ya estamos dispuestos a resolver la recurrencia, entonces, tendremos que saber si $a < b^k$, $a = b^k$ ó $a > b^k$. Por lo que, como antes hicimos

$3 < 6^3$.

Estamos en el caso primero, por lo que el coste del algoritmo es $\boldsymbol{\theta(n^3)}$, lo que significa que realmente el que determina el coste total es el de las llamadas externas a la recursividad.

Función dos: Según vemos en esta función el planteamiento es el mismo que en el anterior. Nos preguntamos antes de nada sobre qué variable determina el tamaño del problema, que es n, como antes. A continuación, vemos que es una **recursión por división**, con el siguiente esquema:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

De nuevo, tenemos que averiguar cuáles son las variables:

a: Número de llamadas recursivas = 3. Este número tiene truco, ya que debajo del bucle "para" hay una llamada recursiva, que, por supuesto, se tiene en cuenta.
b: Reducción del problema en cada llamada recursiva = 8.
$\boldsymbol{c * n^k}$: Coste de las operaciones externas a la recursividad. Seguimos el mismo planteamiento que antes y vemos que el bucle "para" es el siguiente:

PARA $i \leftarrow 1$ HASTA n/2 HACER
    COMIENZO
        $r \leftarrow h(n, r, i)$;
        $r \leftarrow r + h(n, r-1, i)$;
    FIN

$O(n^2)$

$O(n * n^2) = O(n^3)$

Llegamos a la misma conclusión que la función uno, en la que teníamos que k = 3.
Pasamos a calcular el coste de la recurrencia con esta fórmula (aunque esté en la misma página, repito que hay que machacarla mucho)

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Tendremos, por tanto, que $3 < 8^3$, siendo el primer caso y llegando a la conclusión anterior $\theta(n^3)$.

Septiembre 2006 (ejercicio 1)

Enunciado: Demuestra que el tiempo de ejecución en función de n del siguiente fragmento de código está acotado superiormente por $O(n^2)$ e inferiormente por $\Omega(n)$. Demuestra también su orden exacto de complejidad

```
para i desde j hasta n hacer
    para j desde 1 hasta n div i hacer
        escribir "i, j, k"
    fpara
fpara
```

Supón que el coste de "escribir" es constante.

Respuesta: En primer lugar debemos plantear el tiempo en función de n. Para ello, vamos a fijarnos en la **instrucción barómetro**, que por definición es aquélla que se ejecuta por lo menos con tanta frecuencia como cualquier otra instrucción del algoritmo. En este caso, escogeremos la instrucción más interna, que es *escribir "i, j, k"*.

A continuación, tendremos que contar cuántas veces se ejecuta, ya que es la que determinará el coste del algoritmo entero. En este caso, asumimos que la instrucción barómetro tiene coste constante, recordemos que es $O(1)$.

El tiempo $T(n)$ que tenemos será:

$$T(n) = \sum_{i=1}^{n} \sum_{j=1}^{n\ div\ i} 1 = \sum_{i=1}^{n} \frac{n}{i} = n * \sum_{i=1}^{n} \frac{1}{i}.$$

Para calcular la cota superior tendremos:

$$\sum_{i=1}^{n} \sum_{j=1}^{n\ div\ i} 1 \leq \sum_{i=1}^{n} \sum_{i=1}^{n} 1 = \sum_{i=1}^{n} n = n^2. \qquad T(n) \in O(n^2)$$

Esto significa que está acotado superiormente por n^2. Modificando uno de los sumatorios, hemos visto que $n \geq n\ div\ i$ (o n/i), por lo que hemos conseguido acotarlo.

Para la cota inferior tendremos:

$$\sum_{i=1}^{n} \sum_{j=1}^{n\ div\ i} 1 \geq \sum_{i=1}^{n} \sum_{i=1}^{1} 1 = \sum_{i=1}^{n} 1 = n. \qquad T(n) \in O(n)$$

Al igual que antes, hemos modificado el sumatorio para verificar que la cota inferior es n, en este caso, $1 \leq n\ div\ i$.

Se nos pide en la segunda parte del ejercicio que tenemos que demostrar el orden exacto. Por tanto, necesitamos saber cómo crece la serie $\sum_{i=1}^{n} \frac{1}{i}$. Para ello, podemos aproximar la serie con la integral

$$\int_1^n \frac{1}{x} dx$$

Esta integral es el **logaritmo natural** de n. Por tanto, la serie crece tan rápidamente como el $log(n)$. Así pues, se pueden encontrar dos constantes c y d, tal que $T(n)$ esté acotado superiormente por $c * n * log(n)$ e inferiormente por $d * n * log(n)$. En conclusión, el orden exacto es:

$$T(n) \in \theta\big(n * log(n)\big).$$

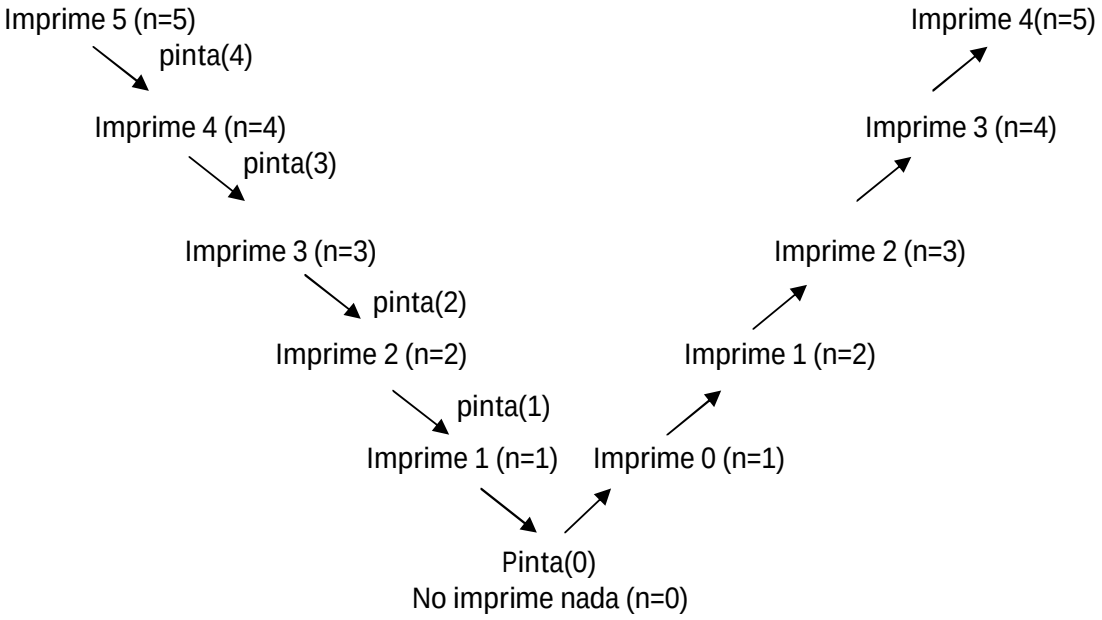
Septiembre 2006 (ejercicio 2)

Enunciado: Escribe la salida al realizar la llamada "pinta (5)", dado el siguiente código:

```
funcion pinta (int n)
    si n > 0 entonces
        escribir "n";
        pinta (n-1);
        escribir "n − 1";
    fsi
ffuncion
```

Demuestra el coste computacional de la función "pinta" suponiendo que "escribir" tiene coste constante.

Resultado: Para escribir la salida al realizar la llamada "pinta (5)" tendremos que tener muy claros los conceptos de **pila**, ya que al llamar a la recursividad apila y al dejar la llamada (finalizarlo) desapila. Por lo que, apilará hasta llegar a n = 0, en este caso, empezará a desapilar. Pondremos un dibujo para que se vea más claro:



Para resolver la recurrencia tendremos que hacerlo usando la **reducción por sustracción**, como hemos visto antes, siendo, por tanto, la ecuación de recurrencia y la resolución las siguientes:

$$T(n) = \begin{cases} c * n^k & \text{si } 0 \leq n < b \\ a * T(n-b) + c * n^k & \text{si } n \geq b \end{cases}$$

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta\left(a^{n \; div \; b}\right) & \text{si } a > 1 \end{cases}$$

En este caso, tendremos:

a: número de llamadas recursivas = 1

b: Reducción del problema en cada llamada = 1

$\boldsymbol{c * n^k}$: Coste de las operaciones externas a la recursividad. Tendremos que todas las operaciones son de coste constante, por lo que:

$c * n^k = 1 \Rightarrow k = 0$

Vemos que es el caso segundo, por ser $a = 1$. Por lo que el coste es $\boldsymbol{\theta(n)}$.

Diciembre 2006 (ejercicio 1)

Enunciado: Cuenta el número de operaciones elementales que efectúa el siguiente fragmento de código *f* en los casos mejor y peor, y a partir de estos resultados da su orden de complejidad, $O(f(n))$, y su orden exacto $\theta(f(n))$.

```
(1)   para i desde 1 hasta n hacer
(2)       para j desde n hasta i + 1 dec -1 hacer
(3)           si a[j − 1] > a[j] entonces
(4)               temp = a[j − 1];
(5)               a[j − 1] = a[j];
(6)               a[j] = temp
(7)           fsi
(8)       fpara
(9)   fpara
```

Respuesta: Este ejercicio se asemeja mucho a los hechos en el tema 2 en la primera parte, por lo que lo dejaremos sin hacer salvo tener en cuenta los distintos tiempos que existen, recordemos que eran tiempo de acceso a vector, tiempo de resta, etc etc. Por otro lado, la segunda parte del ejercicio es propiamente de este tema, por lo que lo hemos añadido por ese motivo. Por eso, no resolveremos el ejercicio aunque sí que diremos algunas pistas claves para ello.

Febrero 2007-2ª (ejercicio 1)

Enunciado: ¿Cuál es el tamaño del problema, n, que determina el orden de complejidad del siguiente procedimiento? Calcula el orden de complejidad del algoritmo en función de dicho tamaño.

```
tipo vector = array de enteros;

procedimiento examen(var a: vector; prim, ult,  x: entero): boolean;
    var
        mitad: entero;

(1)     si (prim ≥ ult) entonces
(2)         devolver a[ult] = x
(3)     si no
(4)         mitad = (prim + ult) div 2;
(5)         si (x = a[mitad]) entonces
(6)             devolver cierto;
(7)         si no si (x < a[mitad]) entonces
(8)             devolver examen(a,prim,mitad-1,x)
(9)         si no
(10)            devolver examen(a,mitad+1,ult,x);
(11)        fsi
(12)    fsi
```

Respuesta:

El tamaño del problema será $n = (ult - prim) + 1$, correspondiente con la condición del bucle "if" $(prim \geq ult)$.

En cuanto al análisis del orden de complejidad tendremos que realizar estos pasos:

1. **Análisis de su funcionamiento**, en la que lo simularemos.
2. **Análisis del coste** propiamente dicho.

En cuanto al análisis del funcionamiento, sólo decir que el objetivo de la búsqueda binaria es hallar un elemento x de un vector $a[1..n]$, el cual está ordenado de modo no decreciente (sólo podremos aplicar la búsqueda binaria si este vector está ordenado así).

Se nos dan tres casos:

- Elemento x está a la izquierda del elemento mitad (usaremos la variable que se llama *mitad* en el algoritmo).
- Elemento x coincide con la mitad del vector.
- Elemento x está a la derecha de la mitad del vector.

En cuanto al análisis del coste, tenemos que si se cumple la condición de la línea (1) el coste es constante, recordemos $O(1)$. A continuación, si nos fuéramos por la línea (3), tendríamos estas derivaciones, es decir, puede ser cualquiera de estas opciones:

1. Por línea (6), en cuyo caso el coste es constante, $O(1)$.
2. Por línea (8), en cuyo caso es coste t(n/2) + cte.
3. Por línea (10), en cuyo caso es coste t(n/2) + cte.

Planteamos la recurrencia con **reducción por división**:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

Tendremos:

a: Número de llamadas recursivas = 1, ya que hay una recursividad por cada derivación.
b: Reducción del problema en cada llamada = 2, que corresponde con la división del problema la realizar la búsqueda.
$\boldsymbol{c * n^k}$: Coste de las llamadas externas a la recursividad. Encontramos que el resto de operaciones es constante, es decir, se asumen que son operaciones elementales:

$c * n^k = 1 \Rightarrow k = 0$

Con estos datos pasamos a resolver la recurrencia:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

En este caso, $a = b^k$, cuyo coste es $\theta\left(n^0 * log(n)\right) = \boldsymbol{\theta(log(n))}$.

Ejercicios resueltos de programación 3

Tema 5. Estructuras de datos.

Alumna: Alicia Sánchez
Centro: UNED-Las Rozas (Madrid)

Los ejercicios de este tema no son muchos, no obstante veremos el *índice* como en el resto de temas. Al no existir problemas de este tema, pondremos un nuevo apartado en el que habrá distintas cuestiones sin solución.

Introducción teórica:

Veremos estas nociones de teoría correspondiente con los **montículos**, ya que es muy importante el tenerlo claro, habiendo multitud de ejercicios que se resuelven de esta manera, aunque luego insistamos de nuevo con esta misma teoría en los ejercicios:

Es importantísimo tener bien claro la **propiedad del montículo**:

El nodo i es el padre de $\left\{\begin{matrix} 2*i \\ 2*i+1 \end{matrix}\right\}$.

El nodo i es el hijo del nodo i div 2 (o i $\div$ 2).

En resumen y para los vectores sería:

$\left\{\begin{matrix} T[i] \geq T[2*i] \\ T[i] \geq T[2*i+1] \end{matrix}\right\}$ Hijos con respecto a padres.

$T[i] \leq T[i\ div\ 2]$. Padres con respecto a hijos.

Relacionado con el ejercicio es que tenemos dos posibles algoritmos para crear un montículo. El primero de ellos será:

```
proc crear-montículo-lento(T[1..n])
    para i ← 2 hasta n hacer flotar (T[1..i], i)
fproc
```

Para seguir el recordatorio veremos el algoritmo iterativo de flotar una posición, que será:

```
proc flotar (T[1..n], i)
    k ← i;
    repetir
        j ← k;
        si j > 1 y T[j ÷ 2] < T[k] entonces k ← j ÷ 2
        intercambiar T[j] y T[k]
        { si j = k, entonces el nodo ha llegado a su posición final }
    hasta que j = k
fproc
```

El segundo algoritmo será el siguiente:

```
proc crear-montículo (V[1..N])
    para i ← [n/2] bajando hasta 1 hacer hundir (T, i)
fproc
```

Vimos en la teoría que uno de ellos es ineficiente con respecto a ello, lo que supone que haría más operaciones de *flotar* y aunque tiene coste lineal la constante multiplicativa es mayor. Evidentemente, es el primero de ellos, lo cual no nos valdría para el ejercicio que queremos resolver. Por tanto, escogeremos el algoritmo segundo que hemos escrito, ya que será más eficiente.

Pasamos a escribir el procedimiento para *hundir* un elemento del montículo:

```
proc hundir (T[1..n], i)
    k ← i;
    repetir
        j ← k;
        { Buscar el hijo mayor del nodo j }
        si 2 * j ≤ n y T[2 * j] > T[k] entonces k ← 2 * j
        si 2 * j ≤ n y T[2 * j + 1] > T[k] entonces k ← 2 * j + 1
        intercambiar T[j] y T[k]
        { si j = k, entonces el nodo ha llegado a su posición final }
    hasta que j = k
fproc
```

Estas funciones serán bastantes importantes el saberlos, ya que nos harán falta en numerosos ejercicios. Por lo que serán de gran importancia el sabérselos (no memorizarlos, por supuesto). Como regla nemotécnica (a mí personalmente me vale) el hundir es ir de arriba abajo (como los buzos) y flotar es ir de abajo a arriba. Es necesario el tener claro estas ideas, ya que son críticas en los montículos.

Otra regla nemotécnica importante y que me ha liado mucho es asumir el significado de estas variables:

i: Indica valor inicial que, dada por argumento, la posición que se flota o hunde.
J: Indica la posición que se refrescará tras intercambiar dentro del bucle "repetir".
k: Indica la nueva posición que asumirá el elemento, previo al intercambio, igualmente dentro del bucle "repetir" o ,dicho de otra manera, la posición a la que se intercambiará el elemento una vez que se cumple la propiedad del montículo bien sea con hundir o flotar.

Vemos que se sale del bucle cuando $j = k$, por lo que ya no se podrá intercambiar dos posiciones.

Por último, decir que siempre consideraremos **montículos de máximos** (la raíz es el máximo elemento), salvo cuando nos lo digan expresamente, ante ese caso tendremos **montículos de mínimos**. No queda decir que las operaciones son las mismas, sólo que cambian el signo.

La idea es considerar el vector como un montículo. Empezamos convirtiendo inicialmente en montículos los subárboles del nivel más bajo, para progresivamente seguir modificando en niveles superiores sucesivos.

1ª parte. Cuestiones de exámenes:

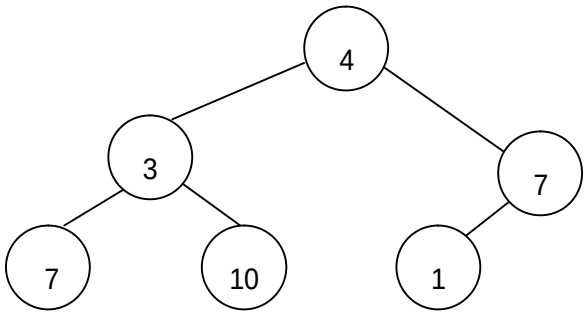
Septiembre 2000-reserva (ejercicio 3)

Enunciado: Explicar cómo funciona el algoritmo que dota a un vector la propiedad del montículo en tiempo lineal (no es necesario demostrar que ese es el coste).

Respuesta: Tenemos en mente los algoritmos vistos en la sección de teoría, siendo uno de ellos ineficiente con respecto al otro, lo que supone que haría más operaciones de *flotar* y aunque tiene coste lineal la constante multiplicativa sería mayor. Evidentemente, es el primero de ellos, lo cual no nos valdría para el ejercicio que queremos resolver. Por tanto, escogeremos el algoritmo segundo que hemos escrito, ya que será más eficiente.

```
proc crear-montículo (V[1..N])
    para i ← [n/2] bajando hasta 1 hacer hundir (T, i)
fproc
```

Veremos el funcionamiento con un **ejemplo**. Se nos da este vector V = [4,3,7,7,10,1]. En forma de árbol será:

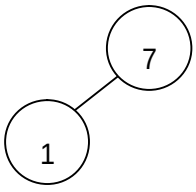


Vemos que al ser el elemento en la posición 2 no cumple la propiedad del montículo de máximos, con lo que aplicaremos el procedimiento antes visto.

Tenemos estos pasos:

1er paso. Empezaremos por $i = 3$, es decir, por la posición 3 en el montículo.

Hacemos una llamada a hundir $(V, 3)$. Tendremos que hundir esta posición para ordenarla. Tendremos este subárbol:

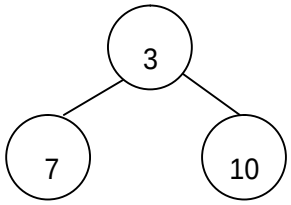


Vemos, por tanto, que cumple la propiedad del montículo, en la que el padre es mayor o igual que el hijo, por lo que no habrá ningún intercambio de posiciones.

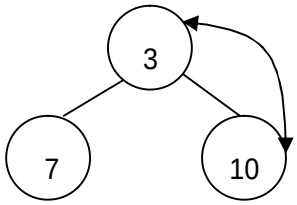
El vector no se modifica.

2º paso. Continuaremos por $i = 2$.

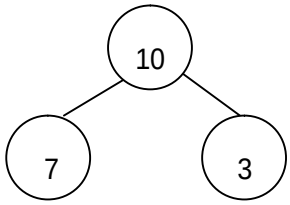
Haremos llamada a hundir $(V, 2)$. Veremos de nuevo el subárbol para así ver qué posición hay que hundir:



Observamos que el nodo padre es menor que los hijos por lo que **NO** cumple la propiedad del montículo. Siguiendo el algoritmo hay que intercambiarlo con el hijo mayor, por lo que intercambiamos estas posiciones:



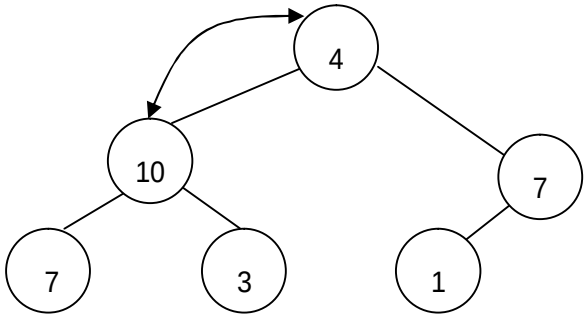
Con lo que el resultado quedaría así:



El vector tras este paso es $V = [4,10,7,7,3,1]$.

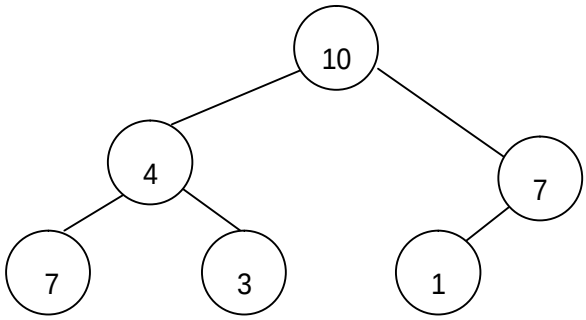
3er paso. Seguimos por $i = 1$ o lo que es lo mismo la raíz.

Hacemos llamada a hundir $(V, 1)$. Nuestro subárbol será:



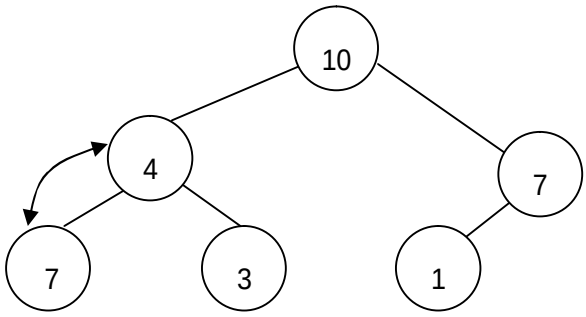
Vemos que el nodo valor 4 (nuestra raíz) es menor que cualquiera de sus hijos, por lo que a priori no cumple la propiedad del montículo. Seguiremos la misma filosofía de antes e intercambiamos con el hijo mayor, que de nuevo es la posición del vector 2.

Tras intercambiarlo, el montículo quedará:

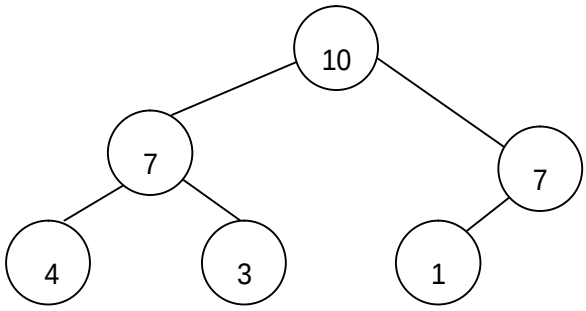


Y el vector será $V = [10,4,7,7,3,1]$.

Viéndolo de nuevo, observamos que sigue sin cumplirse la propiedad del montículo. Nuestro algoritmo de *hundir* seguirá por la posición 2, que es la del último intercambio. Vemos, por tanto, que no cumple la propiedad del montículo de nuevo en dicha posición, por lo que se intercambia con el hijo mayor, como sigue:



El montículo resultante del intercambio será:



En este caso, vemos que ya cumple la propiedad del montículo en todo el árbol. Por tanto, ya se ha creado un montículo y el vector resultante será:

$V = [10,7,7,4,3,1]$.

Nótese varios aspectos importantes:

- Al intercambiar con la posición 2 observamos que el subárbol derecho se deja sin explorar. Esto significa que ya cumple la propiedad previamente analizada en el primer paso (cuando i=3).
- Hemos seguido el planteamiento utilizando árboles, pero tendremos presente que una de las representaciones más utilizadas es con vectores. Dicho esto, podríamos haber hecho esto mismo de forma grafica usando vectores y viendo así las posiciones de intercambio.

NOTA DEL AUTOR: Este ejercicio está tomado de otro ya resuelto en parte y hecho por mí, que soy una alumna, por lo que no es seguro que esté completamente bien.

Febrero 2001-1ª (ejercicio 1)

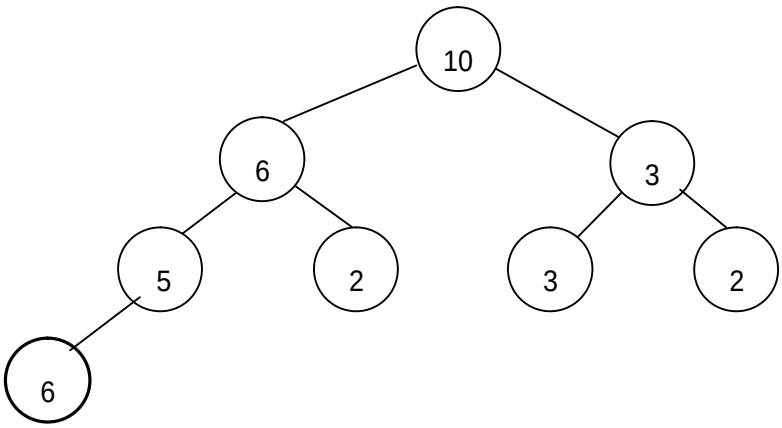
Enunciado: Dado el siguiente montículo [10,6,3,5,2,3,2] se pide insertar el valor 6 describiendo toda la secuencia de cambios en el mismo.

Respuesta: Tendremos el algoritmo de inserción siguiente:

proc añadir-nodo $(T[1..n], v)$
  $T[n+1] \leftarrow v$;
  flotar $(T[1..n+1], n+1)$
fproc

Observamos que se usa la función *flotar* (de abajo a arriba) para que al insertar el nuevo nodo se recupere la propiedad del montículo. En nuestro caso, tenemos que añadir un nodo al montículo [10,6,3,5,2,3,2].

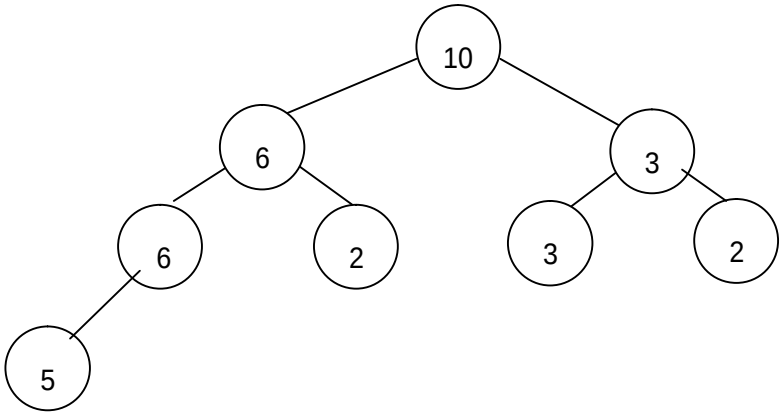
Pasamos como en el ejercicio anterior el vector a un árbol:



El nuevo vector tras insertar el elemento es [10,6,3,5,2,3,2,6]

Resaltamos el nodo insertado en negrita. Vemos que habrá que hacer una llamada a la función flotar según el procedimiento anterior, que será flotar $(T[1..8], 8)$. Como antes, seguiremos estos pasos, que se harán en la misma llamada de *flotar* (no confundir con el ejercicio anterior, que eran distintas llamadas a las funciones).

1. Se compara $T[8]$ con su padre, que es $T[8\ DIV\ 2] = T[4]$. Como es mayor se intercambia resultando:



El vector, por el momento es [10,6,3,6,2,3,2,5].

2. Se comparará, en este caso, $T[4]$ con su padre, que recordemos es $T[4\ DIV\ 2] = T[2]$ y observamos que ambos son iguales. Por tanto, el bucle finalizará, al coincidir j con k y ya será montículo.

Febrero 2001-2ª (ejercicio 3) (parecido a ejercicio 1 de Dic. 02)

Enunciado: Dado un montículo $T[1..n]$, programar completamente en pseudocódigo una función recursiva flotar (T, i) para flotar el elemento de la posición i del vector T. Explicar cómo usar esta función para insertar un elemento en el montículo.

Respuesta: Para realizar la primera parte del ejercicio tendremos que recordar el algoritmo iterativo de *flotar*:

```
proc flotar (T[1..n], i)
    k ← i;
    repetir
        j ← k;
        si j > 1 y T[j ÷ 2] < T[k] entonces k ← j ÷ 2
        intercambiar T[j] y T[k]
        { si j = k, entonces el nodo ha llegado a su posición final }
    hasta que j = k
fproc
```

Por lo que, casi siendo una trascripción directa tendremos el algoritmo recursivo de *flotar*:

```
proc flotar-rec (T[1..n], i)
    si T[i] > T[i DIV 2] entonces
        intercambiar T[i] y T[i DIV 2]
        flotar-rec (T, i DIV 2)
    fsi
fproc
```

La segunda parte de este ejercicio es exactamente igual al anterior, por lo que evitamos dar más detalles del mismo.

Febrero 2002-1ª (ejercicio 3) (Igual a ejercicio 2 Sept. 01-reserva)

Enunciado: Programar en pseudocódigo un algoritmo recursivo para la operación de hundir un elemento en un montículo.

Respuesta: Tendremos al igual que en el ejercicio anterior que partir del algoritmo iterativo de *hundir*. Recordamos, de nuevo, que es:

```
proc hundir (T[1..n], i)
    k ← i;
    repetir
        j ← k;
        { Buscar el hijo mayor del nodo j }
        si 2 * j ≤ n y T[2 * j] > T[k] entonces k ← 2 * j
        si 2 * j ≤ n y T[2 * j + 1] > T[k] entonces k ← 2 * j + 1
        intercambiar T[j] y T[k]
        { si j = k, entonces el nodo ha llegado a su posición final }
    hasta que j = k
fproc
```

Intentaremos razonar de modo similar a como hemos hecho en el algoritmo recursivo de *flotar*, pero nos topamos con un inconveniente, por el que tendremos que averiguar cuál de los dos hijos es el mayor, es decir, el de mayor valor, por lo que deberemos almacenar en una variable temporal el valor del primero hijo para compararlo con el segundo e intercambiar el mayor. Veíamos de modo práctico esta explicación previamente.

```
proc hundir-rec (T[1..n], i)
    hmayor ← i;
    { Buscar el hijo mayor del nodo i }
    si (2 * i ≤ n) y (T[2 * i] > T[hmayor]) entonces
        hmayor = 2 * i;
    fsi
    si (2 * i + 1 ≤ n)  y (T[2 * i + 1] > T[hmayor]) entonces
        hmayor = 2 * i + 1;
    fsi
    { Si cualquier hijo es estrictamente mayor que el padre }
    si (hmayor > i) entonces
        intercambiar T[i] y T[hmayor]
        hundir-rec (T, hmayor)
    fsi
fproc
```

Febrero 2002-1ª (ejercicio 3)

Enunciado: Programar en pseudocódigo todos los procedimientos necesarios para fusionar dos conjuntos implementados con montículos. Suponer que los conjuntos no pueden tener elementos repetidos.

Respuesta: Este ejercicio no tengo ni idea de cómo se puede resolver, sólo sé que pueden pedir los códigos que se han visto en teoría de los montículos, pero en este caso piden fusionar dos conjuntos implementados con montículos, lo cual desconozco. He buscado en internet, en el libro, en muchos sitios más que solución tiene este ejercicio y no he podido llegar a ninguna conclusión.

Diciembre 2003 (ejercicio 2)

Enunciado: ¿Qué diferencias hay entre un montículo y un árbol binario de búsqueda?

Respuesta: Para responder a esta pregunta vamos a definir que es cada uno de ellos y así se verá más adecuadamente sus diferencias. No hay solución oficial de esta pregunta, por lo que se ha tomado apuntes del libro base casi directamente (*Fundamentos de Algoritmia* de G. Brassard y P. Bratley). Pasamos, pues, a las definiciones y a continuación a las diferencias:

Un **árbol de búsqueda** es aquél en el que el valor contenido en todos los nodos internos es mayor o igual que los valores contenidos en su hijo izquierdo o en cualquiera de los descendientes de ese hijo y menor o igual que los valores contenidos en su hijo derecho o en cualquiera de los descendientes de ese hijo.

El **montículo** es un árbol binario esencialmente completo, cada uno de cuyos nodos incluye un elemento de información denominado valor del nodo y que tiene la propiedad consistente en que el valor de cada nodo interno es mayor o igual que los valores de sus hijos. Esto se llama **propiedad del montículo**.

Al eliminar o añadir nodos al árbol binario de búsqueda se puede volver *desequilibrado*, con lo que muchos nodos pueden tener un solo hijo, así que sus ramas se vuelven largas y delgadas. No resulta eficiente hacer búsquedas en el árbol desequilibrado, ya que el coste es lineal, porque hay que recorrer todos los nodos. Para equilibrarlo requiere un coste de $O(\log n)$, en el caso peor, siendo n el número de nodos que hay en el árbol.

Al igual que pasa con el árbol de búsqueda al quitar nodos o añadirlos en un **montículo** hay que restaurar la propiedad del montículo, lo que implica hundir o flotar, cuyo coste es $O(\log n)$.

Observamos cómo hemos puesto en ocasiones anteriores que tomaremos la cota superior, en vez del coste exacto, esto lo haremos porque al fin y al cabo el coste exacto es una unión de cota superior y cota inferior. En el montículo será coste exacto, pero lo dejaremos así.

Septiembre 2004 (ejercicio 2)

Enunciado: Implementar una versión recursiva de una función que tome un vector y le dé estructura de montículo.

Resultado:

Tendremos que hacer la versión recursiva de crear montículo. Hemos tomado de nuevo la solución que ponen (no sabemos si es oficial o no) con algunas modificaciones propias, aunque nos fijamos que el algoritmo que han tomado es el de crear montículo "lento", en el que hay más llamadas.

Recordemos la función *iterativa* de crear montículo:

```
proc crear-montículo-lento (T[1..n])
    para i ← 2 hasta n hacer flotar (T[1..i], i)
fproc
```

La función *recursiva* de crear montículo será, por tanto, la siguiente:

```
proc crear-montículo-rec (T[1..n], i)
    si i > n entonces
        devuelve (T)
    si no
        flotar (T, i)
        crear-montículo-rec (T, i+1)
    fsi
fproc
```

Escribiremos la función recursiva de flotar, aunque lo hemos escrito previamente. Es otra solución posible, por lo que es interesante tenerla en cuenta:

```
proc flotar-rec (T[1..n]:tipo monticulo,VAR i:integer)
    VAR i_padre: integer;
    i_padre = i div 2;
    si (i > 1) and T[i] > T[i_padre] entonces
        intercambiar T[i] y T[i_padre]
        flotar-rec (T, i _padre)
    fsi
fproc
```

La llamada inicial de la función de creación de montículo será:

```
crear-montículo-rec (T, 2)
```

Otra versión es sin usar funciones auxiliares es sustituir el código de flotar por ser versión iterativa e insertarla en la función anterior.

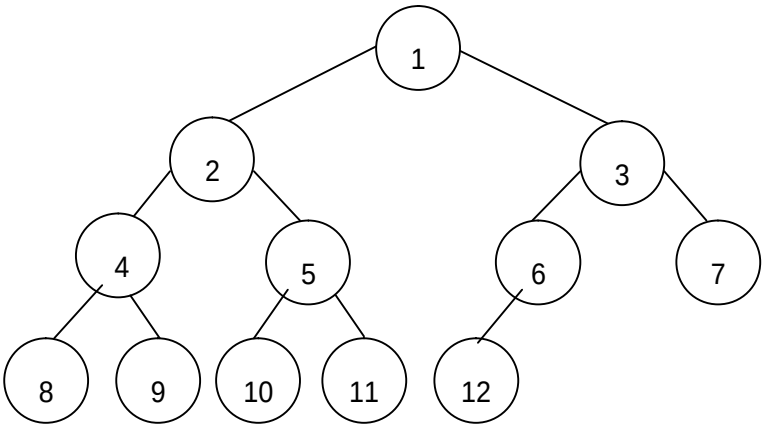
Septiembre 2004-reserva (ejercicio 2)

Enunciado: Sea $T[1..12]$ una matriz tal que $T[i]$ para todo $i \leq 12$. Crear un montículo en tiempo lineal, especificando cada paso y mostrando en todo momento el estado de la matriz T.

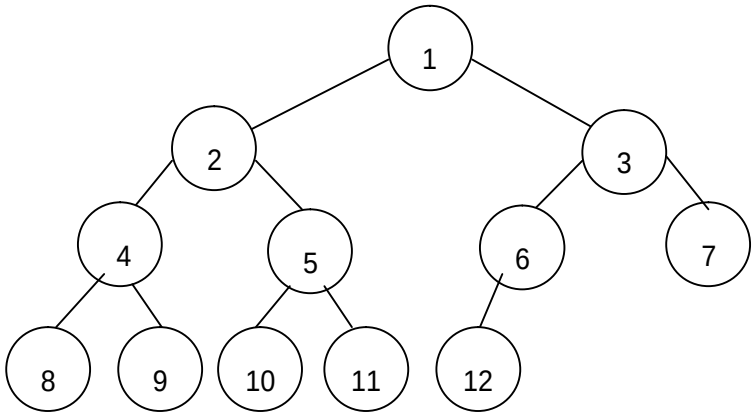
Resultado:

Hemos visto en ejercicios anteriores los distintos algoritmos con los que se resuelven estos ejercicios, por tanto, resolveremos el problema especificando paso a paso:

Inicialmente, el montículo será:

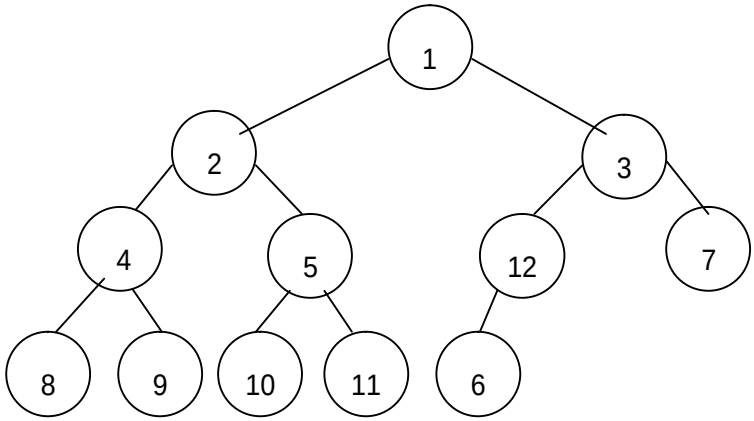


1er paso: Empezaremos por la mitad del montículo, es decir, con $i = 6$:



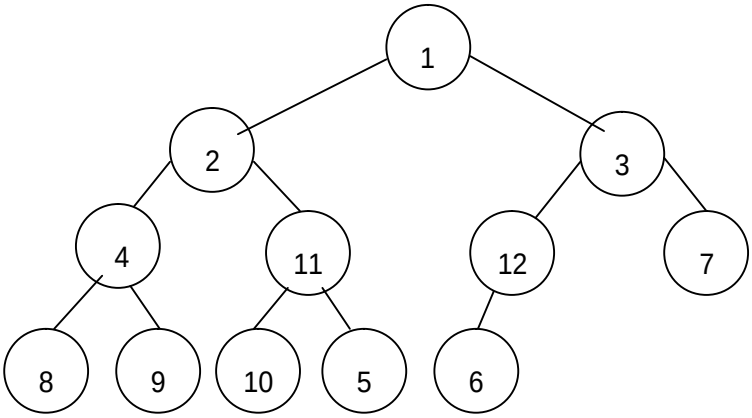
Intercambiamos la posición 6 con la 12, porque al usar el algoritmo de crear montículo rápido, hundiremos (en vez de flotar).

2º paso: Seguiremos por $i = 5$:



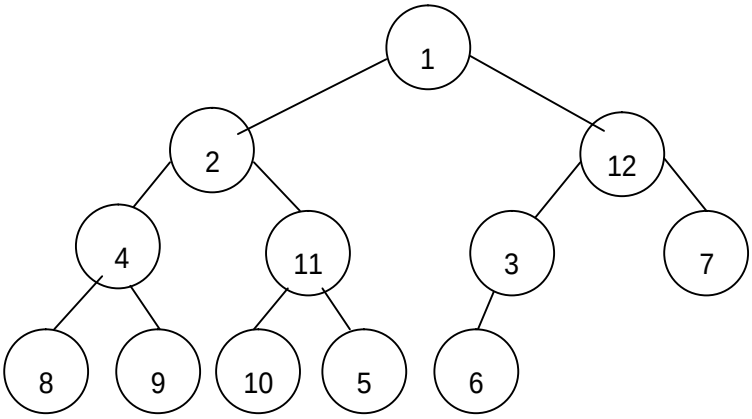
En este caso, intercambiaremos el valor de 5 con el hijo mayor, que es 11. En este caso, por el momento, coincidirán las posiciones con los valores.

3er paso: Seguiremos con $i = 4$, que será el valor 4 y se nuevo hundimos ese valor hasta que cumpla la propiedad del montículo en el árbol:



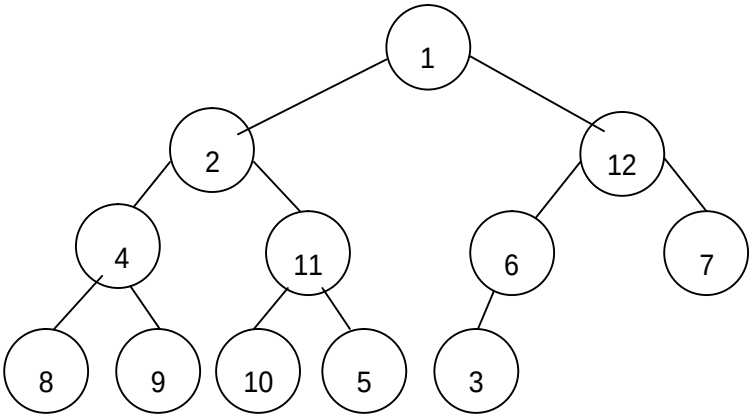
Intercambiaremos de nuevo con el hijo mayor, que es 9.

4º paso: Nos vamos a $i = 3$ y observamos en el grafo anterior que tendremos que intercambiarlo con la posición 6, es decir, el valor 12. Tendremos, por tanto,

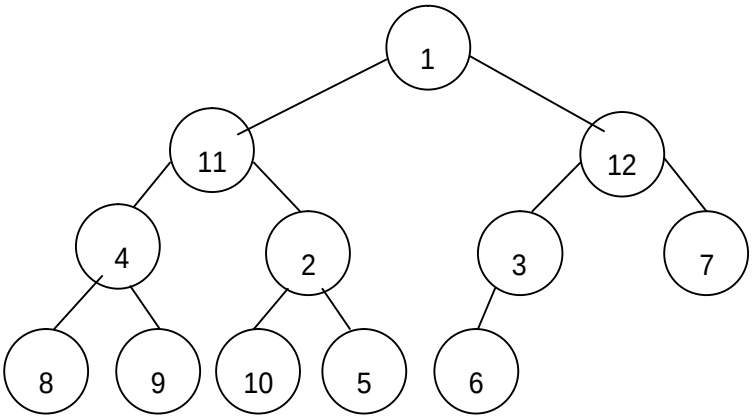


De nuevo, vemos que al intercambiarlo no se cumple la propiedad del montículo, por lo que de nuevo intercambiamos 3 con 6 (posición 6 con la 12, al tener solo un hijo). Lo haremos en la misma llamada a hundir, mucho cuidado con eso.

5º paso: El penúltimo paso será aquel en que $i = 2$.

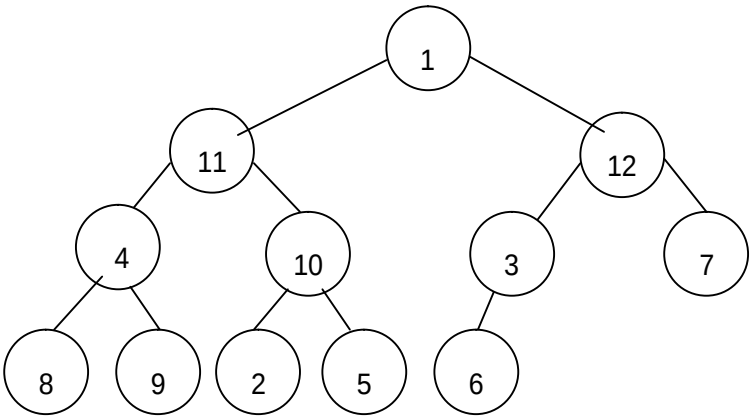


Intercambiaremos la posición 2 con la 5, que es su hijo mayor, resultando:

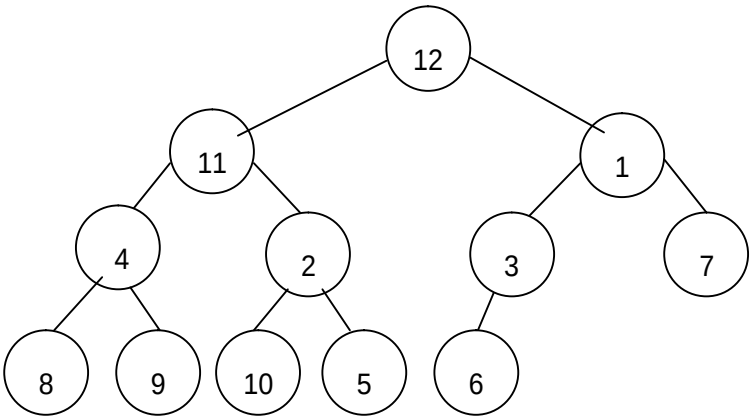


Al igual que el paso anterior necesitaremos hundir de nuevo este valor intercambiado, por lo que será la posición 5 con la 10, su hijo mayor.

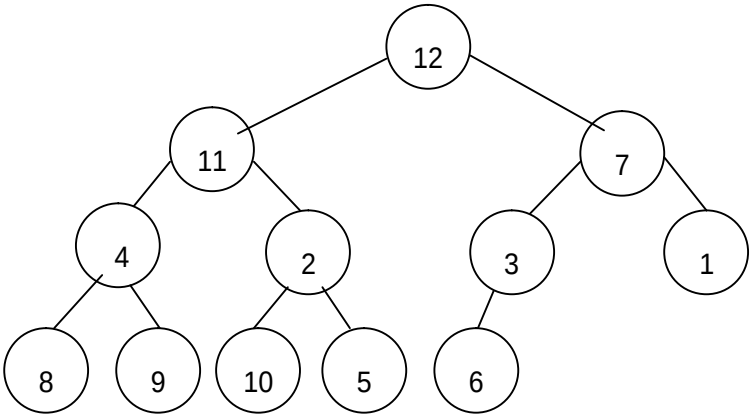
6º paso y último: Estamos en $i = 1$, que es el nodo raíz, por lo que tendremos que ver este árbol:



Intercambiaremos la posición 1 con la 3 (el valor 1 con el 12), quedando:



Vemos de nuevo que no cumple la propiedad del montículo, por lo que intercambiaremos la posición 3 con la 7 (el valor 1 con el 7), quedando el montículo completamente creado así:



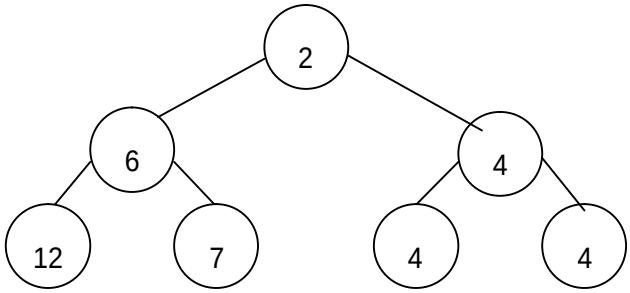
No hemos puesto los vectores intermedios tras hundir los elementos, pero pondremos el vector final, que será [12,11,7,4,2,3,1,8,9,10,5,6].

Septiembre 2005-reserva (ejercicio 1)

Enunciado: Dado m = [2,6,4,12,7,4,4]. Comprobar si es o no montículo de mínimos. Si no lo es, programar una función para convertirlo en montículo y aplicarlo a 'm'. Si no, programar una función de añadir elemento mediante la función "flotar" y aplicarlo al valor 3. En ambos casos, escribir el montículo resultante y detallar todos los pasos.

Resultado:

Verificaremos si es montículo de mínimos, que recordemos que es aquél en el que la raíz es el menor elemento. Tendremos en forma de árbol lo siguiente:



Verificaremos que cumple la propiedad del montículo en todo ello, que será

$$\left.\begin{matrix} T[i] \leq T[2 * i] \\ T[i] \leq T[2 * i + 1] \end{matrix}\right\} \text{ Hijos con respecto a padres.}$$

$$T[i] \geq T[i \; div \; 2]. \quad \text{Padres con respecto a hijos.}$$

Observaremos que cumple la propiedad del montículo en todo el árbol. Por tanto, ya es correcto.

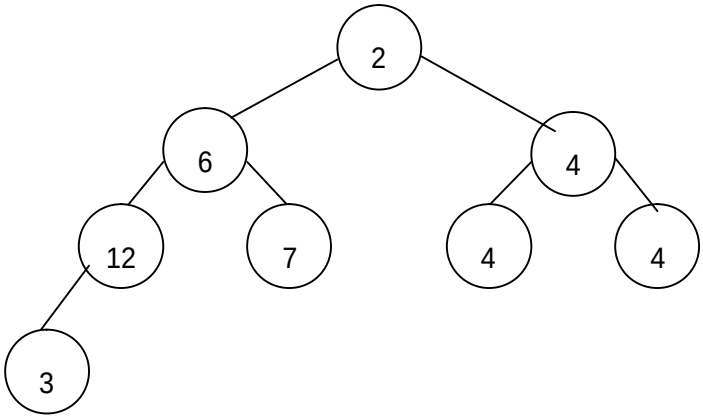
Añadimos un elemento al final del montículo. Recordemos el algoritmo de *añadir-nodo*:

```
proc añadir-nodo (T[1..n], v)
    T[n + 1] ← v;
    flotar (T[1..n + 1], n + 1)
fproc
```

En nuestro caso, variaremos conforme al montículo de mínimos la función *flotar*, como sigue:

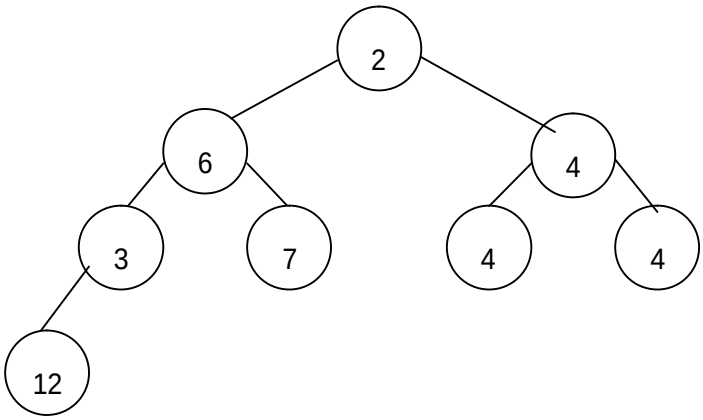
```
proc flotar (T[1..n], i)
    k ← i;
    repetir
        j ← k;
        si j > 1 y T[j ÷ 2] > T[k] entonces k ← j ÷ 2
        intercambiar T[j] y T[k]
        { si j = k, entonces el nodo ha llegado a su posición final }
    hasta que j = k
fproc
```

Al añadir el elemento 3, tendremos este árbol, para convertirlo en montículo:



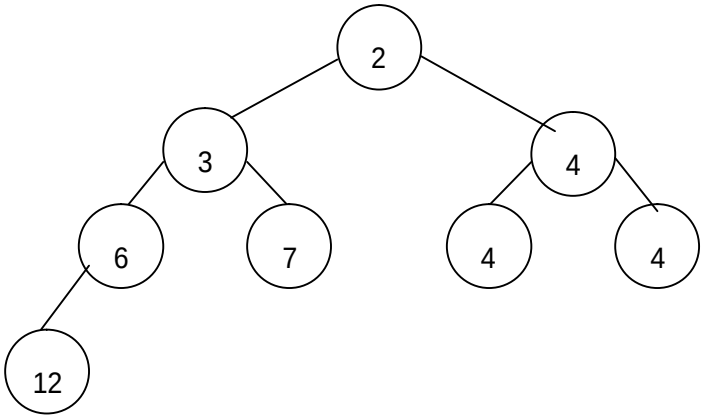
Observamos como en los ejercicios anteriores, que no cumple la propiedad del montículo, por lo que hay que flotar ese elemento (de abajo a arriba, ojito). Haremos estos pasos:

1º paso. Intercambiamos 3 con 12 (posición 8 con 4):



De nuevo no cumple la propiedad del montículo, por lo que tendremos que hacer otro paso más.

2º paso. Intercambiamos 3 con 6 (posición 4 con 2):



Vemos que al acabar esta llamada a flotar ya queda restaurada la propiedad del montículo en todo el árbol, por lo que se da por finalizado el ejercicio.

Febrero 2008-2ª (ejercicio 2)

Enunciado: Sea $T[1..n]$ con k elementos $(k < n)$ un montículo de mínimos. Se pide programar una función "flotar" que dado un nuevo elemento $T[k + 1]$ restaure la propiedad del montículo en T. Una función iterativa que lo resuelva puntuará cero puntos.

Respuesta:

Tendremos la siguiente función que se asemeja enormemente a la dada en el ejercicio 3 de Febrero de 2001-2ª semana. Lo vamos a tratar, ya que es interesante verlo con más calma:

```
proc flotar-rec (T[1..n]:tipo monticulo,VAR i:integer)
    VAR i_padre: integer;
    i_padre = i div 2;
    si (i > 1) and T[i] < T[i_padre] entonces
        intercambiar T[i] y T[i_padre]
        flotar-rec (T, i _padre)
    fsi
fproc
```

Nos fijamos que con respecto a los algoritmos antes puestos, sólo cambiará en el signo >, puesto antes y ahora al ser un montículo de mínimos será <.

Ejercicios resueltos de programación 3

Tema 6. Algoritmos voraces.

Alumna: Alicia Sánchez
Centro: UNED-Las Rozas (Madrid)

En estos ejercicios se incluyen tanto las cuestiones como los problemas de exámenes, por lo que lo distinguiremos en tres partes. Aprovechamos para poner el *índice* donde estarán estas partes, por ser muy extenso el documento:

Introducción teórica:

Antes de resolver estas cuestiones y problemas recordaremos la teoría más general de los algoritmos voraces, como hemos hecho en temas anteriores, ya que nos serán muy útiles para resolverlos.

Empezaremos a ver los algoritmos voraces, ya que son los más fáciles de ver. Resultan **fáciles de inventar e implementar** y cuando funcionan son **muy eficientes**. Sin embargo, hay muchos problemas que no se pueden resolver usando el enfoque voraz.

Los algoritmos voraces se utilizan típicamente para resolver problemas de optimización. Por ejemplo, la búsqueda de la recta más corta para ir desde un nodo a otro a través de una red de trabajo o la búsqueda del mejor orden para ejecutar un conjunto de tareas en una computadora.

Un algoritmo voraz **nunca reconsidera su decisión**, sea cual fuere la situación que pudiera surgir más adelante.

Generalmente, los algoritmos voraces y los problemas que éstos resuelven se caracterizan por la mayoría de propiedades siguientes:

- Son adecuadas para **problemas de optimización**, tal y como vimos en el ejemplo anterior.
- Para construir la solución de nuestro problema disponemos de un **conjunto (o lista) de candidatos**. Por ejemplo, para el caso de las monedas, los candidatos son las monedas disponibles, para construir una ruta los candidatos son las aristas de un grafo, etc. A medida que avanza el algoritmo tendremos estos conjuntos:
    - Candidatos considerados y seleccionados.
    - Candidatos considerados y rechazados.

Las **funciones empleadas** más destacadas de este esquema son:

1. **Función de solución**: Comprueba si un cierto conjunto de candidatos constituye una solución de nuestro problema, ignorando si es o no óptima por el momento. Puede que exista o no solución.
2. **Función factible**: Comprueba si el candidato es compatible con la solución parcial construida hasta el momento; esto es, si existe una solución incluyendo dicha solución parcial y el citado candidato.
3. **Función de selección**: Indica en cualquier momento cuál es el más prometedor de los candidatos restantes, que no han sido seleccionados ni rechazados. Es la más importante de todas.
4. **Función objetivo**: Da el valor de la solución que hemos hallado: el número de monedas utilizadas para dar la vuelta, la longitud de la ruta calculada, etc. Está función no aparece explícitamente en el algoritmo voraz.

Para resolver nuestro problema, buscamos un conjunto de candidatos que constituyan una solución y que optimice (maximice o minimice, según los casos) el valor de la función objetivo. Los algoritmos voraces avanzan paso a paso:

- Inicialmente, el conjunto de elementos seleccionados está vacío y el de solución también lo está.
- En cada paso, se considera añadir a este conjunto el mejor candidato sin considerar los restantes, estando guiada nuestra selección por la **función de selección**. Se nos darán estos casos:

    1. Si el conjunto ampliado de candidatos seleccionados ya no fuera factible (no podemos completar el conjunto de solución parcial dado por el momento), **rechazamos** el candidato considerado por el momento y no lo volvemos a considerar.
    2. Si el conjunto aumentado sigue siendo factible, entonces **añadimos** el candidato actual al conjunto de candidatos seleccionados. Cada vez que se amplía el conjunto de candidatos seleccionados comprobamos si este constituye una solución para nuestro problema. Se quedará en ese conjunto para siempre.

Cuando el algoritmo voraz funciona correctamente, la primera solución que se encuentra es la óptima.

El **esquema voraz** es el siguiente:

```
funcion voraz (C: Conjunto): conjunto
    { C es el conjunto de candidatos }
    S ← ∅          { Construimos la solución en el conjunto S }
    mientras C ≠ 0 y ¬solución (S) hacer
        x ← seleccionar (C)
        C ← C\{x}
        si factible (S ∪ {x}) entonces S ← S ∪ {x}
    si solución (S) entonces devolver S
    si no devolver "no hay solución"
```

La función de selección suele estar relacionada con la función objetivo. Por ejemplo, si estamos intentando maximizar nuestros beneficios, es probable que seleccionemos aquel candidato restante que posea mayor valor individual. En ocasiones, puede haber varias funciones de selección plausibles.

1ª parte. Cuestiones de exámenes:

Febrero 2000-1ª (ejercicio 3)

Enunciado: ¿Qué variante del problema de la mochila admite solución voraz, y por qué? ¿Qué variante no admite solución voraz? Poner un ejemplo del segundo caso en el que la solución voraz no nos lleve a la solución voraz.

Respuesta:

El problema de la mochila se puede enunciar de esta manera:

Maximizar $\sum_{i=1}^{n} x_i * v_i$ con la restricción $\sum_{i=1}^{n} x_i * w_i \leq W$

Teniendo en cuenta que los objetos son ilimitados y que cada uno tiene asociado un valor positivo y un peso positivo. Queremos llevar la mochila respetando la limitación de la mochila W. Veremos las variantes del problema en cuestión a continuación.

La variante de la mochila que admite solución voraz es la variante donde **podemos fragmentar los objetos**, siendo el número de objetos ilimitado. Esta variante admite solución voraz porque encontramos una función de selección que nos permite escoger un candidato a cada paso, de forma que obtengamos una **solución óptima**. Dicha función consiste en escoger los objetos por orden decreciente (de mayor a menor) según su relación valor/peso, lo que nos lleva a una solución óptima.

La variante de la mochila que *no* admite solución voraz es aquélla en la que **no se nos permite fragmentar los objetos**, aunque sea ilimitado como antes. Para ello, pondremos un contraejemplo, que no llega a solución óptima. Tenemos que la mochila tiene un peso máximo de $W = 10$, por lo que el valor y peso de los distintos objetos será:

i	1	2
v_i	8	5
w_i	6	5
$\frac{v_i}{w_i}$	1.33	1

Según el algoritmo voraz escogeremos los objetos por orden decreciente en función de la relación valor/peso. Sin embargo, como en esta ocasión no podemos fragmentar los objetos, al introducir el objeto 1 de peso 6, siendo menor que 10, ya no podremos introducir más. Habríamos llegado a solución óptima si escogemos dos objetos número 2, que llenarían la mochila y tendría valor 10 sin sobrepasar el peso máximo de la mochila ($W = 10$).

Vemos en este contraejemplo que pese a seguir la función de selección del algoritmo voraz en esta segunda variante no llegamos a solución óptima, por lo que deducimos que la primera variante llega siempre a solución óptima, por poder fraccionarse los objetos.

Septiembre 2000-reserva (ejercicio 2)

Enunciado: Los algoritmos de Prim y Kruskal calculan, de distintas formas, el árbol de recubrimiento mínimo de un grafo. ¿Cuál de los dos es más eficiente para un grado de alta densidad de aristas?

Respuesta: El algoritmo de Kruskal tiene un coste algorítmico $\theta(a * \log(n))$, siendo a el número de aristas, mientras que el de Prim tiene coste de $\theta(n^2)$, independientemente del número de aristas.

Si la densidad del grafo es alta (**grafo denso**) *a* tiende a $\frac{n*(n-2)}{2}$, por lo que el coste del algoritmo de Kruskal es $\theta(n^2 * \log(n))$, quedando igual el coste del algoritmo de Prim: $\theta(n^2)$. Si la densidad del grafo es baja (**grafo disperso**) *a* tiende a n, por lo que el coste del algoritmo de Kruskal sería $\theta(n * \log(n))$, quedándose el de Prim: $\theta(n^2)$.

En el ejercicio se nos pregunta sobre la comparación de costes en grafo denso, por lo que sería, en este caso, más eficiente el de Prim con coste $\theta(n^2)$ en comparación con el de Kruskal $\theta(n^2 * \log(n))$.

Como añadido decir que si se compararan los costes para los grafos dispersos, el más eficiente sería el de Kruskal con coste $\theta(n * \log(n))$ en comparación con el de Prim $\theta(n^2)$.

Por último, si implementamos el algoritmo de Prim usando montículos invertidos (o de mínimos) tendría coste algorítmico de $\theta(a * \log(n))$, igualándose al de Kruskal.

Febrero 2001-2ª (ejercicio 1)

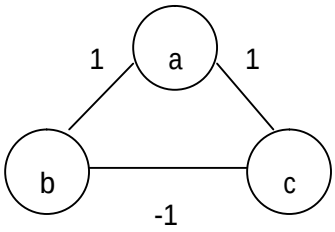
Enunciado: Mostrar mediante un ejemplo que el algoritmo de Dijkstra no siempre encuentra el camino mínimo si existe una distancia negativa.

Respuesta: Recordemos que el algoritmo de Dijkstra devuelve los caminos mínimos desde el origen hasta cada uno de los demás nodos. El pseudocódigo es el siguiente:

```
funcion Dijkstra (L[1..n, 1..n]): matriz[2..n]
    matriz D[2..n]
    { Iniciación }
    C ← {2, 3,.., n}        { S = N/C sólo existe implícitamente }
    para i ← 2 hasta n hacer D[i] ← L[1, i]
    { Bucle voraz }
    repetir n − 2 veces
        v ← algún elemento de C que minimiza D[v]
        C ← C\{v}        { e implícitamente S ← S ∪ {v} }
        para cada w ∈ C hacer
            D[w] ← min(D[w], D[w] + L[v, w]);
    devolver D
```

Haremos dos **ejemplos** distintos para verificar que no encontrara camino mínimo si existe una distancia negativa. El primero de ellos es un ejemplo puesto por el autor, mientras que el segundo es el del propio ejercicio. Pasamos a verlos:

1er ejemplo: Se nos da este grafo:

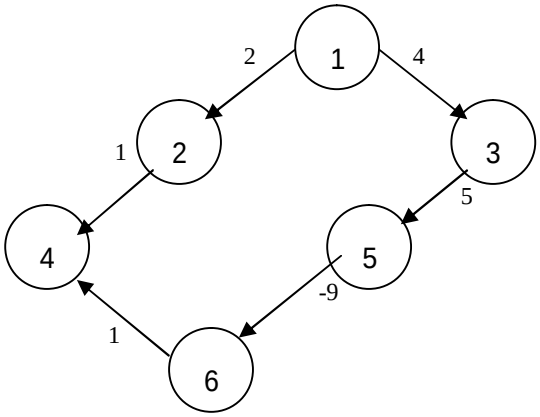


Siguiendo el algoritmo de Dijkstra empezando por el nodo a, tendremos:

Paso	v	C	D	P
			b c	
Inicialización	-	$\{b, c\}$	[1,1]	$[a, a]$
1	b	$\{c\}$	[1,0]	$[a, b]$

Observamos que para ir del nodo a al c tendremos dos caminos el directo con coste 1 y tal y como hemos visto antes pasando a través del nodo b, con coste 0. Por tanto, vemos que es **imposible** que exista un camino con coste 0, por lo que no podemos encontrar el camino mínimo con aristas de distancias negativas (además de ser físicamente imposible tener distancia negativa).

2º ejemplo: Se nos da este grafo:



Siguiendo paso a paso el algoritmo de Dijkstra tendremos:

Paso	v	C	D	P
			2 3 4 5 6	
Inicialización	-	{2,3,4,5,6}	$[2,4,\infty,\infty,\infty]$	$[1,1,-,-,-]$
1	2	{3,4,5,6}	$[2,4,3,\infty,\infty]$	$[1,1,2,-,-]$
2	4	{3,5,6}	$[2,4,3,\infty,\infty]$	$[1,1,2,-,-]$
3	3	{5,6}	$[2,4,3,9,\infty]$	$[1,1,2,3,-]$
4	5	{6}	[2,4,3,9,0]	[1,1,2,3,5]

Observamos que el coste hasta llegar al nodo 6 es 0, siendo de nuevo imposible, porque no es posible llegar a un nodo con coste 0, siendo el resto positivos. Con este nuevo ejemplo, se observa que no llega a hallar los caminos mínimos nunca usando el algoritmo de Dijkstra.

Septiembre 2001 (ejercicio 1)

Enunciado: Comenta de qué formas se puede mejorar la eficiencia del algoritmo de Dijkstra el uso de estructuras de datos adecuada.

Respuesta: Con esta respuesta completaremos la teoría dada en el resumen del tema (pag. 227 del libro de teoría). Utilizando una matriz de adyacencia y matrices para representar el algoritmo, el coste es $O(n^2)$, teniendo en cuenta estas operaciones:

- Inicializar la matriz de adyacencia: $O(n)$.
- Bucle del algoritmo voraz: $\theta(n^2)$.

Si resulta que el número de aristas *a* es mayor que el de nodos al cuadrado ($a > n^2$), resulta apropiado utilizar una lista de adyacencia, evitando examinar entradas innecesarias (donde no existan aristas), ahorrando así tiempo en el bucle más interno del algoritmo.

Por otro lado, podemos utilizar un montículo invertido (la raíz es el elemento menor) para representar el camino mínimo que se irá generando (que llamaremos D), esto hace que buscar un nodo que minimice el coste conlleve estas operaciones:

- Inicializar el montículo: $\theta(n)$.
- Eliminar la raíz del montículo: $O(\log(n))$.

Igualmente, las operaciones empleadas en el bucle más interno del algoritmo reducirán su coste, situándose en $O(\log(n))$.

Si se produce la eliminación de la raíz del montículo (el bucle se ejecuta $n-2$ veces) y hay que flotar un máximo de *a* nodos (siendo *a* el número de aristas) obtenemos un tiempo total de $\theta((a+n)*\log(n))$.
Si el grafo es conexo, es decir, $a \geq n-1$, el tiempo total es $\theta(n*\log(n))$.
Si el grafo es denso será preferible la implementación con matrices, si es disperso es mejor la implementación con montículos.

NOTA DEL AUTOR: Se observa en este ejercicio una incongruencia en los enunciados, en uno pide esto mismo y en otro dice: 'halla el grado de expansión mínimo del grafo de la figura mediante el algoritmo de Kruskal. Detalla cada paso'. No sé cuál es el correcto.

Septiembre 2001 (ejercicio 2)

Enunciado: Explica porqué una estructura de montículo suele ser adecuada para representar el conjunto de candidatos de un algoritmo voraz.

Respuesta: En un algoritmo voraz iremos escogiendo el candidato más apropiado a cada paso para hallar la solución según el valor de la función de selección. Para agilizar esa selección podemos tener dichos candidatos almacenados en un **montículo de mínimos**, de forma que el valor sea el mínimo en la raíz. De este modo, la selección del candidato consistirá en ir escogiendo la cima de dicho montículo y actualizarlo cuando así proceda, operaciones éstas que resultan más eficientes en los montículos que en otros tipos de estructuras de datos.

Septiembre 2001 (ejercicio 3)

Enunciado: Explica en qué consiste un problema de planificación con plazo fijo. Por un ejemplo con $n = 4$ y resuélvelo aplicando el algoritmo correspondiente.

Respuesta: Un problema de **planificación con plazo fijo** consiste en lo siguiente:

Tenemos que ejecutar un conjunto de n tareas, cada una de las cuales requiere un tiempo unitario. En cualquier instante $T = 1, 2, \ldots$ podemos ejecutar únicamente una tarea. La tarea i nos produce unos beneficios $g_i > 0$ sólo en el caso en que sea ejecutada en un instante anterior a d_i.

En resumen:

n: Número de tareas de tiempo unitario. Por ejemplo, una hora, días,...
$T = 1, 2, \ldots, n$ En cada instante solo podemos realizar una tarea.
$\boldsymbol{g_i}$: Beneficio asociado a la tarea i.
$\boldsymbol{d_i}$: Plazo máximo de la tarea i.

El problema consiste en **maximizar el beneficio total**.

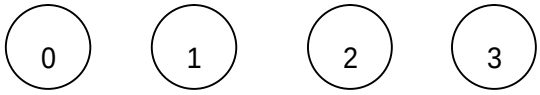
Como añadido al ejercicio del autor, pondremos este ejemplo con $n = 4$ (se resuelve igual que el del libro en la página 241 quitando las dos últimas tareas, pero sirve):

i	1	2	3	4
g_i	20	15	10	7
d_i	3	1	1	3

Hemos ordenado previamente por **orden decreciente** de ganancias.

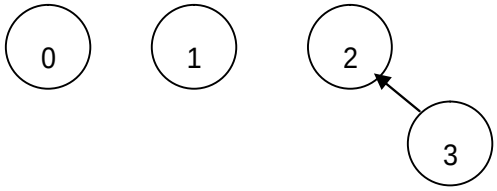
Los pasos son los siguientes:

Inicialmente: $p = min(n, max(d_i)) = min(4,3) = 3$.
Por tanto, como máximo tendremos una planificación de 3 tareas:

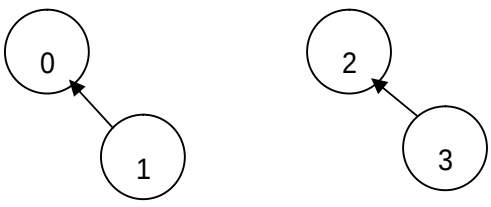


Primer intento: $d_1 = 3$. Se asigna la tarea 1 a la posición 3.

$F(K) = 3$, $F(L) = F(K) - 1 = 2$. Fusionamos K con L

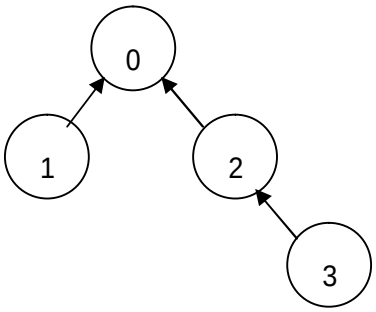


Segundo intento: $d_2 = 1$. Se asigna la tarea 2 a la posición 1.
$F(K) = 1$, $F(L) = F(K) - 1 = 0$. Fusionamos K con L



Tercer intento: $d_3 = 1$. No hay posiciones libres disponibles porque el valor de F es 0.

Cuarto intento: $d_4 = 3$. Se asigna la tarea 4 a la posición 3.
$F(K) = 1$, $F(L) = F(K) - 1 = 0$. Fusionamos K con L



La secuencia óptima es la 2, 4, 1 con valor 42.

Febrero 2002-1ª (ejercicio 1)

Enunciado: Describir la estrategia de selección voraz para el problema de planificación con plazo fijo. Aplicarla a la resolución del siguiente ejemplo detallando cada paso:

i	1	2	3	4	5	6
d_i	4	3	1	1	2	2
g_i	10	30	20	30	50	20

Respuesta: Este ejercicio está hecho completamente por el autor (alumna), así que no aseguro que esté correcto. Emplearemos la estrategia voraz para este tipo de problemas, en la que tomaremos por orden decreciente de ganancias.

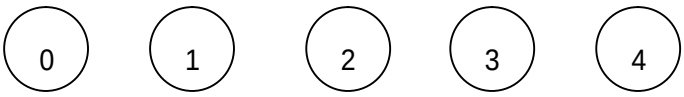
El primer paso es ordenar por orden decreciente de ganancias las tareas, donde añadimos una nueva fila a_i, que indicará la nueva ordenación de las tareas:

i	1	2	3	4	5	6
a_i	5	2	4	6	3	1
d_i	2	3	1	2	1	4
g_i	50	30	30	20	20	10

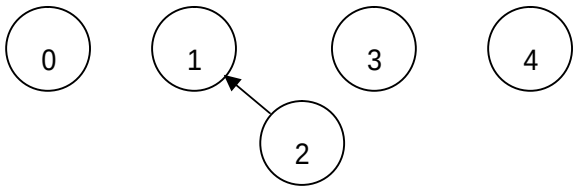
Emplearemos la técnica en la que resolveremos más rápidamente el problema. Para ello, usaremos estructura de partición y haremos lo siguiente:

Inicialmente: $p = min(n, max(d_i)) = min(6,4) = 4$.

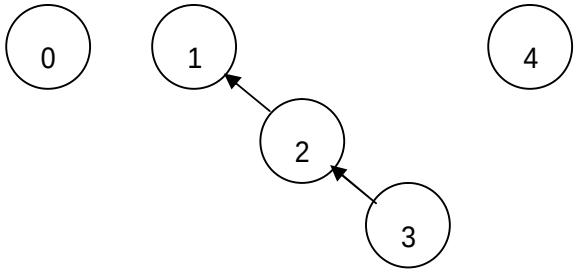
Por tanto, como máximo tendremos una planificación de 4 tareas:



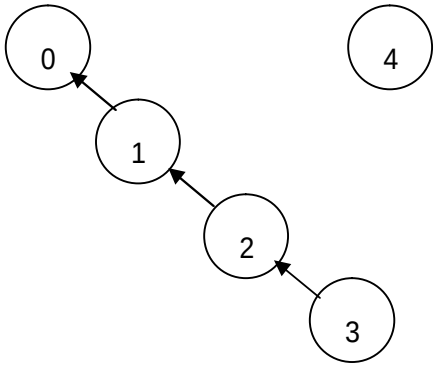
Primer intento: $d_1 = 2$. Se asigna la tarea 1 a la posición 2.
$F(K) = 2, F(L) = F(K) - 1 = 1$. Fusionamos K con L



Segundo intento: $d_2 = 3$. Se asigna la tarea 2 a la posición 3.
$F(K) = 3, F(L) = F(K) - 1 = 2$. Fusionamos K con L



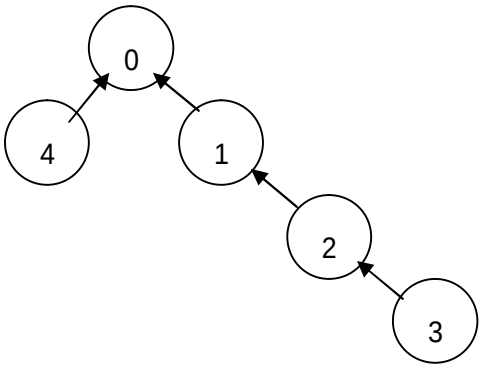
Tercer intento: $d_3 = 1$. Se asigna la tarea 3 a la posición 1.
$F(K) = 1, F(L) = F(K) - 1 = 0$. Fusionamos K con L



Cuarto intento: $d_4 = 2$. No hay posiciones libres disponibles.

Quinto intento: $d_5 = 1$. No hay posiciones libres disponibles.

sexto intento: $d_6 = 4$. Se asigna la tarea 6 a la posición 4.
$F(K) = 4$, $F(L) = F(K) - 1 = 3$. Fusionamos K con L. Nos fijamos que al fusionar K con L cambiamos el puntero de la partición 4 al nodo 0, porque sería el rótulo de la partición con la que se fusiona.



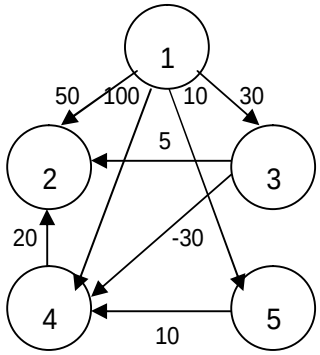
Septiembre 2002 (ejercicio 2)

Enunciado: Aplicar el algoritmo de Dijkstra al grafo dirigido representado por la siguiente matriz de adyacencia:

	1	2	3	4	5
1	-	50	30	100	10
2	-	-	-	-	-
3	-	5	-	-30	-
4	-	20	-	-	-
5	-	-	-	10	-

Tomando el nodo 1 como origen. ¿Encuentra los caminos mínimos? Si la respuesta es negativa, ¿cuáles serían los verdaderos caminos mínimos y porque no la encuentra el algoritmo de Dijkstra? ¿Qué pasaría si se invirtiese el sentido de la arista que une el nodo 3 con el 2?

Respuesta: Daremos una solución por el autor. Según la matriz de adyacencia el grafo, en este caso, dirigido será:



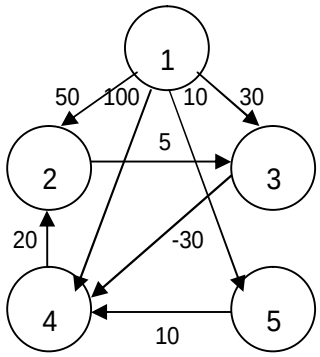
Realizaremos estos pasos para hallar los caminos mínimos, empleando para ello el algoritmo de Dijkstra:

Paso	v	C	D	P
			2 3 4 5	
Inicialización	-	{2,3,4,5}	[50,30,100,10]	[1,1,1,1]
1	5	{2,3,4}	[50,30,20,10]	[1,1,5,1]
2	4	{2,3}	[40,30,20,10]	[4,1,5,1]
3	3	{2}	[20,30,0,10]	[3,1,3,1]

Tras aplicar el algoritmo de Dijkstra observamos que la distancia al nodo 4 es 0, cuando vemos en el grafico hay distancias positivas en el resto de aristas, por tanto, es **imposible** que encuentre caminos mínimos el algoritmo (incluso físicamente distancias negativas lo es igualmente). Tenemos en cuenta que la arista que une del nodo 3 al 4 es negativa (−30), siendo éste el que causa estos problemas.

Por ello, a la *primera pregunta* podremos responder que **no** encuentra caminos mínimos. Los verdaderos caminos mínimos serían los que no pasan a través del nodo 4, es decir, en el último paso puesto sería los que pasan a través del nodo 1, vamos las aristas ({1,3},{1,5}) (apreciación del autor).

En la *segunda pregunta*, si se invirtiese el sentido de la arista que une el nodo 3 con el 2 quedaría el grafo como sigue:



No habría ningún cambio de costes entre aristas. Quedaría igual el algoritmo de Dijkstra (apreciación del alumno).

Febrero 2003-1ª (ejercicio 1)

Enunciado: ¿Para qué se pueden utilizar montículos en el algoritmo de Kruskal? ¿Qué mejoras introduce en términos de complejidad?

Respuesta: Recordamos que el algoritmo de Kruskal era aquél en el que queríamos hallar un árbol de recubrimiento mínimo (con $n-1$ aristas) de un grafo dado, para ello necesitábamos ordenar de modo creciente los costes de las distintas aristas, seleccionándolas sin importar que sean conexos hasta llegar a dicho árbol con todas las componentes conexas.

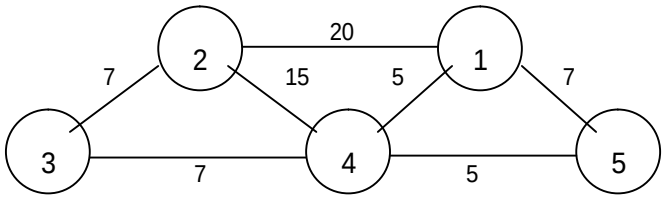
Emplearemos **montículos invertidos o de mínimos** (la raíz es el menor elemento) para ordenar las aristas de menor coste. El coste de esta implementación usando montículos será:

- Inicialización: $\theta(a)$
- Bucle "repetir" hasta que las $n-1$ aristas formen árbol de recubrimiento mínimo será: $\theta(a * \log(n))$, siendo a el número de aristas.

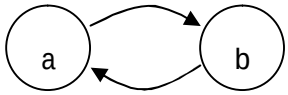
Por tanto, tendremos que el coste del algoritmo será más ventajoso usando montículos cuando el grafo es disperso, es decir, cuando a tiende a n, siendo el coste $\theta(n * \log(n))$. En tales casos, el algoritmo original desperdicia el tiempo ordenando estas aristas inútiles, aunque asintóticamente tenga el mismo coste.

Febrero 2003-1ª (ejercicio 2)

Enunciado: Dado el grafo de la figura, aplicar el algoritmo de Dijkstra para hallar los caminos más cortos desde el nodo 1 hasta cada uno de los demás nodos, indicando en cada paso nodos seleccionados, nodos no seleccionados, vector de distancias y vector de nodos precedentes.



Respuesta: Vemos en el grafo que las aristas son no dirigidas, recordemos que aunque el algoritmo de Dijkstra está diseñado para aristas dirigidas, éstas equivalen a:



Aplicaremos estos mismos conceptos para aplicar el **algoritmo de Dijkstra** en el grafo anterior, tomando el nodo 1 como nodo origen:

Paso	v	C	D	P
			2 3 4 5	
Inicialización	-	{2,3,4,5}	[20, ∞, 5,7]	[1, –,1,1]
1	4	{2,3,5}	[20,12,5,7]	[1,4,1,1]
2	5	{2,3}	[20,12,5,7]	[1,4,1,1]
3	3	{2}	[19,30,0,10]	[3,4,1,1]

NOTA: Se ha modificado la solución original, ya que pienso que el vector de nodos precedentes de la inicialización es [1, –,1,1], al no poder llegar al nodo 3 a directamente desde el origen (nodo 1).

Febrero 2003-2ª (ejercicio 2) (igual a ejercicio 3 de Septiembre 2006-reserva)

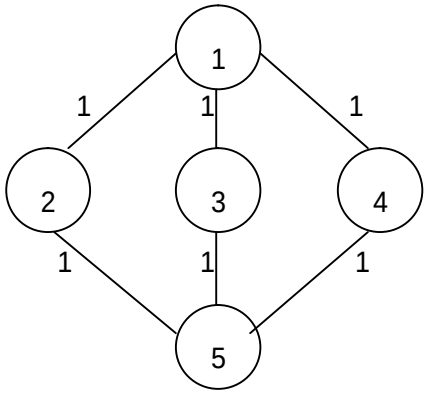
Enunciado: ¿Puede un grafo tener dos árboles de recubrimiento mínimo diferentes? En caso afirmativo, poner un ejemplo. En caso negativo, justificar la respuesta.

Respuesta:

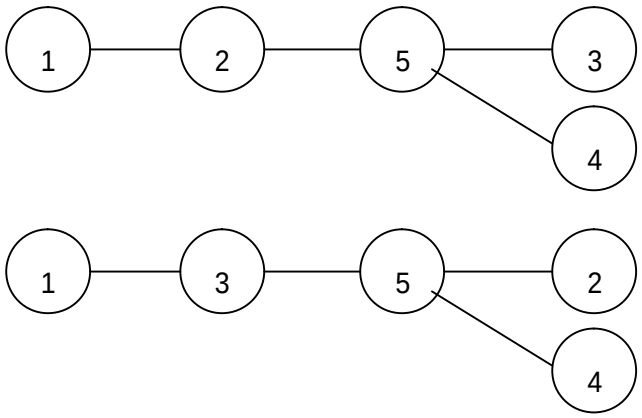
Podemos decir que un grafo **sí** que puede tener dos árboles de recubrimiento mínimo diferentes.

Siendo $G = \langle N, A \rangle$ un grafo conexo en donde N es el conjunto de nodos y A es el de aristas. Suponiendo que cada arista posee una longitud no negativa, encontrar un **árbol de recubrimiento mínimo** consiste en hallar un subconjunto T de las aristas de G tal que utilizando solamente las aristas de T, todos los nodos deben quedar conectados y además la suma de las longitudes de las aristas de T debe ser tan pequeña como sea posible.

Un **ejemplo** de eso podría ser este grafo:



Tendremos dos árboles de recubrimiento mínimo:



Febrero 2003-2ª (ejercicio 3)

Enunciado: Aplicar el algoritmo de planificación con plazo fijo para las actividades a_i maximizando el beneficio g_i en el plazo d_i. Detallar todos los pasos con claridad.

i	1	2	3	4	5	6	7	8
g_i	20	10	7	15	25	15	5	30
d_i	4	5	1	1	3	3	1	2

Respuesta:

Empezaremos por ordenar la matriz de costes y plazos por **orden decreciente** de ganancias, por ser lo que queremos maximizar, incluyendo la nueva fila como vimos antes. Quedaría así:

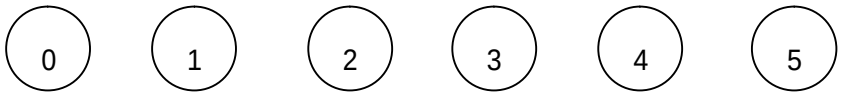
i	1	2	3	4	5	6	7	8
a_i	8	5	1	4	6	2	3	7
g_i	30	25	20	15	15	10	7	5
d_i	2	2	4	1	3	5	1	1

Para resolverlo hemos cogido el algoritmo *rápido*, en el que tomaremos una estructura de partición e iríamos fusionándolas con las tareas. La solución dada con punteros es mía personal, salvo que el orden de las tareas se ha respetado con respecto al de la solución (oficial

u “oficiosa”). En la dada del ejercicio lo resuelven como conjuntos disjuntos, añadiendo la tarea cuando entra en la planificación. Veremos paso a paso este algoritmo:

Inicialmente: $p = min(n, max(d_i)) = min(8,5) = 5$.

Tendremos como máximo una planificación de 5 tareas, siendo conjuntos distintos:

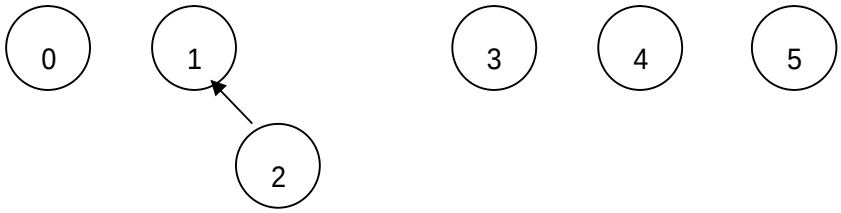


Como añadido pondremos la matriz j, donde reflejaríamos la **planificación parcial** paso a paso de las tareas con respecto a los plazos, que sería:

1	2	3	4	5	
0	0	0	0	0	j[]

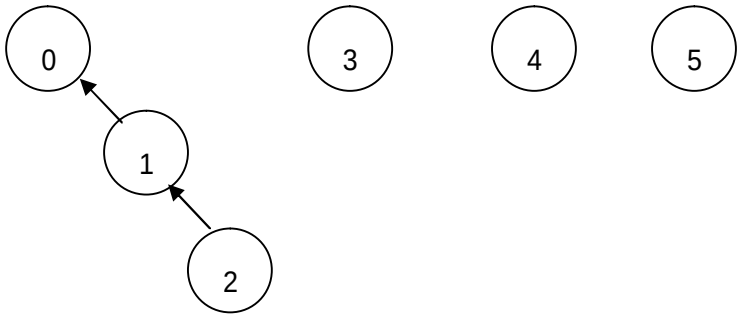
En la solución de este ejercicio nos añaden también los rótulos, pero recordemos que se considera el *rótulo* el menor de los elementos. Por eso y para agilizar el ejercicio evitamos ponerlo.

Primer intento: $d_1 = 2$. Se asigna la tarea 1 a la posición 2.
$F(K) = 2, F(L) = F(K) - 1 = 1$. Fusionamos K con L



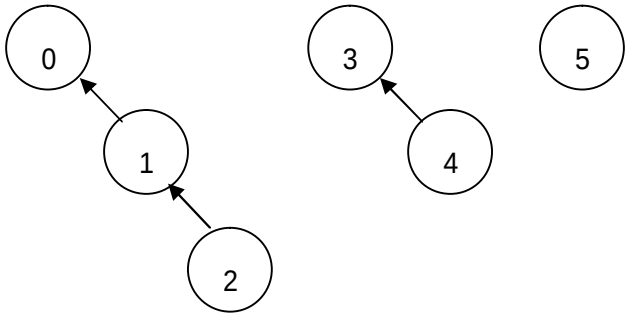
1	2	3	4	5	
0	1	0	0	0	j[]

Segundo intento: $d_2 = 2$. Se asigna la tarea 2 a la posición 2.
$F(K) = 2, F(L) = F(K) - 1 = 1$. Fusionamos K con L



1	2	3	4	5	
2	1	0	0	0	j[]

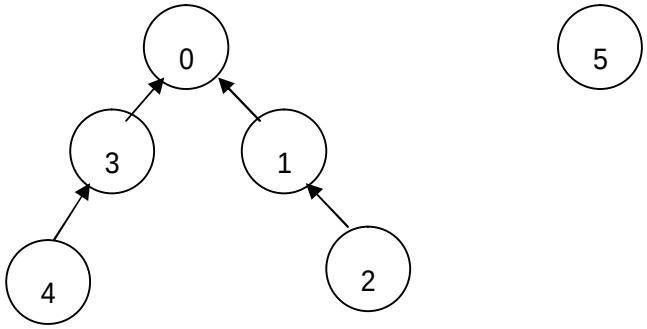
Tercer intento: $d_3 = 4$. Se asigna la tarea 3 a la posición 4.
$F(K) = 4$, $F(L) = F(K) - 1 = 3$. Fusionamos K con L



1	2	3	4	5	
2	1	0	3	0	j[]

Cuarto intento: $d_4 = 1$. Se rechaza la tarea, porque no hay posiciones libres disponibles.

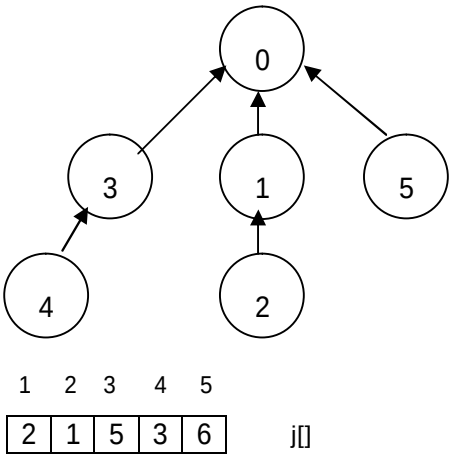
Quinto intento: $d_5 = 3$. Se asigna la tarea 4 a la posición 3.
$F(K) = 3$, $F(L) = F(K) - 1 = 2$. Fusionamos K con L



1	2	3	4	5	
2	1	5	3	0	j[]

NOTA DEL AUTOR: Esta solución no está en la dada por la respuesta, pero lo añadiremos, en este caso nos basaremos en el algoritmo página 241 del libro. Es decir, al fusionarse dos conjuntos se fusionarían los rótulos, que en este caso el del $0 - 1 - 2$ es 0 y el del $3 - 4$ es 3, por lo que apuntaría al nodo 0. Es insignificante esta apreciación, ya que el orden de ejecución de tareas queda igual pero entiendo que es lo suyo el hacerlo bien. No sé si estaré equivocada, es una apreciación mía personal.

Sexto intento: $d_6 = 5$. Se asigna la tarea 4 a la posición 3.
$F(K) = 5$, $F(L) = F(K) - 1 = 4$. Fusionamos K con L



NOTA: Pasaría algo similar al intento anterior, por lo que seguiríamos nuestra filosofía. En la solución (insisto, no sé si es oficial) que se ve en otros exámenes el puntero del nodo 5 apuntaría a 4. No estoy realmente segura, pero creo que es lo suyo.

Séptimo y octavo intento: Se rechazan las tareas, por no haber posiciones libres.

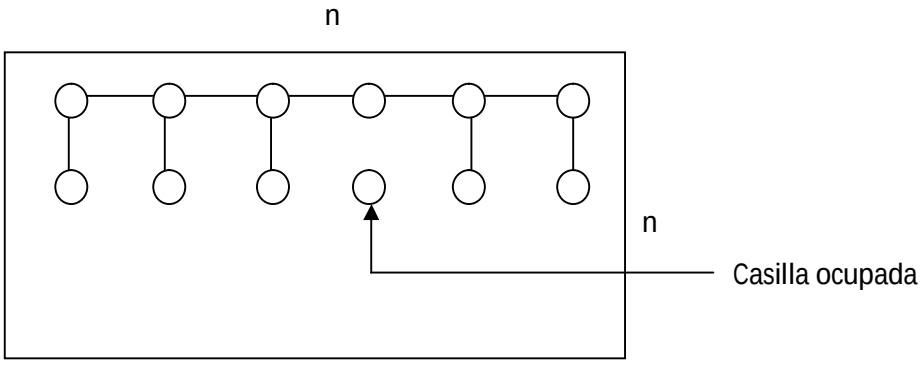
Queda decir que las tareas que salen en la solución son las ordenadas, con lo que habría que hacer el cambio con respecto a a_i, que serían las tareas reales. Según dicha solución el orden de ejecución de las tareas sería: a_5, a_8, a_6, a_1, a_2, que realmente serían el orden de las tareas tras hacer el cambio tras ordenar las tareas (para eso hay que ver la tabla de arriba).

Septiembre 2003 (ejercicio 3)

Enunciado: Explicar de qué manera se puede implementar mediante un esquema voraz el conocido problema de la búsqueda del camino más corto hacia la salida a un laberinto descrito por la matriz rectangular de casillas de tipos *libre* y *ocupada*, y otras dos de tipo *entrada* y *salida*. Compararlo en términos de costes con otras soluciones.

Respuesta: En nuestra implementación usando algoritmos voraces, cada casilla se corresponde con un nodo. Casillas libres adyacentes tendrían aristas dirigidas en ambas direcciones. El peso sería unitario para cada arista. Las casillas de tipo *ocupada* no tienen aristas origen ni destino.

Quedaría gráficamente algo así, teniendo en cuenta que las aristas en ambas direcciones equivalen a aristas no dirigidas y que no representamos el coste de cada una por ser unitario (añadido del autor):



El resto de la respuesta es la dada en el ejercicio. De esta manera, un algoritmo de Dijkstra puede hallar el camino más corto de un nodo *llegada* a todos los demás, incluyendo el nodo s*alida*.

En términos de coste, sin embargo, es necesario tener en cuenta que si el laberinto es un cuadrado de lado n, como hemos dibujado previamente, el grafo tendrá $v = n^2$ nodos y alrededor de $a = 4 * n^2$ aristas (no puedo explicar con detalle de donde salen estos valores, haré acto de fe). En el análisis del coste, la resolución de Dijkstra si v (número de nodos del grafo) es lo suficientemente grande hace cierta la expresión $a \ll v^2$ y, por tanto, podemos aproximarnos al coste $O\big((a + v) * \log(v)\big)$.

Septiembre 2003-reserva (ejercicio 2)

Enunciado: Con el criterio de tomar primero la moneda de mayor valor que sea menor o igual que el cambio que queda por devolver, ¿existe un algoritmo voraz para el problema de devolver cambio con el menor número de monedas en cualquier sistema monetario? Justifique su respuesta.

Respuesta: Solución propuesta por el autor. **No**, no existe un algoritmo voraz para el problema de devolver cambio con el menor número de monedas, debido al sistema monetario que escogeremos, tal y como nos dicen en el enunciado. Para comprobar esto tendríamos el siguiente contraejemplo (está en el libro de ejercicios):

Se nos dan monedas de 30, 24, 12, 6, 3 y 1 unidades monetarias.

Tendremos que escoger el mínimo número de monedas que sumen 48, para ello aplicando este algoritmo voraz cogeríamos 1 de 30, 1 de 12 y 1 de 6 unidades monetarias. Si hubiéramos cogido 2 de 24 habríamos conseguido la solución óptima, por lo que se descarta que se pueda resolver estos tipos de problemas en cualquier sistema monetario empleando algoritmos voraces, en cualquier caso y siendo de optimización serían de ramificación y poda.

Febrero 2004-1ª (ejercicio 3)

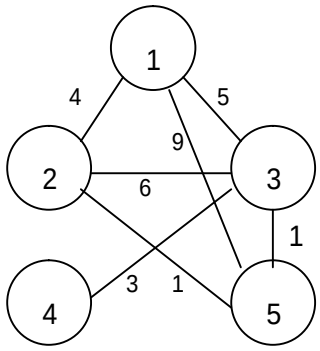
Enunciado: Resolver mediante el algoritmo de Kruskal el árbol de expansión mínimo del grafo definido por la siguiente matriz de adyacencia:

	1	2	3	4	5
1	-	4	5	∞	9
2	-	-	6	∞	1
3	-	-	-	3	1
4	-	-	-	-	∞
5	-	-	-	-	-

Respuesta: Esta solución es dada por el autor. Vemos las siguientes características de la matriz de adyacencia no dadas en el enunciado:

- Al ser un algoritmo de Kruskal suponemos que el grafo tiene aristas no dirigidas, lo que significa que la matriz es simétrica.
- Por eso, existen símbolos "–", cuyo significado es que ya nos lo han dado antes en la mitad superior de la matriz, es decir, por ser matriz simétrica.
- Por último, el símbolo ∞ indica que no existe conexión entre aristas.

Como en ejercicios anteriores tomaremos las filas como origen y las columnas como destino, aunque siendo grafo no dirigido nos dará igual, por el momento (en los dirigidos de antes no era así). El grafo sería:



Tendremos estos **pasos**:

1º. Ordenamos las aristas de menor a mayor valor:

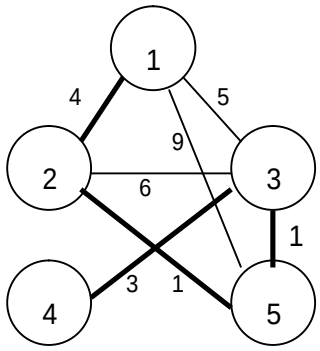
Nodo	Coste
{3,5}	1
{2,5}	1
{3,4}	3
{1,2}	4
{1,3}	5
{2,3}	6
{1,5}	9

2º. Una vez ordenadas las aristas, pasamos a resolverlo por el algoritmo de Kruskal, que como no nos piden más, ponemos estos campos:

Paso	Arista seleccionada	Componentes conexas
Inicialización	-	{1}, {2}, {3}, {4}, {5}
1	{3,5}	{1}, {2}, {3,5}, {4}
2	{2,5}	{1}, {2,3,5}, {4}
3	{3,4}	{1}, {2,3,4,5}
4	{1,2}	{1,2,3,4,5}

Proceso terminado, ya no queda ninguna componente no conexa.

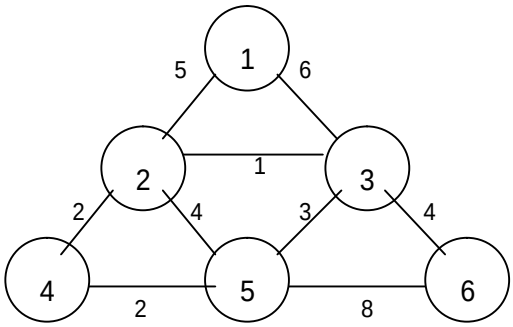
Para que se vea de modo mejor resaltaremos las aristas que forman el **árbol de recubrimiento mínimo** (será el primer y último ejercicio en el que lo hagamos):



Observamos que quedan unidas todas las aristas y que forman un conjunto conexo. Esto tratábamos de demostrar.

Febrero 2004-2ª (ejercicio 2)

Enunciado: Aplica el algoritmo de Kruskal al siguiente grafo indicando claramente en cada paso que arista se selecciona, la evolución de las componentes conexas y la evolución de la solución.



Respuesta: Como en el ejercicio anterior, el primer paso es ordenar las aristas de menor a mayor valor:

$\{\{2,3\},\{2,4\},\{4,5\},\{3,5\},\{3,6\},\{2,5\},\{1,2\},\{1,3\},\{5,6\}\}$

NOTA: Por ser coherente con los ejercicios anteriores nuestro convenio para las aristas entre nodos es usar llaves, pero en la solución oficial (u oficiosa, lo desconozco) usan paréntesis. No va a significar un cambio en el algoritmo, sólo a nivel de significado. Seguimos, por tanto, la notación del libro, dada en los apuntes.

Paso	Arista seleccionada	Componentes conexas	Solución
Inicialización	-	$\{1\},\{2\},\{3\},\{4\},\{5\},\{6\}$	$\emptyset$
1	$\{2,3\}$	$\{1\},\{2,3\},\{4\},\{5\},\{6\}$	$\{\{2,3\}\}$
2	$\{2,4\}$	$\{1\},\{2,3,4\},\{5\},\{6\}$	$\{\{2,3\},\{2,4\}\}$
3	$\{4,5\}$	$\{1\},\{2,3,4,5\},\{6\}$	$\{\{2,3\},\{2,4\},\{4,5\}\}$
4	$\{3,5\}$	Rechazada, por estar en el mismo conjunto	
5	$\{3,6\}$	$\{1\},\{2,3,4,5,6\}$	$\{\{2,3\},\{2,4\},\{4,5\},\{3,6\}\}$
6	$\{2,5\}$	Rechazada, por estar en el mismo conjunto	
7	$\{1,2\}$	$\{1,2,3,4,5,6\}$	$\{\{2,3\},\{2,4\},\{4,5\},\{3,6\},\{1,2\}\}$

Proceso terminado. No queda ninguna componente no conexa.

NOTA DEL AUTOR: Vemos que en la solución que se nos da al ordenar el vector se pone antes la arista {4,5} y {2,4}. Observamos, que al intercambiarlos la solución no cambia en absoluto.

Septiembre 2004 (ejercicio 3)

Enunciado: ¿En qué se diferencian los algoritmos de Prim y de Kruskal? Discute tanto los algoritmos como su complejidad en los casos peores de cada caso.

Respuesta: Los algoritmos de Prim y Kruskal se asemejan en que ambos crean árboles de recubrimiento mínimo, aunque difieren en la función de selección de las aristas. Mientras que el **algoritmo de Prim** selecciona un nodo y construye un árbol a partir de él, seleccionando en cada etapa la arista más corta posible que pueda extender el árbol hasta un nodo adicional, el **de Kruskal** escoge las aristas de menor a mayor coste sin importar si están conexas o no.

El coste del algoritmo de Prim es $\theta(n^2)$, mientras que el de Kruskal es $\theta(a * \log(n))$, siendo a el número de aristas. Distinguiremos estos casos para este último:

- Si el grafo es disperso (a tiende a n), el coste del algoritmo es $\theta(n * \log(n))$.
- Si el grafo es denso (a tiende a n^2), el coste es $\theta(n^2 * \log(n))$.

Es mejor, por tanto, el coste cuando el grafo es disperso en el algoritmo de Kruskal que en el de Prim. En los grafos densos pasaría al revés.

NOTA DEL AUTOR: Estos ejercicios se hacen igual. Se amplia, en este caso, la solución dada con respecto a la del ejercicio.

Septiembre 2004-reserva (ejercicio 1)

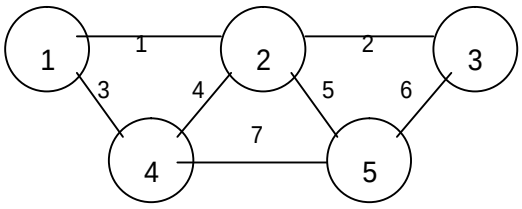
Enunciado: Defina que es un árbol de recubrimiento mínimo, explica alguno de los algoritmos propuestos en la asignatura para hallarlos, explicando paso a paso un ejemplo.

Respuesta: La solución está hecha completamente por un alumno.

Un **árbol de recubrimiento mínimo** es un subgrafo que contiene a todos los vértices, que es conexo y que el coste de total de sus aristas sea mínimo, ocupando n – 1 aristas.

Entre los algoritmos que hay para hallarlos encontramos dos, el de Kruskal y el de Prim. Ambos dos llegan a crear un árbol de recubrimiento mínimo, pero difieren en la función de selección de las aristas y el modo de cogerlos.

Veremos, por tanto, el algoritmo de Kruskal con un ejemplo:



Ordenamos las aristas de menor a mayor coste:

$\{\{1,2\},\{2,3\},\{1,4\},\{2,4\},\{2,5\},\{3,5\},\{4,5\}\}$

Los pasos nos quedarían así:

Paso	Arista seleccionada	Componentes conexas	Solución
Inicialización	-	$\{1\},\{2\},\{3\},\{4\},\{5\}$	$\emptyset$
1	$\{1,2\}$	$\{1,2\},\{3\},\{4\},\{5\}$	$\{\{1,2\}\}$
2	$\{2,3\}$	$\{1,2,3\},\{4\},\{5\}$	$\{\{1,2\},\{2,3\}\}$
3	$\{1,4\}$	$\{1,2,3,4\},\{5\}$	$\{\{1,2\},\{2,3\},\{1,4\}\}$
4	$\{2,4\}$	Rechazada, por estar en el mismo conjunto	
5	$\{2,5\}$	$\{1,2,3,4,5\}$	$\{\{1,2\},\{2,3\},\{1,4\},\{2,5\}\}$

Finaliza el problema, ya que existe una única componente conexa.

Septiembre 2004-reserva (ejercicio 3)

Enunciado: Se dispone de n productos diferentes, infraccionables y en cantidades ilimitadas. Cada tipo de producto tiene asociado un peso y un beneficio concreto. Deseamos cargar un camión, que puede transportar un peso máximo PMAX, maximizando el beneficio de los productos transportados por éste. ¿Sería posible un algoritmo voraz a este planteamiento? Responde razonadamente y plantear un ejemplo que justifique la respuesta.

Respuesta:

No, no es posible ningún algoritmo cuya función de selección llegue a encontrar solución óptima, por ser infraccionables los objetos. Pondremos un **contraejemplo**:

Se nos da estos productos con $PMAX = 100$

i	1	2
v_i	150	100
w_i	51	50
$\frac{v_i}{w_i}$	2,9	2

Tomaremos los objetos por orden decreciente de relación $\frac{v_i}{w_i}$ y vemos que al coger el primer objeto según está función de selección, tenemos que cogeremos 1 objeto $i = 1$, resultando:

$w = 51$.
$v = 150$.

Al no poder entrar ya más objetos en la mochila, vemos que ya sería la solución mejor. Aun así, encontramos otra solución más óptima, que sería:

$w = 100$.
$v = 200$.

En este caso, seleccionamos dos objetos $i = 2$ y vemos que el valor es mayor que en el caso antes. Por tanto, no encontramos solución óptima usando algoritmos voraces.

NOTA: Vemos que en el enunciado nos hablan de beneficios, pero al ser un problema de la mochila, lo resolveremos como valor, que para este caso es similar. De nuevo, está resuelto por el autor completamente el ejercicio.

Febrero 2005-1ª (ejercicio 1)

Enunciado: Se dispone de ficheros $f_1, f_2, \dots, f_n$ con tamaños $l_1, l_2, \dots, l_n$ y de un disquete con una capacidad total de almacenamiento $d < l_1 + l_2 + \cdots + l_n$

a) Suponiendo que se desea maximizar el número de ficheros almacenados y que se hace uso de un planteamiento voraz basado en la selección de ficheros de menor a mayor tamaño, ¿el algoritmo propuesto siempre obtendría la solución óptima? En caso afirmativo, demostrar la optimalidad y en caso negativo, ponga un contraejemplo.
b) En caso de que quisiéramos ocupar la mayor cantidad de espacio en el disquete independientemente del número de ficheros almacenados, ¿una estrategia voraz basada en la selección de mayor a menor tamaño obtendría en todos los casos la solución óptima? En caso afirmativo, demostrar la optimalidad y en caso negativo, ponga un contraejemplo.

Respuesta:

a) El algoritmo voraz ***si*** obtendría la solución óptima, es decir, un disquete con el mayor número posible de ficheros.

   Para la **demostración**, tendríamos lo siguiente:
   Supongamos que los programas se encuentran inicialmente ordenados de menor a mayor tamaño, vamos a hacer la **demostración de optimalidad** de la solución comparando una solución óptima con una obtenida por el algoritmo voraz. Si ambas soluciones no fueran iguales, iremos transformando la solución óptima de partida, de forma que continúe siendo óptima, pero asemejándola cada vez con la obtenida por el algoritmo voraz. Si consiguiéramos igualar ambas soluciones en un número finito de pasos podremos afirmar que la solución obtenida es óptima (de modo nemotécnico es el método dado en teoría de *reducción de diferencia*s).

   > Notación: Una solución cualquiera viene representada por $Z = (z_1, z_2, \dots, z_n)$ donde $z_1 = 0$ implica que el fichero f_i no ha sido seleccionado como parte de la solución. De este modo, $\sum_{i=1}^{n} z_i$ indicará el número de ficheros seleccionados como solución al problema.

   Siendo X la solución devuelta por la estrategia voraz e Y la solución al problema. Supongamos que la estrategia voraz selecciona los k primeros ficheros (con $1 \leq k \leq n$), recordemos que los ficheros se encuentran inicialmente ordenados de menor a mayor tamaño. El fichero $k + 1$ es rechazado, puesto que ya no es posible incluir un solo fichero más. De este modo, la solución X será $(x_1, x_2, \dots, x_k, \dots, x_n)$ donde $\forall\, i \in \{1..k\}.\, x_i = 1$ y $\forall\, i \in \{k + 1..n\}.\, x_i = 0$.

   Comenzando a comparar X con Y de izquierda a derecha, supongamos que $j \geq 1$ sea la primera posición donde $x_j \neq y_j$. En este caso, obligatoriamente $j \leq k$, ya que en caso contrario la solución óptima incluiría todos los ficheros escogidos por la estrategia voraz y alguno más, lo que se ***contradice*** con el hecho de que los ficheros del $k + 1$ al n se rechazan por la estrategia voraz porque no caben.

   Este modo, $x_j \neq y_j$, implica que $y_i = 0$, por lo que $\sum_{i=1}^{j} y_i = j - 1$, que es menor que el número de ficheros seleccionados por nuestra estrategia voraz como solución al problema $\sum_{i=1}^{j} x_i = j$. Como suponemos que Y es una solución óptima $\sum_{i=1}^{n} y_i \geq \sum_{i=1}^{n} x_i = k$, esto significa que existe un $l > k \geq j$ (siendo *l* el tamaño) tal que $y_l = 1$, es decir, existe un fichero posterior para compensar el que no se ha escogido antes. Por la orden impuesta a los ficheros, sabemos que $l_j \leq l_i$, es decir, que si f_l cabe en el disco, podemos poner f_j sin sobrepasar la capacidad total. Realizando este cambio en la solución óptima Y, obtenemos otra solución Y' en la cual $y'_j = 1 = x_i$, $y'_j = 0$ para el resto $y'_j = y_i$. Esta **nueva solución es más parecida a X**, y tiene el mismo número de ficheros que Y', por lo que sigue

siendo *óptima*. Repitiendo este proceso, podemos ir igualando los ficheros en la solución óptima a los de la solución voraz X, hasta alcanzar la posición k.

b) El algoritmo voraz no obtendría la solución óptima en todos los casos.
Contraejemplo: Supongamos la siguiente lista de ficheros con tamaños asociados:

Fichero	F_1	F_2	F_3
Tamaño	40	30	15

Supongamos que la capacidad del disquete es de 45 (este contraejemplo es parecido al hecho anteriormente en el ejercicio 1 de Septiembre 2004-reserva).

Aplicando la estrategia voraz propuesta, es decir, escoger el fichero con mayor tamaño únicamente podríamos almacenar el fichero F_1, ocupando 40 de los 45 de capacidad que tiene el disquete. Sin embargo, si hubiéramos seleccionado los ficheros F_2 y F_3 hubiera sido posible maximizar el espacio ocupado en el disco.

Febrero 2005-1ª (ejercicio 2)

Enunciado: Exponga y explique el algoritmo más eficiente que conozca para realizar una planificación de tareas de plazo fijo maximizando el beneficio.
Dada la tabla adjunta de tareas con sus beneficios (g_i) y caducidades (d_i), aplique paso a paso el algoritmo propuesto, suponiendo que se desea realizar una planificación con tiempo $t = 5$

i	1	2	3	4	5	6	7	8	9
g_i	30	10	2	11	10	9	2	56	33
d_i	5	3	2	2	1	2	7	5	4

Respuesta: El ejercicio está hecho de manera más didáctica, aunque respetando la solución. Dividiremos este ejercicio en dos apartados:

a) El algoritmo más apropiado es el algoritmo secuencia2, que sería el algoritmo denominado *rápido* y el que emplearemos en este ejercicio:

```
funcion secuencia2 (d[1..n]): k, matriz [1..k]
    matriz j, F[0..n]
    { Iniciación }
    p = min(n, max{d[i]|1 ≤ i ≤ n});
    para i ← 0 hasta p hacer j[i] ← 0
                             F[i] ← i
                             Iniciar el conjunto {i}
    { Bucle voraz }
    para i ← 1 hasta n hacer      { Orden decreciente de g }
        k ← buscar(min(p, d[i]))
        m ← F[k]
        si m ≠ 0 entonces
            j[m] ← i;
            l ← buscar(m − 1)
            F[k] ← F[l]
            { El conjunto resultante tiene la etiqueta k o l }
            fusionar (k, l)
    { Sólo queda comprimir la solución }
    k ← 0
    para i ← 1 hasta p hacer
        si j[i] > 0 entonces k ← k + 1
                             j[k] ← j[i]
    devolver k, j[1..k]
```

El **algoritmo** será el siguiente:

i. Iniciación: Toda posición 0,1,2, … , p está en un conjunto diferente y $F([i]) = i$, $0 \leq i \leq p$.

$p = min(n, max(d_i))$.
- $max(d_i)$ → Mayor de los plazos
- n → Número de tareas

La posición 0 sirve para ver cuando la planificación está llena.

ii. Adición de una tarea con plazo d; se busca un conjunto que contenga a d; sea K este conjunto. Si $F(K) = 0$ se rechaza la tarea, en caso contrario:

- Se asigna la nueva tarea a la posición $F(K)$.
- Se busca el conjunto que contenga $F(K) - 1$. Llamemos L a este conjunto (no puede ser igual a K).
- Se fusionan K y L. El valor de F para este nuevo conjunto es el valor viejo de $F(L)$.

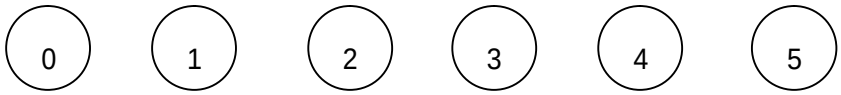
b) Como en ejercicios anteriores seguiremos estos pasos:

Como primer paso, ordenamos la tabla propuesta por **orden decreciente** de beneficios como sigue:

i	1	2	3	4	5	6	7	8	9
a_i	8	9	1	4	5	2	6	3	7
g_i	56	33	30	11	10	10	9	2	2
d_i	5	4	4	2	1	3	2	2	7

Veremos paso a paso el algoritmo como sigue:

Inicialmente: $p = min(n, max(d_i)) = min(9,5) = 7$. En este caso concreto queremos planificar las 5 primeras tareas, según nos dicen en el enunciado, por lo que p se reduce a 5 ($p = 5$).



Ponemos de nuevo la matriz de resultados, donde reflejaríamos la planificación parcial de las tareas con respecto a los plazos (las tareas reales), que sería:

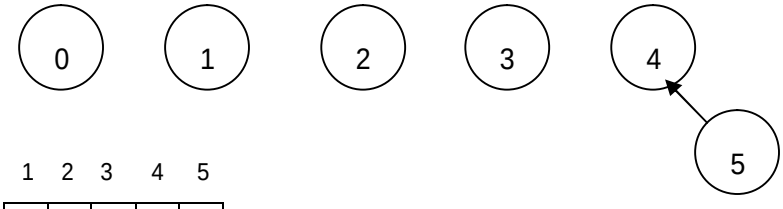
1	2	3	4	5
0	0	0	0	0

res[]

En la solución de este ejercicio nos añaden también los rótulos, pero recordemos que se considera el **rótulo** el menor de los elementos. Por eso y para agilizar el ejercicio evitamos ponerlo.

Vamos escogiendo cada una de las tareas por el orden antes puesto e incluyéndolos en su correspondiente de partición. Esta operación implica fusionar las estructuras en la cual se ha incluido la tarea con la estructura de partición inmediatamente inferior. Escogeremos, en este caso (como en anteriores), las tareas ordenadas siguiendo i de la tabla, para luego tomar las tareas empleando a_i.

Primer intento: $d_1 = 5$. Se asigna la tarea 1 a la posición 5.
$F(K) = 5$, $F(L) = F(K) - 1 = 4$. Fusionamos K con L

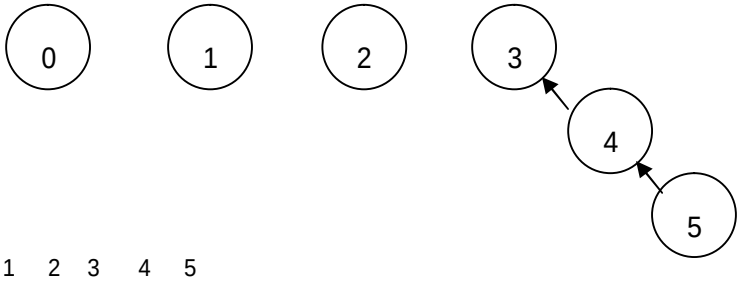


1	2	3	4	5
0	0	0	0	1

res[]

Segundo intento: $d_2 = 4$. Se asigna la tarea 2 a la posición 4.

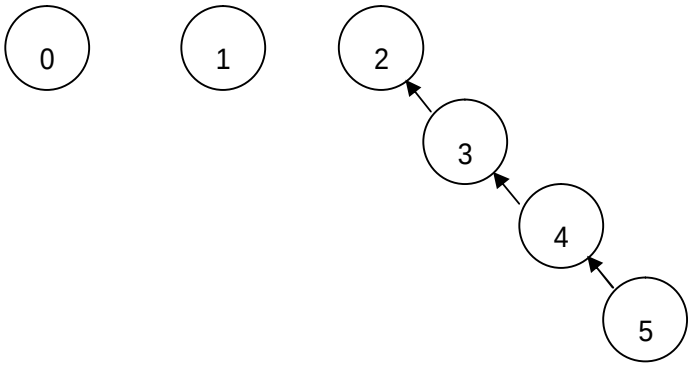
$F(K) = 4, F(L) = F(K) - 1 = 3$. Fusionamos K con L



1	2	3	4	5
0	0	0	2	1

res[]

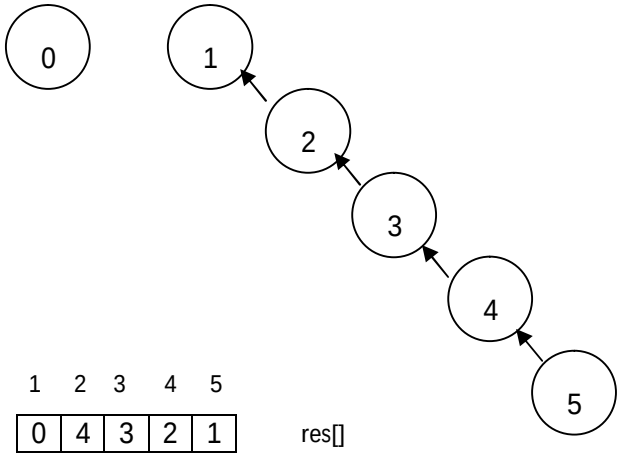
Tercer intento: $d_3 = 4$. Se asigna la tarea 3 a la posición 4.
$F(K) = 4, F(L) = F(K) - 1 = 3$. Fusionamos K con L



1	2	3	4	5
0	0	3	2	1

res[]

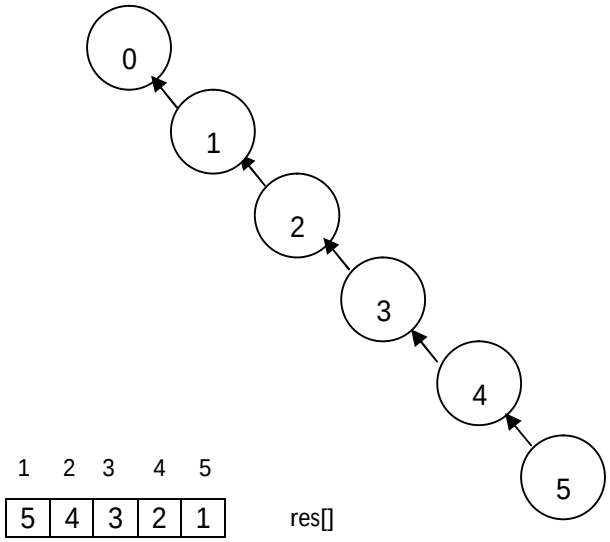
Cuarto intento: $d_4 = 2$. Se asigna la tarea 4 a la posición 2.
$F(K) = 2, F(L) = F(K) - 1 = 1$. Fusionamos K con L



1	2	3	4	5
0	4	3	2	1

res[]

Quinto intento: $d_5 = 1$. Se asigna la tarea 5 a la posición 1.
$F(K) = 1$, $F(L) = F(K) - 1 = 0$. Fusionamos K con L



El algoritmo termina cuando ya no queda ninguna estructura de partición libre para asignar tareas.
Por tanto, haciendo el cambio de i a a_i quedaría el resultado total así:

1	2	3	4	5	
5	4	1	9	8	res[]

NOTA DEL AUTOR: Este ejercicio está reescrito tomando como base la solución aportada del mismo.

Febrero 2005-2ª (ejercicio 3)

Enunciado: Demuestra por inducción que el algoritmo de Dijkstra halla los caminos mínimos desde un único origen hasta todos los demás nodos del grafo

Respuesta:

Esta solución es totalmente dada en el libro, y hecho en el resumen del tema. Demostraremos por inducción matemática que:

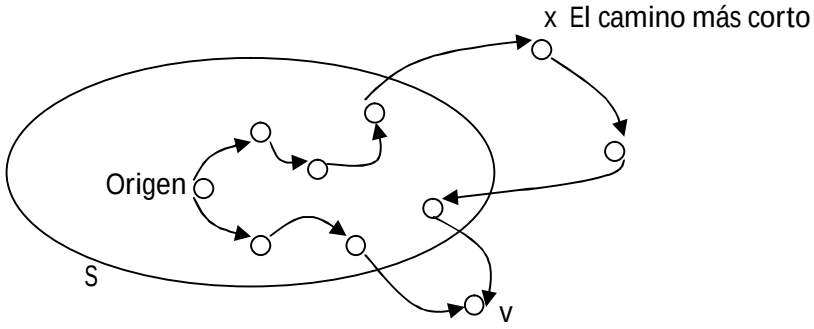
a) Si un nodo $i \neq 1$ está en S, entonces $D[i]$ da la longitud del camino más corto desde el origen hasta i, y
b) Si un nodo y no está en S, entonces $D[i]$ da la longitud del camino *especial* más corto desde el origen hasta i.

- Base: Inicialmente, solo el nodo 1, que es el origen, se encuentra en S, así que la situación a) es cierta sin más demostración. Para los demás nodos, el único camino especial desde el origen es el camino directo y D recibe valores iniciales en consecuencia. Por tanto, la situación b) es también cierta cuando comienza el algoritmo.
- Hipótesis de inducción: La hipótesis de inducción es que tanto la situación a) como la b) son válidas inmediatamente antes de añadir un nodo v a S (conjunto de nodos seleccionados). Detallamos los pasos de inducción por separado para ambas situaciones.

- Paso de inducción para la situación a): Para todo nodo que ya esté en S antes de añadir v no cambia nada, así que la situación a) sigue siendo válida. En cuanto al nodo v, ahora pertenecerá a S. Antes de añadirlo a S, es preciso comprobar que D[v] proporcione la longitud del camino más corto que va desde el origen hasta v. Por hipótesis de inducción, nos da ciertamente la longitud del camino más corto. Por tanto, hay que verificar que el camino más corto desde el origen hasta v no pase por ninguno de los nodos que no pertenecen a S.

  Supongamos lo contrario; supongamos que cuando se sigue el camino más corto desde el origen hasta v, se encuentran uno o más nodos (sin contar el propio v) que no pertenecen a S. Sea x el primer nodo encontrado con estas características. Ahora el segmento inicial de esa ruta, hasta llegar a x, es una ruta especial, así que la distancia hasta x es D[x], por la parte b) de la hipótesis de inducción. Claramente, la distancia total hasta v a través de x no es más corta que este valor, porque las longitudes de las aristas son no negativas. Finalmente, D[x] no es menor que D[v], porque el algoritmo ha seleccionado a v antes que a x. Por tanto, la distancia total hasta v a través de x es como mínimo D[v] y el camino a través de x no puede ser más corto que el camino especial que lleva hasta v.

  Gráficamente, sería:



El camino especial más corto

- Paso de inducción para la situación b): Considérese ahora un nodo w, distinto de v, que no se encuentre en S. cuando v se añade a S, ha dos posibilidades para el camino especial más corto desde el origen hasta w:
  1. O bien no cambia.
  2. O bien ahora pasa a través de v.

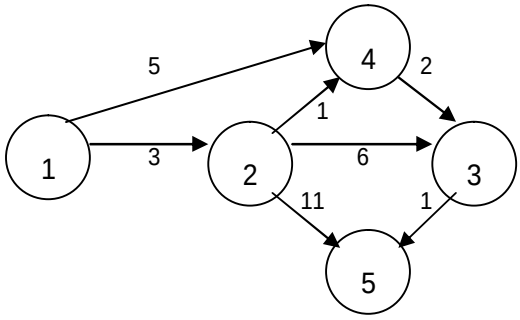
  En el **segundo caso**, sea x el último nodo de S visitado antes de llegar a w. La longitud de este camino es $D[x] + L[x, w]$. Parece a primera vista que para calcular el nuevo valor de $D[w]$ deberíamos comparar el valor anterior de $D[w]$ con $D[x] + L[x, w]$ para todo nodo x de S (incluyendo a v). Sin embargo, para todos los nodos x de S salvo v, esta comparación se ha hecho cuando se añadió x a S y $D[x]$ no ha variado desde entonces. Por tanto, el nuevo valor de $D[w]$ se puede calcular sencillamente comparando el valor anterior con $D[v] + L[v, w]$.

  Puesto que el algoritmo hace esto explícitamente, asegura que la parte b) de la inducción siga siendo cierta también cuando se añade a S un nuevo nodo v.

Para completar la demostración de que el algoritmo funciona, obsérvese que cuando se detenga el algoritmo, todos los nodos menos uno estarán en S. En ese momento queda claro que el camino más costo desde el origen hasta el nodo restante es un camino especial.

Febrero 2006-1ª (ejercicio 3)

Enunciado: Dado el siguiente grafo, rellenar la tabla adjunta indicando paso a paso como el algoritmo de Dijkstra encuentra todos los caminos mínimos de menor coste desde el nodo 1.



Respuesta:

Paso	Nodos seleccionados	Nodos no seleccionados	Vector de distancias (2 3 4 5)	Vector de predecesores
Inicialización	{1}	{2,3,4,5}	[3, ∞, 5, ∞]	[1, −,1, −]
1	{1,2}	{3,4,5}	[3,9,4,14]	[1,2,2,2]
2	{1,2,4}	{2,3}	[3,6,4,14]	[1,4,2,2]
3	{1,2,3,4}	{2}	[3,6,4,7]	[1,4,2,3]

En ejercicios anteriores no hemos puesto los nodos seleccionados como en este ejercicio, pero es otra interpretación perfectamente válida (de hecho no estoy segura que sea la oficial, aunque en la interpretación anterior seguimos la del libro de Brassard). Al igual que en ejercicios anteriores el paso de inicialización el vector de predecesores cambia con respecto a la solución oficial (insistimos u "oficiosa"), en la que no hay conexión directa, entre los nodos 1 al 3 ni el 1 al 5, por eso en el vector de predecesores está indicada esta unión con −.

Febrero 2006-2ª (ejercicio 2)

Enunciado: Sea el famoso problema de la mochila. Se dispone de n objetos y una mochila. Para i = 1,2, … . n el objeto i tiene un peso positivo w_i y un valor positivo v_i. La mochila puede llevar un peso que no sobrepase W. El objetivo es llenar la mochila de tal manera que se maximice el valor de los objetos almacenados, respetando la limitación de peso impuesta. Indique que esquema considera más adecuada para resolver este problema en los siguientes casos:

a) Los objetos se pueden fraccionar, luego se puede decidir llevar una fracción x_i del objeto i, tal que $0 \leq x_i \leq 1$ para $1 \leq i \leq n$.
b) Los objetos no se pueden fraccionar, por lo que un objeto puede o no ser añadido, pero en este último casi, solo se añade 1.

Además de nombrar el esquema o esquemas, explica el porqué de su elección, los aspectos destacados de cómo resolverías el problema y el coste asociado. No se piden los algoritmos.

Respuesta:

En el caso a) se puede utilizar el **esquema voraz**, ya que existe una función de selección que garantice obtener una solución óptima. La función de selección consiste en considerar los

objetos por orden decreciente de $\frac{v_i}{w_i}$. El coste está en $O(n * \log(n))$, incluyendo la ordenación de los objetos.

En el caso b) no se puede utilizar el esquema voraz, ya que no existe una función de selección que garantice obtener una solución óptima. Al ser un problema de optimización se puede utilizar el **esquema de ramificación y poda**. Se podrían seleccionar los elementos en orden decreciente de $\frac{v_i}{w_i}$. Así, dado un determinado nodo, una cota superior del valor que se puede alcanzar siguiendo por esa rama se puede calcular suponiendo que la mochila la rellenamos con el siguiente elemento siguiendo el orden decreciente de $\frac{v_i}{w_i}$.

El coste en el caso peor seria de orden *exponencial*, ya que en el árbol asociado al espacio de búsqueda, cada nodo tendrá dos sucesores que representarán si el objeto se añade o no a la mochila, es decir, $O(2^n)$. Sin embargo, sería de esperar que, en la práctica, el uso de la cota superior para podar reduzca el número de nodos que se exploran.

Febrero 2006-2ª (ejercicio 3) (parecido a ejercicio 3 de Febrero 2008-1ª)

Enunciado: Un dentista pretende dar servicio a n pacientes y conoce el tiempo requerido por cada uno de ellos, siendo t_i, $i = 1,2, \dots, n$ el tiempo requerido por el paciente i. El objetivo es minimizar el tiempo total que todos los clientes están en el sistema, y como el número de pacientes es fijo, minimizar la espera total equivale a minimizar la espera media. Se pide:

1. Identificar una función de selección que garantice que un algoritmo voraz puede construir una planificación óptima.
2. Hacer demostración de la optimalidad de dicha función de selección.

Respuesta:

Para la pregunta número 1 tendremos que este problema es uno de minimización del tiempo en el sistema. Para minimizar el tiempo total de los clientes tendremos que cogerlos por orden creciente de tiempos, es decir, atenderemos al cliente de menor tiempo de espera antes que al del mayor. Esta será nuestra función de selección de este problema.

Para la pregunta número 2 veremos la siguiente **demostración**, la cual es una de las más importantes que se dan en este tema, por lo que nos fijaremos muy bien en ella:

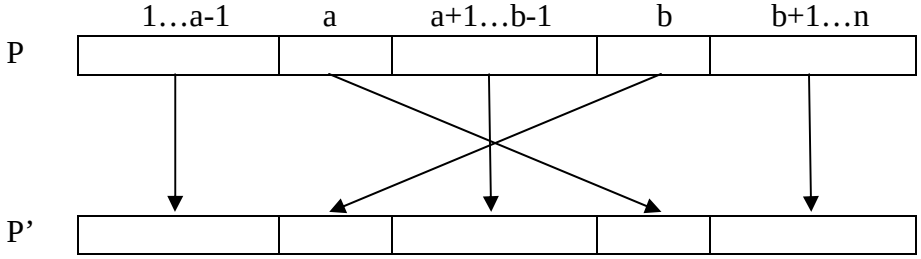
Sea $P = p_1\, p_2 \dots\, p_n$ cualquier permutación de enteros del 1 al n y sea $s_i = t_{pi}$. Si se sirven clientes en el orden P, entonces el tiempo requerido por el j-ésimo cliente que haya que servir será s_j y el tiempo transcurrido en el sistema por todos los clientes es:

$$T(P) = s_1 + (s_1 + s_2) + (s_1 + s_2 + s_3) + \cdots + (s_1 + s_2 + s_3 + \cdots + s_n) =$$
$$= n * s_1 + (n-1) * s_1 + \cdots + 2 * s_{n-1} + s_n =$$
$$= \sum_{k=1}^{n} (n - k + 1) * s_k.$$

Supongamos ahora que P no organiza a los clientes por orden de tiempos crecientes de servicio. Entonces, se pueden encontrar dos enteros a y b con $a < b$ y $s_a > s_b$. Es decir, se sirven al cliente a-ésimo antes que al b-ésimo, aun cuando el primero necesite más tiempo de servicio que el segundo. Sería algo así:

	$1 \dots a-1$	a	$a+1 \dots b-1$	b	$b+1 \dots n$
P					

Si intercambiamos la posición de esos dos clientes, obtendremos un nuevo orden de servicio o permutación P', que es simplemente el orden P después de intercambiar p_a y p_b:



El tiempo total transcurrido pasado en el sistema por todos los clientes si se emplea la planificación P' es:

$$T(P') = (n - a + 1) * s_b + (n - b + 1) * s_a + \sum_{\substack{k=1 \\ k \neq a,b}}^{n} (n - k + 1) * s_k$$

La nueva planificación es preferible a la vieja, porque:

$$T(P) - T(P') = (n - a + 1) * (s_a - s_b) + (n - b + 1) * (s_b - s_a) =$$
$$= \underbrace{(b - a)}_{>0} * \underbrace{(s_a - s_b)}_{>0} > 0$$

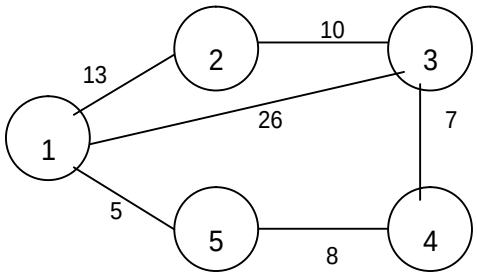
Se observa tras el intercambio que los clientes salen en su posición adecuada, ya que $s_b < s_a$ por nuestra suposición inicial, estando el resto ordenados. Por tanto, *P' es mejor que P* en conjunto.

De esta manera, se puede optimizar toda planificación en la que se sirva a un cliente antes que requiera menos servicio. Las únicas planificaciones que quedan son aquellas que se obtienen poniendo a los clientes por orden creciente de tiempo de servicio. Todas las planificaciones son equivalentes y, por tanto, todas son óptimas.

NOTA DEL AUTOR: Este ejercicio se parece mucho al número 3 de Febrero 2008-1ª, aunque en ese caso sería de unos fontaneros que quieren minimizar el tiempo de atención del cliente para que las ganancias sean máximas. Se haría exactamente igual, incluso la demostración de optimalidad.

Diciembre 2006 (ejercicio 3)

Enunciado: Dado el grado de la figura, aplica el algoritmo de Dijkstra para hallar los caminos más cortos desde el nodo 1 hasta uno de los otros nodos, indicando en cada paso: nodos seleccionados, nodos no seleccionados, vector de distancias y vector de nodos precedentes.



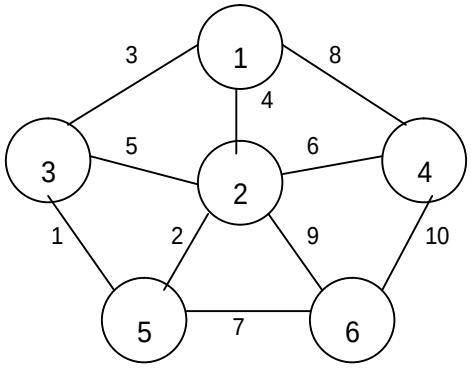
Respuesta: Estos ejercicios se hacen exactamente igual a los que hemos visto previamente, así que pondremos el cuadro con los datos que nos solicitan:

Paso	Nodos seleccionados	Nodos no seleccionados	Vector de distancias	Vector de predecesores
			2 3 4 5	
Inicialización	{1}	{2,3,4,5}	$[13,26,\infty,5]$	[1,1, −,1]
1	{1,5}	{2,3,4}	[13,26,13,5]	[1,1,5,1]
2	{1,2,5}	{3,4}	[13,23,13,5]	[1,2,5,1]
3	{1,2,4,5}	{3}	[13,20,13,5]	[1,4,5,1]

La solución proporcionada es del autor totalmente, por lo que no se asegura que esté correcta.

Febrero 2007-2ª (ejercicio 3)

Enunciado: aplicar el algoritmo de Kruskal al siguiente grafo indicando claramente en cada paso que arista se selecciona, la evolución de las componentes conexas y la evolución de la solución.



Respuesta: Ordenamos las aristas por orden creciente de coste (de menor a mayor):

$\{\{3,5\},\{2,5\},\{1,3\},\{1,2\},\{2,3\},\{2,4\},\{5,6\},\{1,4\},\{2,6\},\{4,6\}\}$

En la solución que tengo, sale que las aristas están entre paréntesis (), en vez de entre corchetes {} como se ha resuelto este ejercicio. Si hiciera falta solo hay que modificar el uso de estos signos y adaptarlo a la solución y cambiarlos.

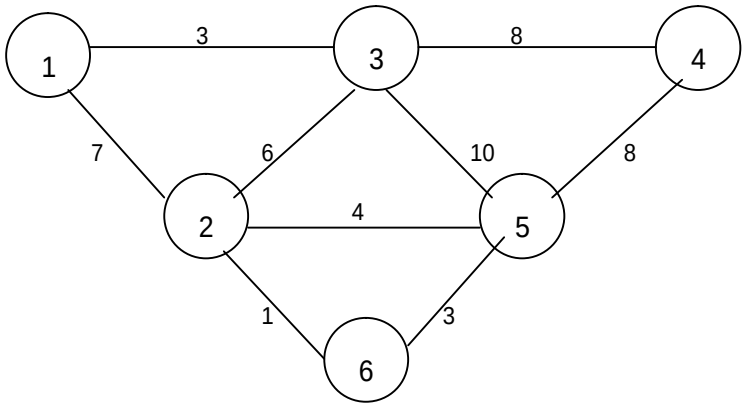
Siguiendo el algoritmo será:

Paso	Arista seleccionada	Componentes conexas	Solución
Inicialización	-	{1},{2},{3},{4},{5},{6}	∅
1	{3,5}	{1},{2},{3,5},{4},{6}	$\{\{3,5\}\}$
2	{2,5}	{1},{2,3,5},{4},{6}	$\{\{3,5\},\{2,5\}\}$
3	{1,3}	{1,2,3,5},{4},{6}	$\{\{3,5\},\{2,5\},\{1,3\}\}$
4	{1,2}	Rechazada, por estar en el mismo conjunto	
5	{2,3}	Rechazada, por estar en el mismo conjunto	
6	{2,4}	{1,2,3,4,5},{6}	$\{\{3,5\},\{2,5\},\{1,3\},\{2,4\}\}$
7	{5,6}	{1,2,3,4,5,6}	$\{\{3,5\},\{2,5\},\{1,3\},\{2,4\},\{5,6\}\}$

Finaliza el problema, ya que existe una única componente conexa.

Septiembre 2008-reserva (ejercicio 3)

Enunciado: Aplique el algoritmo de Prim al siguiente grafo empezando por el nodo 1. Indique claramente en cada paso qué arista se selecciona, y la evolución de la solución.



Respuesta: Hemos visto en muchos ejercicios anteriores como se resuelven este tipo de problemas, por lo que lo dejaríamos pendiente de resolución. Simplemente, es seguir el procedimiento de este algoritmo.

2ª parte. Problemas de exámenes solucionados:

Tendremos en cuenta los pasos a seguir para resolver problemas:

- Elección del esquema (voraz, vuelta atrás, divide y vencerás).
- Identificación del problema con el esquema.
- Estructura de datos.
- Algoritmo completo (con pseudocódigo).
- Estudio del coste.

Antes de ver los ejercicios decir que el esquema nos basaremos en los del libro de teoría, de Brassard, que como diferencia más importante es que existe una función que es *factible*, mientras que en el libro de práctica es *completable*. Ambas hacen la misma función, pero nos seguiremos basando en el libro de teoría.
Hay ejercicios que hemos modificado, sobre todo el código del algoritmo completo, ya que no cuadran la solución dada, por lo que lo especificaremos cuando llegue ese caso en el ejercicio.
Al ser problemas con voraces, tendremos que realizar una demostración de optimalidad y si existe lo pondremos en la elección el esquema, por ser más lógico ponerlo así (lo estimo yo).

Febrero 1996-1ª (problema 2) (igual a 2.4 libro de problemas resueltos)

Enunciado: Un recubrimiento R de vértices de un grafo no dirigido $G = \langle N, A \rangle$ es un conjunto de vértices tales que cada arista del grafo incide en, al menos, un vértice de R. Diseñar un algoritmo que, dado un grafo no dirigido, calcule un recubrimiento de vértices de tamaño mínimo.

Respuesta:

1. Elección razonada del esquema

   El esquema voraz se adapta perfectamente al problema, ya que:

   - Se trata de un **problema de optimización**: No sólo hay que encontrar un recubrimiento, sino que éste ha de ser de tamaño mínimo.
   - De entre un conjunto de vértices (candidatos) hay que seleccionar un subconjunto que será la solución. Solo hay que encontrar la **función de selección** adecuada (si existe) para resolver el problema mediante un algoritmo voraz.

   El esquema de divide y vencerás es descartable, pues no hay forma obvia de dividir el problema en subproblemas idénticos cuyas soluciones puedan combinarse en una solución global. El esquema de vuelta atrás es un esquema muy general y casi siempre muy costoso que no debemos usar si podemos dar un algoritmo voraz que resuelva el problema.

2. Esquema general e identificación con el problema

   El esquema voraz se aplica a **problema de optimización**. Consiste en ir seleccionando elementos de un conjunto de candidatos que se van incorporando a la solución. Se caracterizan porque nunca se deshace una decisión ya tomada: los candidatos desechados no vuelven a ser considerados y los incorporados a la solución permanecen en ella hasta al final del algoritmo.

   Es crucial determinar la función de selección adecuada que nos asegure que la solución obtenida es óptima.

Esta notación algorítmica puede escribirse así (libro de Brassard página 214):

```
funcion voraz (C: Conjunto): conjunto
    { C es el conjunto de candidatos }
    S ← ∅          { Construimos la solución en el conjunto S }
    mientras C ≠ 0 y ¬solución (S) hacer
        x ← seleccionar (C)
        C ← C\{x};
        si factible (S ∪ {x}) entonces S ← S ∪ {x}
    si solución (S) entonces devolver S
    si no devolver "no hay solución"
```

El esquema general para el libro de problemas es:

```
fun voraz (C: Conjunto) dev (S:conjunto)
    S ← ∅
    mientras ¬solución (S) y C ≠ ∅ hacer
        x ← elemento que maximiza objetivo (x)
        C ← C\{x};
        si completable (S ∪ {x}) entonces
            S ← S ∪ {x};
        fsi
    fmientras
    dev S
ffun
```

donde:

C: Conjunto de vértices del grafo.
S: Recubrimiento del grafo.

La forma más "intuitiva" de garantizar que el recubrimiento sea mínimo es tomar vértices de los que salgan muchas aristas, esto es, elegir vértices con mayor grado.

La función de selección debe escoger el candidato con más vértices de los que aun están en el conjunto de candidatos. Para ello, cada vez que seleccionemos un vértice tendremos que disminuir en uno el grado de cada uno de los vértices candidatos con el. Hay que seguir los siguientes pasos:

1 Elegir el vértice de mayor grado.
2 Recorrer su lista asociada (aquellos vértices con los que está conectado) y restar 1 al grado de cada uno de ellos en el campo correspondiente en el vector de vértices.

De esta forma, cuando el grado de todos los candidatos sea cero todas las aristas del grafo tocan al conjunto de selección y será, por tanto, un recubrimiento.

Según este criterio, las funciones del esquema principal tendrán el siguiente significado:

- **solución(S)**: Todas las aristas del grafo tocan al menos un vértice de S.
- **objetivo(x)**: Grado del vértice.
- **factible(S) (o completable)**: Siempre es cierto, ya que se selecciona cada vez un único vértice correcto.

En estos problemas añadimos la **demostración de optimalidad**, que será la siguiente:

Este problema es un ejemplo de que los algoritmos voraces, en determinadas ocasiones, no proporcionan la solución óptima, aunque si una buena aproximación a la misma. Ocurre también en el ejemplo que se cita en el capítulo de metodología del texto con el algoritmo de devolución de monedas en el sistema británico antiguo.

Aunque la intuición nos indique, a veces una heurística que "no puede fallar", un sencillo contraejemplo nos puede hacer ver más claramente lo dificultoso, a veces, del estudio de algoritmia. Se deja al lector que busque dicho contraejemplo para este caso.

3. Estructuras de datos

En el problema intervienen grafos, vértices de un grafo y conjunto de vértices. Para representar el grafo podemos utilizar cualquiera de las dos formas habituales, teniendo en cuenta que el uso de la **matriz de adyacencia** hará menos eficiente el algoritmo. Aquí representar el grafo como un vector de vértices, teniendo asociado cada vértice una lista enlazada con los vértices adyacentes a éste. Como los costes no juegan ningún papel en el algoritmo, lo excluiremos (por comodidad) de la estructura de datos.

```
grafo = vértice [1..N] de vértice

vertice = tupla
    indice: entero                          // Posición en el vector
    grado: entero                           // Grado del vértice
    adyacentes: apuntador a nodo_adyacente

nodo_adyacente = tupla
    adyacente: entero
    siguiente: apuntador a nodo_adyacente
```

4. Algoritmo completo a partir del refinamiento del esquema

Adaptamos el algoritmo general a lo dicho anteriormente:

```
fun recubrimiento-mínimo (G:grafo) dev (S: conjunto de vértices)
    S ← ∅
    mientras C ≠ 0 y ¬solución (S) hacer
        x ← seleccionar (C)    // Elemento que maximiza objetivo (x)
        C ← C\{x};
        disminuir_grado (x,C)
        fmientras
    dev S
    ffun
```

Las funciones son las siguientes:

Solución: El conjunto S será una solución cuando el grado de todos los elementos que restan en C será cero. Será en pseudocódigo así:

```
fun solución (C: Conjunto de vértices) dev (b:booleano)
    b ← cierto
    para c en C hacer
        b ← b and (c.grado = 0)
        fpara
    dev b
ffun
```

Tal como comentábamos anteriormente, algunas soluciones aportadas en estos ejercicios no cuadran, debido a sintaxis, debido a que no es "entendible", etc. En este caso, es una de ellas, ya que dentro del bucle "para" hay un bucle "si" que no comprendo muy bien que hace, por lo que calculamos la variable b (que es un booleano) para averiguar si es solución o no. Cabe destacar que la solución no es mía personal, la aportó otro compañero en los cursos virtuales, por ello gracias ;)

La **función de selección** devuelve el grado del vértice en consideración:

```
fun seleccionar (v: vertice) dev (g:entero)
    dev v.grado
ffun
```

Es otra función modificada, ya que devuelven vértice, cuando debería ser (creo yo) v.grado, que es el parámetro dado en la función.

Por último, la función disminuir-grado resta 1 al grado de todos los elementos conectados con el vértice elegido:

```
fun disminuir_grado(v: vértice, C: conjunto de vertices)
    para w ∈ C en sucesores (v) hacer
        w.grado ← w.grado − 1
    fpara
ffun
```

5. Análisis del coste

El tamaño del problema viene dado por el número de vértices del grafo. El número de veces que se ejecuta el bucle voraz es, en el peor de los casos, n. Dentro del bucle se realizan dos operaciones:

- Encontrar el vértice de mayor grado, que tiene un coste lineal.
- Disminuir en uno el grado de los demás, que tiene también coste lineal.

De modo que el coste del bucle es $O(n)$ y el coste del algoritmo es $\boldsymbol{O(n^2)}$.

Septiembre 1996 (problema 1) (igual a 2.3 libro de problemas resueltos)

Enunciado: Un cartógrafo acaba de terminar el plano de su país, que incluye información sobre las carreteras que unen las principales ciudades y sus longitudes. Ahora quiere añadir una tabla que se recoja la distancia entre cada par de ciudades del mapa (entendiendo por distancia la longitud del camino más corto entre las dos). Escribir un algoritmo que permita realizar esta tabla.

Repuesta:

1. Elección razonada del esquema algorítmico

   Para hallar la distancia mínima desde un vértice de un grafo a cada uno de los demás vértices contamos con el **algoritmo voraz de Dijkstra**. Basta, pues, con aplicarlo para cada una de las ciudades, siendo éstas los vértices, las carreteras las aristas del grafo y sus longitudes los pesos de las aristas.

   CUIDADO: No hay que confundir este problema (de "*caminos mínimos*") con el problema de dar un *árbol de expansión mínimo*, que resuelven algoritmos como el de Prim o Kruskal. En este caso, un árbol de expansión mínimo seria un subconjunto de carreteras que conectara todas las ciudades y cuya longitud total fuera mínima; pero esa condición no nos asegura que la distancia entre cada par de ciudades sea la menor posible.

2. Descripción del esquema usado e identificación con el problema

   Tendremos el esquema general voraz:

```
funcion voraz (C: Conjunto): conjunto
    { C es el conjunto de candidatos }
    S ← ∅          { Construimos la solución en el conjunto S }
    mientras C ≠ 0 y ¬solución (S) hacer
        x ← seleccionar (C)
        C ← C\{x};
        si factible (S ∪ {x}) entonces S ← S ∪ {x}
    si solución (S) entonces devolver S
    si no devolver "no hay solución"
```

   De nuevo hemos escrito el esquema general que viene del libro de teoría, aunque como hemos dicho antes daría igual cual de los esquemas escribir, ya que harían lo mismo ambos.

   El algoritmo de Dijkstra opera como sigue:

   - En un principio, el conjunto de candidatos son todos los nodos del grafo.
   - En cada paso, seleccionamos el nodo de C cuya distancia al origen es mínima y lo añadimos a S. En cada fase del algoritmo, hay una matriz D que contiene la longitud del camino especial más corta que va hasta cada nodo del grafo (llamaremos *especial* a un camino en el que todos los nodos intermedios pertenecen a S).
   - Cuando el algoritmo termina, los valores que hay en D dan los caminos mínimos desde el nodo origen a todos los demás.

El algoritmo es el siguiente:

```
fun Dijkstra (g:grafo) dev (vector[1..n])
        D ← vector[1..n]
        C ← {2,3, … , n}
     para i ← 2 hasta n hacer D[i] ← coste(1, i, g)
        mientras C ≠ ∅ hacer
            v ← elemento de C que minimiza D[v]
            C ←eliminar (v, C)
            para cada w ∈ C hacer
                D[w] ← min(D[w], D[v] + coste(v, w, g))
            fpara
        fmientras
        dev S
ffun
```

3. Estructuras de datos

   El conjunto de ciudades y carreteras viene representado por un **grafo no orientado con pesos**. Podemos implementarlo como una matriz de adyacencia, lista de listas de adyacencias, o como sea necesario.

   Además, necesitaremos otra matriz que acumule las distancias mínimas entre ciudades y que sirva como resultado.

4. Algoritmo completo a partir del refinamiento del esquema general

   La única variación respecto al algoritmo de Dijkstra es que necesitamos saber la distancia mínima entre cada par de ciudades, no solo entre una ciudad y todas las demás. Por ello, es necesario aplicar Dijkstra *n* veces, siendo *n* el número de ciudades (en rigor, no es necesario aplicarlo sobre la última ciudad, pues los caminos mínimos a esa ciudad ya que han sido obtenidas en aplicaciones anteriores):

```
fun mapa (g:grafo) dev (vector[1..N,1..N] de entero)
    m ← vector[1..N, 1..N]
    para cada vértice v hacer
        m ← dijkstra(g, v, m)
    fpara
    dev m
ffun
```

   donde el algoritmo de Dijkstra se implementa mediante una función $Dijkstra\ (g, v, m)$ que devuelve una matriz *m* con la información añadida correspondiente a las distancias entre el grafo *v* y todos los demás grafos de *g*.

5. Estudio del coste

   El coste del algoritmo depende de la implementación para grafos que se escoja. Si se implementa como una matriz de adyacencia, sabemos que el coste del algoritmo de Dijkstra es cuadrático $(O(n^2))$. Como hemos de aplicarlo n veces (o $n-1$ veces, que es de orden n), el coste del algoritmo completo es $\boldsymbol{O(n^3)}$.

Septiembre 1996-reserva (problema 2) (igual a 2.5 libro de problemas resueltos)

Enunciado: Se planea conectar entre sí todos los pueblos de una cierta región mediante carreteras que sustituyan los antiguos caminos vecinales. Se dispone de un estudio que enumera todas las posibles carreteras que podrían construirse y cuál sería el coste de construir cada una de ellas. Encontrar un algoritmo que permita seleccionar de entre todas las carreteras posibles, un subconjunto que conecte todos los pueblos de la región con un coste global mínimo.

Respuesta:

1. Elección razonada del esquema algorítmico

   Si interpretamos los datos como un grafo en el que los pueblos son los **vértices** y las carreteras son las **aristas**, cuyos pesos son el coste de construcción, el problema no es otro que el de hallar un árbol de expansión mínimo para ese grafo. En efecto, un árbol de expansión mínimo es un conjunto de aristas que conecta todos los vértices del grafo en el que la suma de los pesos es mínima (por tanto, el coste de construir el subconjunto de carreteras es mínimo).

   Para resolverlo podemos usar cualquiera de los dos algoritmos voraces estudiados, que resuelven este problema: el de Kruskal o Prim.

2. Descripción del esquema usado e identificación del problema

   El esquema voraz se aplica a **problemas de optimización**. Consiste en ir seleccionando elementos de un conjunto de candidatos que se van incorporando a la solución. Se caracteriza porque nunca deshace una decisión ya tomada: los candidatos no vuelven a ser considerados, y los incorporados a la solución permanecen en ella hasta el final del algoritmo. Es crucial determinar la **función de selección** apropiada que nos asegura que la solución obtenida es óptima.

   En notación algorítmica puede describirse así:

```
funcion voraz (C: Conjunto): conjunto
    { C es el conjunto de candidatos }
    S ← ∅          { Construimos la solución en el conjunto S }
    mientras C ≠ 0 y ¬solución (S) hacer
        x ← seleccionar (C)
        C ← C\{x};
        si factible (S ∪ {x}) entonces S ← S ∪ {x}
    si solución (S) entonces devolver S
    si no devolver "no hay solución"
```

   Para resolver el problema utilizaremos un algoritmo voraz llamado de **Prim**, que resuelve el problema de hallar el recubrimiento mínimo de un grafo. En este algoritmo, el árbol de recubrimiento mínimo crece a partir de una raíz arbitraria. En cada fase, se añade una nueva rama al árbol ya construido, y el algoritmo se detiene cuando se han alcanzado todos los nodos. En cada paso, el algoritmo busca la arista más corta posible $\{u, v\}$ tal que *u* pertenece a *B* (conjunto de nodos) y *v* pertenece a *N* menos *B*. Entonces añade *v* a *B* y $\{u, v\}$ a *S* (conjunto de aristas). Las aristas de *S* forman en todo momento un árbol de recubrimiento mínimo para los nodos de *B*.

```
fun prim (G = ⟨N, A⟩: grafo, longitud: A → R⁺): conj. aristas
    S ← ∅;
    B ← un elemento arbitrario de N
    mientras B ≠ N hacer
    buscar e = {u, v} de longitud mínima tal que u ∈ B y v ∈ N / B
        S ← S ∪ {e};
        B ← B ∪ {v};
    fmientras
    devolver T
ffun
```

3. Estructuras de datos

   Son exactamente las mismas que el algoritmo de Prim, pues se puede aplicar directamente.

4. Algoritmo completo

   El algoritmo de Prim no necesita ninguna modificación posterior.

5. Estudio del coste

   De nuevo, el coste es exactamente el del algoritmo de Prim, es decir, $\boldsymbol{O(n^2)}$.

Febrero 1997-1ª (problema 1) (igual a 2.2 libro de problemas resueltos)

Enunciado: Dado un conjunto de n cintas no vacías con n_i registros ordenados cada uno, se pretende mezclarlos a pares hasta lograr una única cinta ordenada. La secuencia en la que se realiza la mezcla determinara la eficiencia del proceso. Diseñese un algoritmo que busque la solución óptima minimizando el número de movimientos.

Por ejemplo: 3 cintas: A con 30 registros, B con 20 y C con 10.

1. Mezclamos A con B (50 movimientos) y el resultado con C (60 movimientos), con la que realiza en total 110 movimientos.
2. Mezclamos C con B (30 movimientos) y el resultado con A (60 movimientos), con la que realiza en total 90 movimientos.

¿Hay alguna forma más eficiente de ordenar el contenido de las cintas?

Respuesta:

1. Elección razonada del esquema algorítmico

   El problema presenta una serie de elementos característicos de un esquema voraz:

   - Por un lado, se tienen un **conjunto de candidatos** (las cintas) que vamos eligiendo uno a uno hasta completar determinada tarea.
   - Por otro lado, el orden de elección de dichos elementos determina la **optimalidad** de la solución, de manera que para alcanzar una solución óptima es preciso seleccionar adecuadamente al candidato mediante un criterio determinado. Una vez escogido, habremos de demostrar que nos lleva a una solución óptima.

El criterio de elección de las cintas para alcanzar una solución óptima será el de elegir en cada momento aquella con menor número de registros.

Demostración de optimalidad:

La demostración corresponde con la de minimización del tiempo en el sistema, dada ya en ejercicios antes, por lo que evitamos escribirla de nuevo.

2. Descripción del esquema usado e identificación con el problema

El esquema voraz es:

```
funcion voraz (C: Conjunto): conjunto
    { C es el conjunto de candidatos }
    S ← ∅          { Construimos la solución en el conjunto S }
    mientras C ≠ 0 y ¬solución (S) hacer
        x ← seleccionar (C)
        C ← C\{x};
        si factible (S ∪ {x}) entonces S ← S ∪ {x}
    si solución (S) entonces devolver S
    si no devolver "no hay solución"
```

Hemos de particularizar las siguientes funciones:

- **Solución (S)**: El número de cintas, que es *n*.
- **Objetivo (x)**: Función que devuelve la cinta con menor número de registros de entre el conjunto de cintas disponibles.
- **Factible (x) (o completable)**: Esta función es siempre cierta, pues cualquier orden de combinación es válido.

3. Estructura de datos

Se utilizarán vectores de n valores naturales para representar los conjuntos. Por ejemplo, para el conjunto *C* se define la variable *c* como vector de naturales, siendo $c[i] = n_i$ la expresión de que la cinta i consta de n_i registros. El conjunto S puede representarse de manera análoga. Para representar la ausencia de un elemento puede usarse cualquier marcador (por ejemplo, el valor ∅).

4. Algoritmo completo a partir del refinamiento del esquema general

Retocando el **esquema general** tenemos:

```
fun voraz (C: vector) dev (s:vector)
    para i ← 0 hasta n hacer s[i] ← 0
    i ← ∅
    mientras i ≤ n hacer
        x ← seleccionar (C)
        c[i] ← 0
        s[i] ← x
        i ← i + 1;
    fmientras
    dev S
ffun
```

Esta solución está modificada respecto de la solución aportada en el libro de problemas. Se ha añadido una línea (la de $i \leftarrow i + 1$) y se ha modificado la línea $c[i] \leftarrow 0$. Con esto trato de decir, que no es seguro que esté correcta la respuesta, sólo que pienso que había algunas erratas.

La única función (de selección) por desarrollar es aquélla que obtiene en cada momento la cinta con menor número de registros de entre las cintas no usadas todavía y almacenadas en el vector de cintas. La función devuelve la cinta, pero no la elimina del conjunto de candidatos. Los argumentos son *c*, vector de cintas y *cinta* que es un vector de n_i registros.

```
fun seleccionar (c: vector) dev (cinta:vector)
    min ← 1
    para j ← 1 hasta n hacer
        si c[j] < c[min] entonces min ← j        // Escoge la de menor tamaño
        fsi
    fpara
    dev c[min]
ffun
```

5. Estudio del coste

   La función objetivo (para nosotros seleccionar) tiene coste de $O(n)$ y el bucle principal ("mientras") se repite n veces, por lo que el coste es $\boldsymbol{O(n^2)}$.

Problema 2.2 del libro de problemas resueltos (sin correspondencia con ningún ejercicio de examen)

Enunciado: Consideramos un conjunto de programas $p_1, p_2, \dots, p_n$ que ocupan un espacio en cinta $l_1, l_2, \dots, l_n$. Diseñar un algoritmo para almacenarlos en una cinta de modo que el tiempo medio de acceso sea mínimo.

Respuesta:

1. Elección razonada del esquema algorítmico

   Si un programa p_i ocupa la posición x_i el tiempo de acceso a ese programa es la suma del tiempo que se tarda en avanzar la arista hasta x_i y, a continuación, seguir leyéndola hasta l_i. Es decir:

   $$t_i = k * (x_i + l_i)$$

   donde la constante k no afecta al resultado.

   El tiempo medio de acceso es:

   $$T = \frac{\sum_i (x_i + l_i)}{N}$$

   El problema se ajusta al esquema voraz: podemos considerar los programas como candidatos que hay que ir seleccionando en el orden adecuado. Cada orden posible nos da una solución y buscamos entre ellas la solución óptima.

   Si tomamos como función de selección el escoger el programa más corto de los que aún no han sido colocados, el esquema voraz resuelve el problema. Un punto clave, para no confundir los problemas voraces con los de backtracking es asegurarse de que no es necesario deshacer decisiones ya tomadas. Debemos hacer una **demostración de optimalidad** para el algoritmo, una vez especificado, para asegurarnos de este extremo.

Demostración de optimalidad

Hemos cambiado la demostración de optimalidad a su sitio idóneo dentro del problema. El tiempo medio de acceso del sistema T es:

$$T = \frac{\sum_i (x_i + l_i)}{N} = \frac{\sum_i \left(\left(\sum_{j=1}^{i} l_j \right) + l_i \right)}{N} = l_1 + \frac{n-1}{n} * l_2 + \frac{n-2}{n} * l_3 + \cdots + \frac{1}{n} l_n.$$

Hemos tenido en cuenta que x_i es la suma de las longitudes de los programas que están antes de p_i, es decir, $x_i = \sum_{j=1}^{i-1} l_j$.

Para minimizar ese sumatorio es necesario colocar en la primera posición (l_i) el programa más corto, para minimizar el factor más grande l_1 y así sucesivamente. **Supongamos** que hubiera dos elementos $l_i < l_j$ con $i < j$. El sumatorio podría entonces reducirse intercambiando sus posiciones, así que se disminuirá la contribución al sumatorio del término:

$$\frac{n-i+1}{n} * l_i + \frac{n-j+1}{n} * l_j$$

Por tanto, en la única posición en la que el sumatorio no puede ser reducido es aquella en la que los programas están ordenados con longitudes crecientes.

2. Descripción del esquema usado e identificación con el problema

El esquema voraz se aplica a **problemas de optimización**. Consiste en ir seleccionando elementos de un conjunto de candidatos que se van incorporando a la solución. Se caracteriza porque nunca deshace una decisión ya tomada: los candidatos ya desechados no vuelven a ser considerados y los incorporados a la solución permanecen en ella hasta el final del algoritmo.

En notación algorítmica:

```
funcion voraz (C: Conjunto): conjunto
    { C es el conjunto de candidatos }
    S ← ∅          { Construimos la solución en el conjunto S }
    mientras C ≠ 0 y ¬solución (S) hacer
        x ← seleccionar (C)
        C ← C\{x};
        si factible (S ∪ {x}) entonces S ← S ∪ {x}
    si solución (S) entonces devolver S
    si no devolver "no hay solución"
```

En este caso:

- **C**: Es el conjunto de programas.
- **S**: No es exactamente un conjunto, sino una lista de programas, ya que nos interesa conservar la información sobre el orden en el que los programas fueran incorporados a la solución.

3. Estructuras de datos

Hay dos posibles representaciones:

- El conjunto de programas puede ser implementado mediante un vector que cumpla $C[i] = l_i$.
- Otra opción es definir un tipo de datos programa que consista en una etiqueta (un número entero que lo identifica) y una longitud:

```
tipo programa = tupla
    identificador: entero
    longitud: entero
```

Un conjunto de programas puede entonces implementarse mediante una lista de programas. De esta forma, ese tipo nos servirá también para implementar S.

4. Algoritmo completo a partir del refinamiento del esquema general

En vista de que tanto la función *solución* como la *factible* son triviales, podemos reescribir el esquema para este problema simplificándolo:

```
fun voraz (C: conjunto) dev (S: conjunto)
    S ← ∅
    mientras C ≠ ∅ hacer
        x ← elemento que maximiza objetivo (x)
        C ← C\{x}
        S ← S ∪ {x}
    fmientras
    dev S
ffun
```

La **función objetivo** devuelve la longitud del programa:

```
fun objetivo (p: programa) dev entero
    dev p.longitud
ffun
```

5. Estudio del coste

El bucle principal se recorre n veces. Dentro del bucle hay estas operaciones:

- Selección del candidato adecuado: $O(n)$
- Eliminación del candidato $O(n)$ si el conjunto es una lista.
- Adición del candidato seleccionado al conjunto solución: $O(1)$

Tendremos, por tanto, que el coste total es $O(n^2)$.

Para mejorar el coste usaremos un montículo invertido o de mínimos, como en ocasiones anteriores, teniendo en la raíz el programa más corto. Haría estas operaciones:

- Eliminar el programa de C tendría coste $O(\log(n))$, será lo que tarde en restaurar la propiedad del montículo cuando se extrae la raíz. Como el bucle se ejecuta n veces el coste es $O(n * \log(n))$.
- Hay que considerar una nueva instrucción para inicializar el conjunto de candidatos como un montículo de mínimos, que tiene coste $O(n * \log(n))$.

Por tanto, el coste total del algoritmo es $\boldsymbol{O(n * \log(n))}$.

Septiembre 2003 (problema)

Enunciado: Una operadora de telecomunicaciones dispone de 10 nodos conectados todos entre sí por una tupida red de conexiones punto a punto de fibra óptica. Cada conexión $c(i,j)$ entre el nodo i y j (con $i,j \in \{1..10\}$) tiene un coste asignado que sigue la fórmula $c(i,j) = (i + j)\ MOD\ 8$. La operadora quiere reducir gastos, para lo cual está planificado asegurar la conectividad de su red de nodos minimizando el coste. Diseñar un algoritmo que resuelva el problema y aplicarlo a los datos del enunciado.

Respuesta: Se trata de un grafo dirigido de 10 nodos $n_1, n_2, \dots, n_{10}$ con una matriz simétrica de costes:

	1	2	3	4	5	6	7	8	9	10
1	-	3	4	5	6	7	0	1	2	3
2	3	-	5	6	7	0	1	2	3	4
3	4	5	-	7	0	1	2	3	4	5
4	5	6	7	-	1	2	3	4	5	6
5	6	7	0	1	-	3	4	5	6	7
6	7	0	1	2	3	-	5	6	7	0
7	0	1	2	3	4	5	-	7	0	1
8	1	2	3	4	5	6	7	-	1	2
9	2	3	4	5	6	7	0	1	-	3
10	3	4	5	6	7	0	1	2	3	-

Se trata de conseguir minimizar el coste de los enlaces asegurando únicamente la conectividad de la red.

El enunciado describe un problema de optimización en el que se nos pide que el grafo sea conexo ("asegurar la conectividad de la red") y contenga un árbol de expansión mínimo ("que el coste sea mínimo"), ya que la conectividad se asegura no dejando subgrafos no conexos.

1. Elección razonada del esquema algorítmico

   Con las condiciones descritas podemos usar algoritmos que resuelvan el problema del árbol de expansión mínimo, dentro de la familia de los algoritmos voraces.

2. Descripción del esquema usado e identificación con el problema

   Se elige cualquiera de los algoritmos expuestos en el temario, **Kruskal o Prim**, por ejemplo, este último, siendo el enunciado informal sacado del resumen el siguiente:

```
funcion prim (G = ⟨N, A⟩: grafo, longitud: A → R⁺): conj. aristas
    { Iniciación }
    T ← ∅;
    B ← { un miembro arbitrario de N }
    mientras B ≠ N hacer
        buscar e = {u, v} de longitud mínima tal que u ∈ B y v ∈ N/B
            T ← T ∪ {e};
            B ← B ∪ {v};
    devolver T
```

Hemos tomado 1 como nodo arbitrario. El conjunto B va a ir conteniendo los nodos del subgrafo ya conexo y el conjunto T ira teniendo en cada iteración aquellas aristas del árbol de expansión mínimo que contiene los nodos de B. El conjunto de candidatos es B, la condición de finalización es $B = N$ y la función de optimización es elegir aquella arista del subgrafo B que conecte con algún nodo de $N \setminus B$ con menor coste.

Una **aplicación al problema**, que hace que se vea con más claridad porque se ha escogido el esquema voraz (reubicación de apartados de la solución por el autor):

Tenemos los siguientes conjuntos inicialmente $B = \{1\}$ y la arista mínima entre un nodo de B y otro de N\B es $u = (1,7)$ con valor 0.

Los valores de B y la u elegida en cada momento evolucionan como sigue:

$B = \{1,7\}$ $u = (7,9)$ Coste: 0

$B = \{1,7,9\}$ $u = (7,2)$ Coste: 1

$B = \{1,2,7,9\}$ $u = (2,6)$ Coste: 0

$B = \{1,2,6,7,9\}$ $u = (6,10)$ Coste: 0

$B = \{1,2,6,7,9,10\}$ $u = (6,3)$ Coste: 1

$B = \{1,2,3,6,7,9,10\}$ $u = (3,5)$ Coste: 0

$B = \{1,2,3,5,6,7,9,10\}$ $u = (6,3)$ Coste: 0

$B = \{1,2,3,5,6,7,8,9,10\}$ $u = (9,8)$ Coste: 1

$B = \{1,2,3,4,5,6,7,8,9,10\}$ $u = (9,8)$ Coste: 1

Coste del árbol de expansión mínimo: 4

NOTA DEL AUTOR: Con calma he visto este ejercicio y veo que esta aplicación es algo rara, porque vuelve a coger la misma arista, lo cual según el algoritmo voraz no puede ocurrir nunca, por llegar a ciclos. No entiendo, por tanto, esta aplicación.

Pondremos la **demostración de optimalidad** a continuación:

El algoritmo de Prim encuentra la solución óptima. Se puede demostrar por inducción sobre T que añadir la arista más corta $\{e\}$ que sale de T forma en $T \cup \{e\}$ un árbol de recubrimiento mínimo que contendrá al final $n - 1$ aristas y todos los nodos del grafo G. Para esta demostración tendremos en cuenta este lema (sacado del resumen):

> $Lema$ 6.3.1 Sea $G = \langle N, A \rangle$ un grafo conexo no dirigido en el cual está dado la longitud de todas las aristas. Sea $B \subset N$ un subconjunto estricto de los nodos de G. Sea $T \subseteq A$ un conjunto prometedor de aristas, tal que no haya ninguna arista de T que salga de B. Sea v la arista más corta que sale de B (o una de las más cortas si hay empates). Entonces $T \cup \{v\}$ es prometedor.

3. Estructuras de datos

El grafo se representara mediante una **matriz de costes**. La estructura de datos tendrá un método que implementa el cálculo de la distancia entre dos nodos. En el caso de esta implementación, la distancia entre dos nodos *i* y *j* es el valor de la matriz de distancias y su coste $O(1)$.

4. Algoritmo completo a partir del esquema general

Tendremos este algoritmo completo a partir del tomado en el punto anterior. Sería la misma que el algoritmo más refinado de la teoría:

```
funcion prim (L[1..n,1..n]): conj. aristas
    { Iniciación: solo el nodo 1 se encuentra en B }
    T ← ∅;     { Contendrá las aristas del árbol de recubrim. mínimo }
    para i ← 2 hasta n hacer
        mas próximo [i] ← 1
        distmin [i] ← L[i, 1]
    { Bucle voraz }
    repetir n − 1 veces
        min ← ∞
        para j ← 2 hasta n hacer
            si 0 ≤ distmin[j] < min entonces min ← distmin[j]
                                             k ← j
        T ← T ∪ { más próximo [k], k }
        distmin [k] ← −1
        para j ← 2 hasta n hacer
            si L[j, k] ≤ distmin[j] entonces distmin[k] ← L[j, k]
                                            más próximo [j] ← k
    devolver T
```

5. Coste del algoritmo

El coste del algoritmo de Prim es $\boldsymbol{O(n^2)}$, que puede mejorarse utilizando una representación de montículos para el vector $distmin[]$.

3ª parte. Problemas de exámenes sin solución o planteados:

Febrero 1997-2ª (problema 1)

Enunciado: Queremos grabar *n* canciones de duraciones $t_1, t_2, \dots, t_n$ en una cinta de audio de duración $T < \sum_{i=1}^{n} t_i$.
Diseñar un algoritmo que permita almacenar el máximo número de canciones en el espacio disponible.

Respuesta: No hay solución oficial de este ejercicio, por lo que al ser de optimización puede ser o bien un esquema voraz o bien un ramificación y poda. Hemos tomado parte de la solución (el esquema era nuestra duda) del libro de *Estructuras de datos y métodos algorítmicos. Ejercicios resueltos* de Narciso Martí, Y. ortega, J.A. Verdejo, ejercicio 12.2.

Por tanto, en este caso, tenemos que en este caso iríamos ordenaremos los programas de menor a mayor tamaño y para cada programa grabaremos si entra en el disco. Cuando los siguientes no caben significa que no entrarán ninguno más. Recordemos que esta solución se había dado en el tema del problema de la mochila, en el que al verificar la optimalidad (lo más importante del esquema voraz) se tenía que usar el método de *reducción de diferencias*, en el que se comparaban dos soluciones y se verificaba que diferencias había entre ellas. Para eso habría que remontarse a la teoría y a los ejercicios que hemos visto previamente.

Febrero 1998-1ª (problema 1)

Enunciado: En la compleja red de metro de Japón, la cantidad que se paga por un billete es proporcional a la distancia que se recorre. Por tanto, es necesario instalar en cada estación un panel informativo que informe del precio a cualquier otra estación de la red. Describir un algoritmo que deduzca la información de todos estos paneles, basando el cálculo en la suposición de que el viajero se trasladará de una estación a otra por el camino más corto.

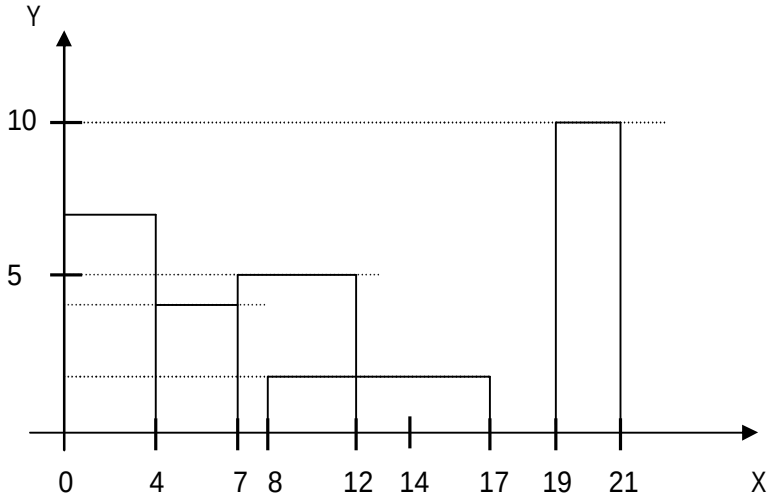
Respuesta: Este ejercicio nos está hablando del metro de Japón, considerando este como un grafo, en el que hay que seleccionar las aristas más cortas. Por ello y sin ánimo de equivocarme estimo que corresponde con el algoritmo de Dijkstra, en el que se calcula la distancia menor entre cada nodo, seleccionando el camino menor. Recuerda enormemente al **problema 2.3** del libro de problemas resueltos de nuestra bibliografía básica.

Febrero 1998-2ª (problema 1)

Enunciado: Sobre un eje de coordenadas positivo se tienen representados edificios en forma de rectángulos. Cada rectángulo descansa sobre el eje horizontal y se representa por sus abscisas inicial y final y por su altura. La línea de horizonte es el contorno que forman los edificios. Se pide programar un algoritmo eficiente que encuentre la línea de horizonte que forma un conjunto de edificios.

Ejemplo: Una ciudad $C = \{(0,4,7), (2,14,4), (7,12,5), (8,17,2), (19,21,10)\}$ tendría una línea de horizonte $H = \{(0,4,7), (4,7,4), (7,12,5), (14,17,2), (17,19,0)(19,21,10)\}$

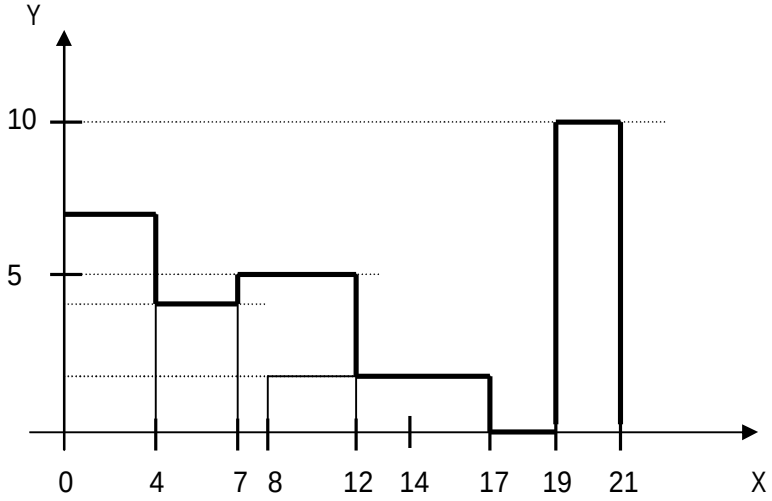
Respuesta: No sé exactamente el tipo de problema al que pertenece el mismo. Lo he preguntado y al final hemos llegado a la conclusión que puede ser un voraz. Realmente como digo no lo tengo nada claro, solo sé que los candidatos son los edificios y se trata de un problema de maximización (muy disimulado), ahora explico el porqué de este problema, representándolo gráficamente:



Hemos representado el ejemplo anterior. Pasamos a explicar que es cada valor:

- La primera cifra es el eje de abscisas inicial, es decir, donde empieza el rectángulo del edificio.
- La segunda cifra es el eje de abscisas final, es decir, donde finaliza el rectángulo del edificio.
- La última cifra es la altura del edificio.

Resaltaremos con negrita el contorno que nos piden que hagamos.



Esto que hemos resaltado es el contorno solución que nos han dado en el enunciado.

Como he puesto anteriormente no tengo nada claro que el esquema sea el voraz, ya que no veo ni la función de selección que haga el contorno ni nada así, grafo descartado (supongo), porque no lo parece. En conclusión, que no sabría cómo se resolvería, pero al menos queda para estrujarnos la cabeza.

Una posible solución podría ser ordenar las alturas por orden creciente y verlas, pero me temo que es algo así como escoger la mayor longitud de los edificios de entre dos que se solapen, como pueden ser los edificios (7,12,5) y (8,17,2).

Septiembre 1998 (problema 1)

Enunciado: Dado un conjunto de *n* rectángulos cuyos lados son paralelos a los ejes del plano, hallar el rectángulo intersección de todos los rectángulos mediante un algoritmo eficiente.

Respuesta: Este ejercicio se parece mucho al anterior, el de los rectángulos. Lo hemos puesto separado, ya que el enunciado en cierta manera es distinta. De nuevo, no resolveremos este ejercicio, y nos remontaremos al anterior.

Septiembre 1998 (problema 2)

Enunciado: Una *cadena euleriana* en un grado no orientado es una cadena que une cada arista del grafo exactamente una vez. Escribir un algoritmo que decida si un grafo dado tiene una cadena euleriana y, si es así, que devuelva esa cadena.

Respuesta: Este ejercicio se asemeja bastante al **problema 2.4** del libro de problemas, en el que se nos pedía un recubrimiento mínimo. De hecho, hemos puesto separado este ejercicio, como en otras ocasiones por el enunciado que aunque realmente piden lo mismo suena a distinto y así se hace más rico el estudio de los problemas.

Febrero 2000-2ª (problema)

Enunciado: La Base Aérea de Gando (Gran Canaria) posee una flota variada de n cazas de combate c_i $(con\ i \in \{1..n\})$. Cada caza tiene que salir del bunker y esperar un tiempo de rodadura k_i más un tiempo despegue t_i para estar en el aire y operativo. Durante este proceso la nave es vulnerable. Suponiendo que se produce un ataque sorpresa, construir un algoritmo que averigüe el orden de despegue de las aeronaves de manera que se minimice el tiempo medio durante el cual son vulnerables.

Supongamos ahora que cada aeronave c_i posee un índice acumulativo b_i de importancia estratégica siempre que despegue antes de la ranura temporal h_i $(con\ i \in \{1..n\})$. si queremos maximizar la importancia estratégica una vez que hayan despegado todas. ¿Qué tipo de problema es éste? ¿Con que esquema se resuelve? Explica en un párrafo breve el funcionamiento del algoritmo.

Respuesta: Este ejercicio es el típico con dos partes, que se resuelven empleando distintos esquemas algorítmicos, aunque son de distintos temas los veremos juntos (por no separar el enunciado). En este caso, el primero de ellos es un **esquema voraz**, en el que como hemos visto en este tema numerosas ocasiones se emplearía para ello la planificación en tiempo fijo, ya visto en temas anteriores, por lo que dejaríamos el ejercicio para su resolución posterior.

El segundo de ellos igualmente es un problema de optimización, sólo que nos dan una ranura temporal h_i, además del tiempo que tarda en despegar cada nave. Este problema, por tanto, se resolvería empleando un esquema de **ramificación y poda**, además se asemeja mucho al de Febrero de 2008-1ª semana (véase ejercicios tema 9), en la que el tío Facundo quiere recopilar una huertas con el máximo beneficio.

Febrero 2001-2ª (problema)

Enunciado: Se tiene que organizar un torneo con n participantes (con n potencia de 2). Cada participante tiene que competir exactamente una vez con todos los posibles oponentes. Además cada participante tiene que jugar exactamente 1 partido al día. Se pide construir un algoritmo que permita establecer el calendario del campeonato para que concluya en $n-1$ días. Sugerencia: Dos grupos disjuntos de m jugadores juegan entre ellos m días mediante rotación. Ejemplo: $\{a,b,c\}$ contra $\{d,e,f\}$ juegan: Día 1: ad, be y cf. Día 2: ae, bf y cd y finalmente día 3: af, bd y ce.

Respuesta: No estoy segura de la solución, pero creo que es como el **problema 2.4**, donde pedían recubrimiento de vértices de un grafo. Entiendo que es un grafo con todas las posibles uniones y cada vez hay una posible arista por día jugado que uniría dos equipos distintos. Por tanto, este ejercicio, al no estar resuelto se deja sólo planteado. Por ello, estimo que se usaría un algoritmo de Prim o Kruskal, llegando a un recubrimiento mínimo, que es lo que parece que solicitan.

Diciembre 2003 (problema) (igual a problema Febrero 2007-1ª)

Enunciado: Hoy es un día duro para el taller Sleepy. Llegan las vacaciones y a las 8:00 de la mañana n clientes han pedido una revisión de su coche. Como siempre, todos necesitan que les devuelvan el coche en el menor tiempo posible. Cada coche necesita un tiempo de revisión r_i y al mecánico le da lo mismo por cuál empezar: sabe que en revisar todos los coches tardará lo mismo independientemente del orden que elija. Pero al jefe del taller no le da lo mismo, la satisfacción de sus clientes es lo que importa: es mejor tener satisfechos al mayor número de ellos. Al fin y al cabo, la planificación la hace él y, evidentemente, un cliente estará más satisfecho cuanto menos tarden en devolverle el coche. Implementar un programa que decida el orden en el que revisar uno a uno los coches para maximizar la satisfacción de los clientes de Sleepy.

Respuesta: Este ejercicio no lo voy a resolver, ya que se trata exactamente el mismo planteamiento que el de **los dentistas**, es decir, ordenar los clientes del taller por orden creciente de tiempos e ir atendiéndoles. La demostración de optimalidad ya se ha visto varias veces, tanto en la teoría (en el resumen) como en los ejercicios resueltos. Lo único el algoritmo, que sería lo más complicado, pero aun así es la aplicación casi exacta del esquema general, en el que se ordenarían los clientes $(coste\ O(n*\log(n)))$, luego se escogerían por este orden y se devolvería la suma de tiempos.

Septiembre 2004-reserva (problema)

Enunciado: Sea una red de compartición de ficheros, similar a las que actualmente se utilizan para intercambiar globalmente archivos por internet. Esta red se encuentra formada por n servidores, siendo todos ellos capaces de distribuir un número n de archivos, de tamaño T_i Kilobytes, a diferentes velocidades de transmisión. La velocidad de transmisión de datos de cada uno de los servidores viene determinada por una tabla de velocidades de transmisión S, donde S_{ij} es la velocidad de transmisión del servidor i para el archivo j (en K/seg). Se pide diseñar un algoritmo capaz de repartir la descarga de los n archivos entre los n servidores disponibles, minimizando el tiempo global de descarga de todos los archivos. La función deberá indicar el tiempo óptimo de descarga, así como los servidores desde los que serán descargados los n archivos. Suponga que la descarga se lleva a cabo de manera secuencial, lo que significa que no es posible descargar más de un archivo al mismo tiempo.

Tome como ejemplo ilustrativo de los datos de entrada una red de compartición de 3 ficheros (A1, A2 y A3) en 3 servidores distintos (S1, S2 y S3). El tamaño de los 3 ficheros es $T_1 = 100K$, $T_2 = 200K$, $T_3 = 300K$ y la velocidad de transmisión de los 3 servidores viene dado por la matriz:

		Servidores		
		S1	S2	S3
Archivos	A1	50 K/seg	12 K/seg	6 K/seg
	A2	10 K/seg	20 K/seg	50 K/seg
	A3	200 K/seg	50 K/seg	1 K/seg

Respuesta: Nos evitaremos dar más detalles, ya que el ejercicio no está resuelto. Sólo decir que este ejercicio es igual al anterior, al del taller Sleepy, aunque en vez de tener un único servidor se tienen n servidores, por lo que hay que distribuir en n servidores distintos. Es decir, se repartirán los tiempos entre los n servidores, como en el ejemplo anterior. Resumiendo, sin decir más, nos quedaría el tiempo asociado algo así (tomado del libro de Martí):

$$T(P) = \sum_{j=1}^{s} \sum_{k=1}^{n_j} (n_j - k + 1) t_{(k-1)s+j}$$

siendo

$$n_j = \begin{cases} n \; div \; s + 1 & si \; j \leq (n \; mod \; s) \\ n \; div \; s & si \; j > (n \; mod \; s) \end{cases}$$

La *demostración de optimalidad* se hará como en los casos anteriores, basándose en el T (P) antes escrito. Ya lo hemos visto, como digo no lo vuelvo a poner de nuevo.

Septiembre 2006-reserva (problema)

Enunciado: Un repartidor de pizzas tiene que entregar K pedidos de diferente valor de recaudación como mucho hasta la ranura de tiempo concreta que tiene asignada en la tabla adjunta.

Pedido i:	1	2	3	4	5	6	7	8	9
Ranura:	1	5	5	6	6	4	4	2	2
Recaudación:	60	70	80	20	20	30	50	50	90

Si un pedido se entrega tarde la recaudación es 0. Construir un algoritmo que devuelva un plan de trabajo para el repartidor que maximice el beneficio. Aplíquelo al ejemplo detallando TODOS los pasos. La resolución del problema debe incluir, por este orden: elección razonada del esquema algorítmico y esquema, algoritmo completo a partir del refinamiento del esquema general y estudio del coste del algoritmo desarrollado.

Respuesta: Este problema que yo sepa es similar al visto anteriormente de ordenar los pedidos por orden decreciente de beneficios. Es como el de la minimización en tiempo del sistema que hemos visto previamente en numerosos ejercicios.

Ejercicios resueltos de programación 3

Tema 7. Divide y vencerás.

Alumna: Alicia Sánchez
Centro: UNED-Las Rozas (Madrid)

De nuevo, haremos la distinción entre estas partes en el documento, por lo que el *índice* es:

Introducción teórica:

Vamos a ver una breve introducción teórica sobre este tema, como sigue:

Tendremos el siguiente esquema, que está sacado del libro de problemas, ya que creo que es más claro como lo ponen allí que en el libro de teoría (Brassard):

```
fun divide-y-vencerás (problema)
    si suficientemente-simple (problema) entonces
        dev solucion-simple (problema)
    si no                                  { No es solución suficientemente simple }
        {p_1 .. p_k} ← decomposicion (problema)
        para cada p_i hacer
            s_i ← divide-y-vencerás (p_i)
        fpara
        dev combinacion (s_i ... s_k)
    fsi
ffun
```

Las funciones que han de particularizarse son:

- **suficientemente-simple**: Decide si un problema está por debajo del tamaño umbral o no.
- **solucion-simple**: Algoritmo para resolver los casos más sencillos, por debajo del tamaño umbral.
- **descomposicion**: Descompone el problema en subproblemas en tamaño menor.
- **combinacion**: Algoritmo que combina las soluciones a los subproblemas en solución al problema del que provienen.

Algunos algoritmos de divide y vencerás no siguen exactamente este esquema, puesto que hay casos en los que no tiene sentido reducir la solución de un caso muy grande a la de uno más pequeño. Entonces, divide y vencerás recibe el nombre de **reducción** (**simplificación**).

Para que el enfoque de divide y vencerás merezca la pena es necesario que se cumplan estas **tres condiciones**:

1. La decisión de utilizar el subalgoritmo básico (suficientemente-simple) en lugar de hacer llamadas recursivas debe tomarse *cuidadosamente*.
2. Tiene que ser posible descomponer el ejemplar y en subejemplares y recomponer las soluciones parciales de forma bastante eficiente.
3. Los subejemplares deben ser en la medida de lo posible aproximadamente del mismo tamaño.

1ª parte. Cuestiones de exámenes:

Trataremos de distinguir dos tipos de ejercicios que veremos en esta parte:

- Ejercicios directamente deducidos de la teoría, es decir, las típicas ordenaciones por fusión, ordenaciones rápidas, búsquedas binarias, etc. Se solicitarán ejemplos de este tipo o bien su código.
- Ejercicios **no** directamente deducidos de la teoría, es decir, aquellos en los que piden por ejemplo ordenar votos con coste lineal. Esto implicara usar los esquemas vistos en la teoría, pero de modo más o menos indirecto o no especificado en el enunciado de modo claro.

Hemos tratado de definir los dos tipos, para que así se pueda estudiar los ejercicios, asociándolo a cada uno de ellos.

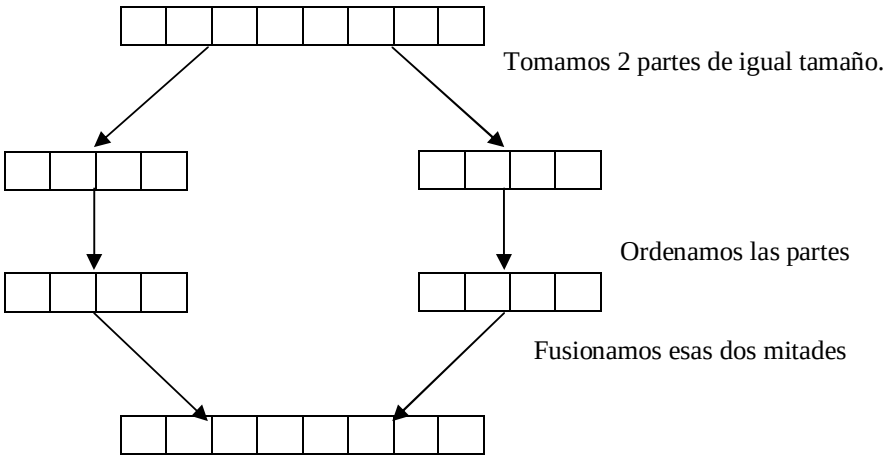
Febrero 2000-1ª (ejercicio 1)

Enunciado: Comparar la eficiencia de los algoritmos de ordenación quicksort (ordenación rápida) y mergesort (ordenación por fusión).

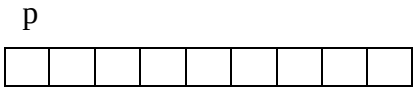
Respuesta: El algoritmo *quicksort* emplea un tiempo promedio de $n * \log(n)$, en el peor caso de n^2. El algoritmo de ordenación por fusión (*mergesort*) utiliza un tiempo de $n * \log(n)$ (siempre observando la precaución de equilibrar los subcasos a tratar). Pese a esto, en la práctica quicksort es más rápido en un tiempo *c* constante. Además, el algoritmo *mergesort* requiere de un **espacio extra** para tratar los distintos casos si perder eficiencia (hacer la ordenación *in situ* asociado un incremento de la constante oculta bastante alto).

Como añadido a la solución del problema veremos estos ejemplos, el primero de ellos el del mergesort u ordenación por fusión y el segundo quicksort u ordenación rápida. Pasamos a verlos:

El algoritmo de ordenación por fusión (mergesort) realiza lo siguiente:



El algoritmo de ordenación rápida (quicksort) hace:



Recordemos que según la colocación del pivote tendremos estos casos, que veremos brevemente:

Mejor caso: Pivote exactamente en el centro tras ordenarlo. Coste $\theta(n * log(n))$.

Peor caso: Pivote en la primera posición tras ordenarlo. Coste $\theta(n^2)$.

Febrero 2000-2ª (ejercicio 1)

Enunciado: Poner un ejemplo en el que un enfoque de Divide y Vencerás nos lleve a un coste exponencial de forma innecesaria.

Respuesta: Un ejemplo claro de esto es utilizar la definición recursiva de la función de Fibonacci sin más, de este modo el siguiente algoritmo conlleva un **coste exponencial**, existiendo formas de obtener la solución en un coste inferior utilizando un enfoque *distinto* al de divide y vencerás:

```
fun fib (n)
    si (n < 2) entonces devolver n
    si no
        devolver fib (n – 1) + fib (n – 2)
    fsi
ffun
```

Septiembre 2000 (ejercicio 2)

Enunciado: La sucesión de Fibonacci se define como $fib\,(0) = fib\,(1) = 1$; $fib\,(n) = fib\,(n-1) + fib\,(n-2)$ si $n \geq 2$. ¿Qué ocurre si aplicamos una estrategia divide y vencerás para calcular $fib\,(n)$ usando directamente la definición? ¿Qué otras opciones existen?

Respuesta: Si utilizamos directamente la definición obtendremos un algoritmo con un **coste exponencial** (coste muy malo, es igual al ejercicio anterior). Dicho algoritmo es el siguiente:

```
fun fib (n)
    si (n < 2) entonces devolver n
    si no
        devolver fib (n – 1) + fib (n – 2)
    fsi
ffun
```

El principal problema es que calcularía varias veces los mismos valores, para evitar esto existe otra opción dada por el siguiente algoritmo iterativo, que requiere un *tiempo lineal*. Es el siguiente:

```
fun fib-iter (n)
    i ← 1; j ← 0
    para k ← 1 hasta n hacer
        j ← i + j
        i ← j − i
    devolver j
```

Septiembre 2000 (ejercicio 3) (parecido a ejercicio 1 de Septiembre 2002-reserva)

Enunciado: ¿Cuál es la relación entre la ordenación por selección y la ordenación por montículo? ¿Cómo se refleja en la eficiencia de los dos algoritmos?

Respuesta: La solución dada por el ejercicio es completamente del autor, por lo que como siempre no sabemos si estará bien o por el contrario hay algo incorrecto.

Nos piden en el enunciado, según entendemos, que nos piden las diferencias entre ambas, así como sus parecidos. Pasamos a definirlos y luego a responder a las preguntas. La **ordenación por selección** es aquélla en la que se toma el menor elemento de la parte desordenada y lo intercambia con el mejor que encuentra de tal manera que quede ya ordenado en su posición, mientras que la **ordenación por montículo** es aquélla en la que se ordena el vector mediante las propiedades del montículo siguientes (relación de padre e hijo en montículo de máximos):

$T[i] \geq T[2 * i]$
$T[i] \geq T[2 * i + 1]$

Se parecen en que usan el mismo vector para realizar la ordenación, mientras que se diferencian en que mientras uno emplea las propiedades del montículo teniendo coste $\log(n)$ de flotar, el otro realiza el intercambio de un elemento a otro recorriendo el vector hasta encontrar el menor elemento, lo que significa coste lineal $(O(n))$.

El **coste asintótico**, en el caso peor, del primer algoritmo de ordenación sería cuadrático $(\theta(n^2))$, mientras que la de la ordenación por montículo seria $\theta(n * log(n))$, por tanto, la segunda ordenación evidentemente es mucho más eficiente. Aun así, merecería la pena remontarse a los apuntes de estructuras de datos y rememorarlos. Trataremos de verlos a continuación.

Además de poner esto, vamos a ampliar más conocimientos tomando apuntes del libro de *estructuras de datos y algoritmos* de R. Hernández. Empezaríamos a ver la ordenación por selección y luego la ordenación por montículo, como sigue:

La **ordenación por selección** directa es aquélla cuya idea es seleccionar el menor elemento de una parte desordenada y colocarlo en la posición del primer elemento no ordenado. En un primer paso, se recorre el arreglo hasta encontrar el elemento menor. Para ello, se coloca el primer elemento en una variable temporal y se va comparando con los demás elementos del arreglo tal que si se encuentra uno menor se asigna a la variable temporal. Recorrido todo el arreglo, el elemento de la variable temporal (que será el menor) se intercambia con el de la primera posición. Seguidamente, se considera únicamente la parte del arreglo no ordenado y se repite el proceso de búsqueda el menor, y así sucesivamente. En conclusión, el algoritmo puede describirse de la siguiente forma:

- Seleccionar el elemento menor de la parte del arreglo no ordenada.
- Colocarlo en la primera posición de la parte no ordenada del arreglo.

El pseudocódigo del algoritmo (del libro de Brassard, para ello estudiamos de nuevo el tema 4 de nuestro resumen) es el siguiente:

```
procedimiento seleccionar(T(1..n))
    para i ← 1 hasta n − 1 hacer
        minj ← 1; mink ← T[i];
        para j ← i + 1 hasta n hacer
            si T[j] < minx entonces
                minj ← j;
                mink ← T[j];
            fsi
        fpara
        T[minj] ← T[i]; T[i] ← minx;
    fpara
fprocedimiento
```

Veremos un **ejemplo** que corresponde con este algoritmo para así verlo más claro, aunque en el ejercicio de Septiembre de 2002 sí que nos piden que pongamos el ejemplo. Aun así, lo pondremos:

El vector a ordenar

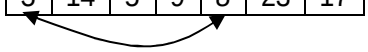
8	14	5	9	3	23	17

Seleccionamos el menor de la parte desordenada,
que sería el elemento 3

8	14	5	9	3	23	17

Se intercambian los dos elementos,
y ya queda ordenado el elemento con valor 3, resaltado en negrita

3	14	5	9	*8*	23	17



Seleccionamos el siguiente menor de la parte desordenada,
que sería el elemento 5

3	14	5	9	8	23	17

Se intercambian los dos elementos de nuevo,
y ya queda ordenado el elemento con valor 5, resaltado en negrita

3	**5**	14	9	8	23	17

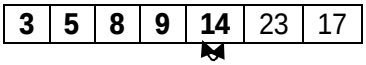
Continuamos con el procedimiento, intercambiando este

3	**5**	**8**	9	14	23	17

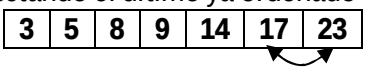
Continuamos con el procedimiento, al no haber elemento menor,
se intercambiaría con él mismo

3	**5**	**8**	**9**	14	23	17

Continuamos con el procedimiento, al no haber elemento menor,
se intercambiaría con él mismo



Por último, haremos este intercambio y ya queda ordenado el vector,
estando el último ya ordenado

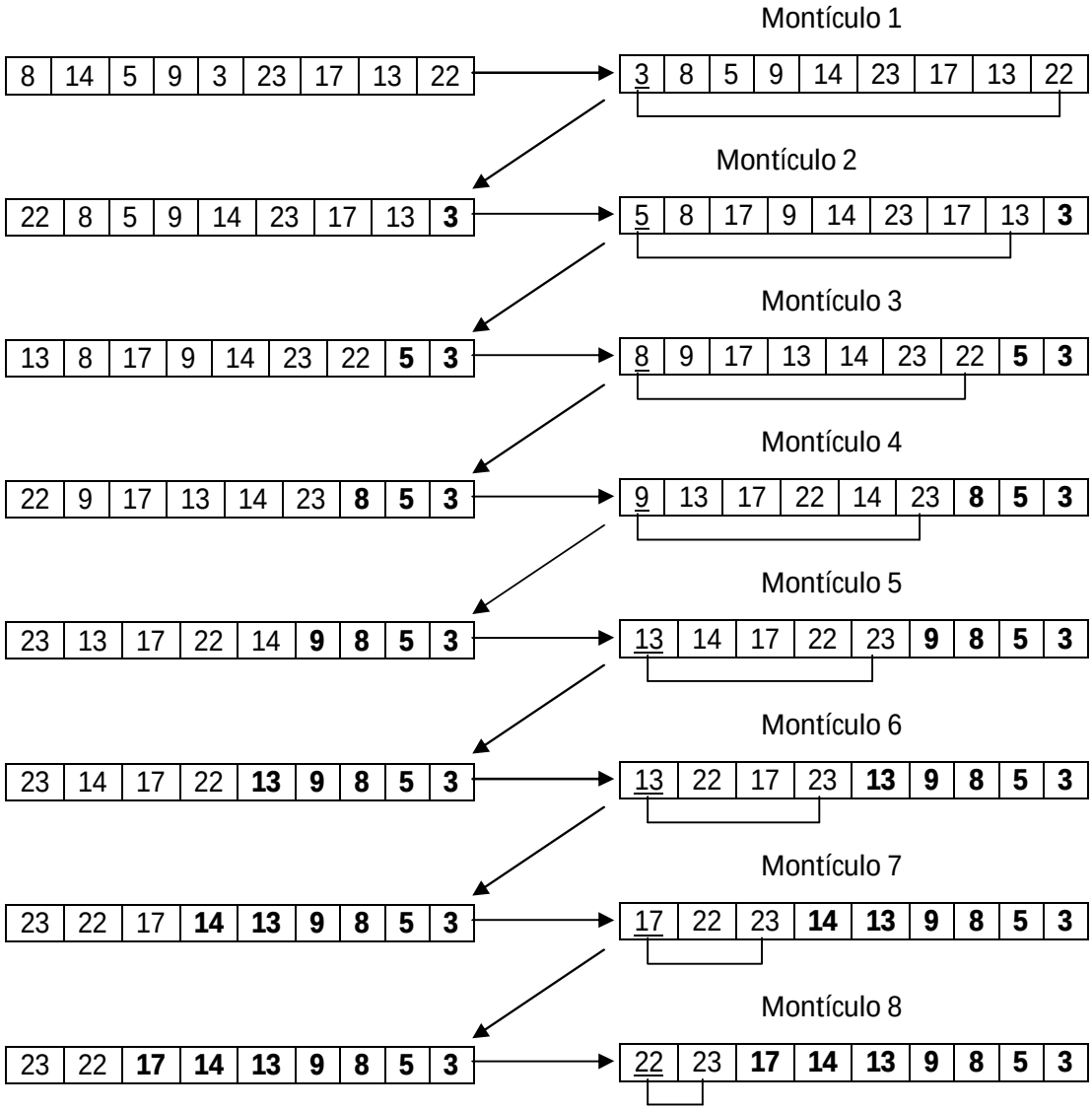


En cuanto a la **ordenación por montículo** decir que una vez organizado el arreglo como un montón, la clasificación (ordenación) del mismo se obtiene teniendo en cuenta que la cima, el elemento h_1, es el menor de todos los elementos del montón. Así para clasificar u ordenar un arreglo de *n* elementos se retira de la cima y se construye un montón con los demás $n - 1$ elementos. Su cima será de nuevo el menor de los que quedan, se retira y vuelve a construir un montón con los $n - 2$ restantes, y así sucesivamente. Una cuestión es donde almacenar las sucesivas cimas, para ello la solución adecuada consiste en intercambiar la cima, primer elemento del arreglo y construir un montón con los $n - 1$ elementos. Esta ampliación está sacada del libro de *estructuras de datos*, aunque no es literal del todo.

Para ello, veremos el algoritmo en pseudocódigo (visto ya en el tema 5 de nuestro resumen, de estructura de datos):

procedimiento ordenación por montículo $(T[1..n])$
{ T es la matriz que hay que ordenar }
crear-montículo (T);
para $i \leftarrow n$ bajando hasta 2 hacer
    intercambiar $T[1]$ y $T[i]$
    hundir $(T[1..i-1], 1)$

Igualmente, veremos un **ejemplo** sacado del libro de *estructuras de datos y algoritmos*, como sigue:



Por tanto, tras el último montículo ya quedaría el vector ordenado, como sigue :

23	**22**	**17**	**14**	**13**	**9**	**8**	**5**	**3**

Por último, se podría hacer una ordenación *in situ* del vector. Pero eso ya no lo haríamos, ya que nuestro código correspondería con el anterior ejemplo. En el libro donde hemos sacado estos ejemplos estará esta última ordenación, que dejaríamos sin verlo (aunque se puede ver en el tema 5 de nuestro resumen ejemplos similares).

Septiembre 2000-reserva (ejercicio 1)

Enunciado: En el algoritmo quicksort (ordenación rápida), ¿qué implicaciones tiene para la eficiencia el escoger como pivote la mediana exacta del vector?

Respuesta:

El **algoritmo quicksort** tiene un tiempo promedio de $O(n * \log(n))$. En la práctica es más rápido que la ordenación por montículo (heapsort) y la ordenación por fusión (mergesort) en un tiempo constante.

Al ser un algoritmo recursivo sería ideal poder dividir el caso en subcasos de tamaños similares de manera que cada elemento sea de profundidad $\log(n)$, es decir, poder equilibrar los subcasos. Sin embargo, aunque seleccionemos la mediana del vector en el peor caso (todos los elementos a ordenar son iguales) el orden será cuadrático.

Para obtener alguna ventaja utilizando como pivote la mediana tendremos que hacer algunas modificaciones al algoritmo original. Utilizaremos una nueva función pivote (recordemos que se llamaba *pivotebis*) que divida en 3 secciones el vector, de modo que dado un pivote *p* una parte conste de elementos menores que él, otra de elementos iguales y otra con los más grandes que *p*. Tras hacer esto se harían las llamadas recursivas correspondientes al algoritmo, una para el subvector de elementos menores que *p*, y otra para el de los mayores que *p*. Esta modificación consigue que, utilizando la **mediana** como pivote el orden del algoritmo sea $O(n * \log(n))$ incluso en el peor caso. Pero el coste que implica esta modificación hace que el tiempo constante crezca haciendo mejor el algoritmo de ordenación por montículo que el quicksort en todos los casos, lo que hace que la modificación no sea factible.

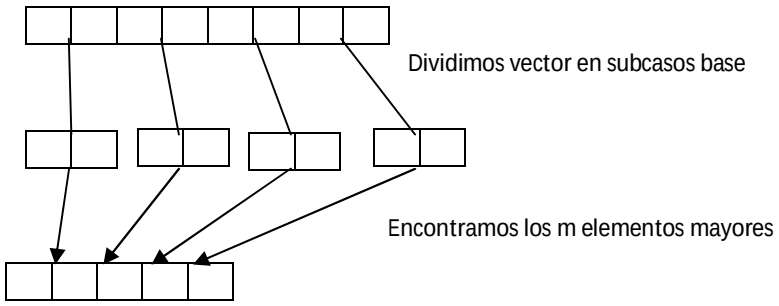
Febrero 2001-1ª (ejercicio 3)

Enunciado: Un vector T contiene n elementos. Se quieren encontrar los *m* mayores elementos del vector y se supone $n \gg m$ (*n* mucho mayor que *m*). Describe una forma eficiente de hacer esto sin ordenar el vector y calcula qué coste tiene.

Respuesta: Podemos hacer un enfoque similar a como hacemos en la **ordenación por fusión**, pero en este caso iremos generando un subvector con aquellos elementos mayores que *m*. Dividimos, por tanto, el vector original de forma equilibrada y progresivamente, hasta llegar a un tamaño base adecuado (lo suficientemente pequeño) para aplicar en él la búsqueda de todo elemento mayor que *m*. Cada vez que encontremos un elemento mayor que *m* lo almacenaremos en un vector resultado, que posteriormente se fusionará con los encontrados en las distintas llamadas. Igualmente podemos utilizar otro algoritmo de Ordenación pero sin ordenar el vector, si no almacenar en otro vector los índices en el orden correspondiente, por ejemplo.

NOTA DEL AUTOR: Se ha hecho una modificación al resultado de este ejercicio debido a que nos solicitan que encontremos los *m* elementos mayores. Por ello, al encontrar el elemento mayor lo iremos almacenando en el vector solución. En la solución dada del ejercicio (no sabemos si es oficial o no) está justamente al revés, por lo que hemos modificado esta parte en nuestra solución. Además, este ejercicio es un ejemplo del segundo tipo dado al inicio de los ejercicios.

Como añadido al ejercicio, **gráficamente** sería algo así:



Febrero 2002-2ª (ejercicio 1)

Enunciado: Poner un ejemplo de un vector de 10 elementos que suponga un ejemplo de caso peor para el algoritmo de ordenación Quicksort. Ordenar el vector mediante este algoritmo detallando cada uno de los pasos.

Respuesta: Recordemos que el caso peor de este algoritmo es aquél en el que tras poner el **pivote** en su sitio correcto es hay que ordenarlo salvo un elemento, que sería el ya ordenado. El caso peor es aquel en el que el vector está ordenado de modo creciente, como sigue:

1	2	3	4	5	6	7	8	9	10

Recordemos los procedimientos de este algoritmo:

procedmiento pivote ($T[i..j]$, var l)
{ Permuta los elementos de la matriz $T[i..j]$ y proporciona un valor l, tal que, al final, $1 \leq l \leq j$; $T[k] \leq p$ para todo $i \leq k < l$, $T[l] = p$, y $T[k] > p$ para todo $1 < k \leq j$, en donde p es el valor inicial de $T[i]$ }
$p \leftarrow T[i]$;
$k \leftarrow i$; $l \leftarrow j + 1$
repetir $k \leftarrow k + 1$ hasta que $T[k] > p$ o $k \geq j$
repetir $l \leftarrow l - 1$ hasta que $T[l] \leq p$
mientras $k < l$ hacer
intercambiar $T[k]$ y $T[l]$
repetir $k \leftarrow k + 1$ hasta que $T[k] > p$
repetir $l \leftarrow l - 1$ hasta que $T[l] \leq p$
intercambiar $T[i]$ y $T[l]$

El algoritmo propiamente dicho de ordenación rápida (*quicksort*) es:

procedmiento quicksort ($T[i..j]$)
{ Ordena la submatriz $T[i..j]$ por orden no decreciente }
si $j - i$ es suficientemente pequeño entonces
insertar ($T[i..j]$)
si no
pivote ($T[i..j]$, l)
quicksort ($T[i..l - 1]$)
quicksort ($T[l + 1..j]$)

Seguiremos los pasos como sigue:

1er paso: Hará lo siguiente:

Tenemos que el vector inicial es $T[1..10] = \{1,2,3,4,5,6,7,8,9,10\}$, por lo que la llamada inicial a pivote es pivote ($T[1..10]$, L).

Tendremos, por tanto, que:

$i = 1, j = 10, p = 1, k = 1, L = 11$

Con estos valores seguiremos este procedimiento:

- El puntero k, se mueve hasta el valor de 2, debido a que $T[2] > p$.
- L llega hasta 1, por ser $T[1] \leq p$.
- Al superar el puntero k a L no tendremos que hacer ningún intercambio, excepto en la última instrucción intercambiar $T[i]$ por $T[L]$, como L apunta a 1, e i es 1, $T[L]$ queda igual.

2º paso: A continuación haríamos una llamada a quicksort ($T[1..0]$), siendo la llamada a un caso de tamaño cero. Además, haríamos otra llamada a quicksort ($T[2..10]$), siendo éste el resto del vector.

Ignoramos la primera llamada al ser de un tamaño 0, mientras que la otra tendría un efecto similar al del paso anterior, siguiendo este procedimiento:

- En este caso en el procedimiento pivote L descendería hasta llegar a $T[2]$ y k quedaría en el tercer elemento.
- Tendremos exactamente lo mismo que antes con los punteros, por lo que no se provoca cambio ninguno.

3er paso: De nuevo volvemos a hacer una tercera llamada a quicksort, una de ellas para un caso de tamaño 0, que sería quicksort ($T[3..2]$) y otra para el resto del vector ($T[3..10]$).

El proceso se repetirá hasta llegar al último elemento, produciéndose tantas llamadas como elementos posea el vector.

NOTA DEL AUTOR: Nos fijamos que están mal distribuidos los vectores que hay que ordenar, por ello sería el algoritmo peor, como hemos visto en la teoría de este capítulo. Además, esta solución está sacada de la respuesta dada en el examen, aunque desconocemos si es oficial (u "oficiosa").

Febrero 2002-2ª (ejercicio 1)

Enunciado: Escribe la secuencia completa de pasos para ordenar por fusión el vector [3,2,7,5,9,3,4,8].

Respuesta: Esta solución está hecha por el autor, aunque es simplemente seguir el procedimiento dado en la teoría y hacer el dibujo respectivo a cada paso. Por tanto, veremos los pasos y los explicaremos:

Tenemos el vector:

3	2	7	5	9	3	4	8

1er paso: Dividimos en dos partes el vector, que nos damos cuenta que tiene longitud par. Por tanto, tendremos:

3	2	7	5

9	3	4	8

De momento, no hemos llegado al caso suficientemente simple, por lo que seguiremos dividiendo hasta encontrarlo.

2º paso: Dividiremos la parte izquierda en dos partes, de nuevo:

3	2

7	5

3er paso: Al ser suficientemente simples (tienen dos elementos) ambas partes podremos ordenarlas de modo creciente, quedando como sigue:

2	3

5	7

4º paso: Fusionaremos estas dos partes y tendremos:

2	3	5	7

En este caso, ya está ordenada la parte izquierda, por lo que a continuación veríamos la parte derecha del vector.

5º paso: Seguiríamos con el mismo procedimiento que antes y, por tanto, dividimos la parte izquierda:

9	3

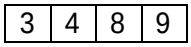
4	8

6º paso: Los ordenamos quedándonos lo siguiente:

3	9

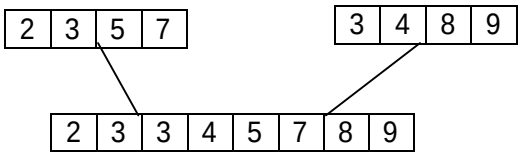
4	8

7º paso: Fusionaremos ambas partes:



A diferencia de la otra fusión, en este caso tendremos que hacer una comparación de ambos punteros, que recordemos de la teoría eran la *i* y *j* para luego fusionarlos en el otro vector.

8º paso: Por último, fusionaremos ambas mitades ordenadas:



Estando ya ordenado el vector correctamente.

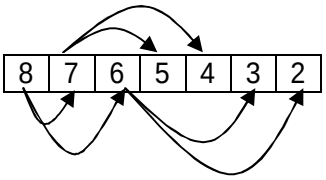
Hemos omitido algunos pasos, en los que se compararían ambos punteros y, por tanto, se vería como se incrementa el puntero en ambas mitades, así como en el vector solución, aunque eso ya lo hemos estudiado en la teoría de la asignatura y en ejercicios anteriores. Con estos pasos, creo que vale para solucionar este ejercicio.

Septiembre 2002 (ejercicio 1)

Enunciado: En el algoritmo de ordenación por montículo (heapsort). ¿Cuáles son las mejores y peores disposiciones iniciales de los elementos que hay que ordenar en cuanto al tiempo de ejecución? Razonar la respuesta y poner ejemplos.

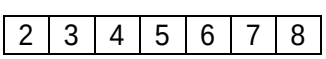
Respuesta: Esta solución está dada igualmente que la anterior por el autor.

El caso mejor del algoritmo de ordenación por montículo (de máximos) es aquél en el que todos los elementos del vector cumple las **propiedades del montículo**, que recordemos eran $T[i] \geq T[2 * i]$ y $T[i] \geq T[2 * i + 1]$. Es decir, ordenado de modo *no* natural (por orden decreciente). Para verlo más claro, pondremos un ejemplo:



En este caso, por tanto, no haremos ningún intercambio, ya que estarían ordenados siguiendo las propiedades del montículo. Por tanto, en el **caso mejor** el coste es $\boldsymbol{\theta(n)}$, en el que sólo recorre los elementos del montículo y lo compara sin intercambiarlo.

El peor caso será justamente lo contrario, es decir, aquél en el que los elementos estén ordenados de modo creciente, como puede ser:



Un ejemplo de ordenación por montículo lo hemos visto previamente, al igual que el pseudocódigo que emplearemos para ordenarlo. Por ello, no lo ordenaremos, dejándolo como ejercicio extra (aunque fácil, una vez visto el ejercicio anterior).
En este caso, en el caso peor el coste es $\boldsymbol{\theta(n * log(n))}$.

Septiembre 2002 (ejercicio 3)

Enunciado: De los algoritmos de ordenación que has estudiado. ¿Cuál es el más eficiente en términos de coste asintótico temporal en el caso peor? Razonar la respuesta.

Respuesta: Esta solución es mía personal. Hemos estudiado por el momento estos algoritmos de ordenación (aunque hay más), que es el algoritmo de ordenación por fusión (*mergesort*), ordenación rápida (*quicksort*), ordenación por montículo (*heapsort*), ordenación por inserción. Estos dos primeros algoritmos tienen coste en el caso peor cuadrático (recordemos que era por estar mal distribuida las partes), aunque el tercer algoritmo tiene coste $O(n * \log(n))$ en el peor caso. El último de ellos, tendrá coste cuadrático. Diríamos, por tanto, aunque no es seguro que esté correcta la solución particular dada.

Septiembre 2002-reserva (ejercicio 2)

Enunciado: Cuando un cartero reparte cartas en los buzones de un edificio, lo hace en un tiempo lineal, suponiendo que cada carta se asigna a un buzón en tiempo constante. Explicar, a partir del comportamiento del cartero, cómo pueden ordenarse n ejemplares de números entre 1 y m, siendo m una cantidad conocida y fija, en un tiempo lineal. ¿Por qué decimos entonces que el algoritmo quicksort es muy eficiente, si tiene un coste promedio de $O(n * \log(n))$?

Respuesta: Este ejercicio se parece mucho al 1 de Febrero de 2003-2ª semana, que más adelante veremos, sólo que pondremos este enunciado por seguir el orden cronológico de ejercicios. En él, se nos pide que ordenemos un vector de unos elementos, al igual que pasa con este ejercicio. Para ello, tendremos que seguir estos pasos:

1. El vector está inicializado a 0 $\left(O(n)\right)$.
2. Recorremos la lista de valores a ordenar $\left(O(n)\right)$.
   a. Por cada valor i extraído de la lista, incrementamos el valor de la posición y del vector.
3. Recorremos el vector generado mostrando los valores generados en el paso anterior (2) y omitiendo aquellas posiciones del vector que contengan un 0.

Además de lo dicho en este ejercicio, la importancia de escribirlo por separado es que la última pregunta es interesante, debido a que nos preguntan sobre el algoritmo quicksort. Esta pregunta realmente no la acabo de comprender, ya que el algoritmo quicksort (ordenación rápida) es igual de eficiente que puede ser el mergesort (ordenación por fusión), siendo ambos costes iguales en el caso mejor. Por ello, no entiendo la pregunta a que se refiere, e incluso según la teoría de Brassard nos comentan que a nivel de espacio requiere más que la ordenación por fusión, aunque ello requiere rebajar la constante multiplicativa.

Diciembre 2002 (ejercicio 3)

Enunciado: Ordenar ascendentemente mediante el algoritmo de quicksort el vector $V \rightarrow$ [2,3,8,1,9,4,2,2,6,5,4,3,7,4] detallando todos los pasos.

Respuesta: Tenemos el siguiente vector:

2	3	8	1	9	4	2	2	6	5	4	3	7	4

Tomamos el pivote como primer elemento, para luego buscar a la derecha el primer elemento mayor (subrayado) y a la izquierda el menor o igual que el pivote (superrayado) e

intercambiamos. Seguimos haciendo esto hasta que los punteros se crucen. Tendremos, por tanto:

2	3	8	1	9	4	2	2	6	5	4	3	7	4

Pondremos los pasos que requieran de intercambio, ya que así haremos el ejercicio algo más corto, por no hacer pasos innecesarios. Por tanto, tendremos:

2	3	8	1	9	4	2	2	6	5	4	3	7	4
2	2	8	1	9	4	2	3	6	5	4	3	7	4
2	2	2	1	9	4	8	3	6	5	4	3	7	4

Tras este último intercambio decir que ya ambos punteros se han cruzado, por lo que se puede intercambiar con el pivote previamente puesto y así ordenar las subpartes que lo separa el pivote. Por ello, tendremos tras este primer paso lo siguiente:

1	2	2	**2**	9	4	8	3	6	5	4	3	7	4

Como anotación de este ejercicio, decir que misteriosamente y curiosamente el realizado con solución al final tras esos intercambios no se queda ordenado el vector. Por ello, lo haremos de modo correcto en nuestra solución.

Partimos, por tanto, por la parte izquierda, aunque si escogiéramos el pivote como el elemento número 1, al momento veríamos que esta parte está ya ordenada. Por tanto, seguiremos por la parte derecha, quedando:

9	4	8	3	6	5	4	3	7	4

En este caso el pivote seria el elemento mayor, por lo que lo podríamos poner a la derecha, intercambiándolo con el 4 superrayado, estando ya este último elemento ordenado. Tendremos:

4	4	8	3	6	5	4	3	7	**9**

De nuevo, ordenaremos la parte izquierda, salvo el valor 9:

4	4	8	3	6	5	4	3	7

Como ya controlamos el procedimiento, realizaremos los intercambios sucesivos como sigue:

4	3	8	3	6	5	4	4	7
4	3	3	8	6	5	4	4	7
3	3	**4**	8	6	5	4	4	7

De nuevo, la parte izquierda ya está ordenada, así que continuamos por la derecha:

8	6	5	4	4	7

No podremos intercambiar ningún elemento, así que nos quedará:

7	6	5	4	4	**8**

De nuevo, la parte izquierda es:

7	6	5	4	4

No podremos intercambiar con ningún elemento mayor, como antes, así que tendremos:

4	6	5	4	**7**

Otra vez, la parte izquierda será:

4	6	5	4

4	4	5	6

Observamos, sin extendernos más, que ya lo que queda está ordenado, por lo que el vector completo ordenado es:

1	2	2	2	3	3	4	4	4	5	6	7	8	9

NOTA DEL AUTOR: Siento si este ejercicio queda algo liado, pero no sé hacerlo de otra manera, en la que quede más claro. Se me ocurre que puede ser la mejor.

Febrero 2003-1ª (ejercicio 3)

Enunciado: Se desea implementar una función para *desencriptar* un mensaje numérico. La función *desencriptar* recibe tres enteros: el mensaje cifrado *c*, la clave privada *s* y la clave publica *z*; y devuelve el mensaje original *a*. El mensaje original se recompone con la fórmula:

$$a \coloneqq c^s \bmod z$$

Sabiendo que no se dispone del operador de potencia, implementar la función utilizando el esquema de divide y vencerás.

Respuesta:

Como hemos dicho en la teoría, este apartado no lo vimos, por lo que es el momento adecuado para ello. Tendremos dos modos de hacerlo:

a) El primero de ellos es en el que nos dan en el enunciado la función de *desencriptar*, que toma 3 parámetros, lo que sería:

```
fun desencriptar (c, s, z: entero)
    devolver expoDV (c, s) mod z
ffun
```

Para calcularlo usando el esquema de divide y vencerás usaremos el algoritmo *expoDV*, visto anteriormente:

```
funcion expoDV (a, n: entero)
    si n = 1 entonces devolver a
    si n es par entonces devolver [expoDV(a, n/2)]^2
    devolver a * expoDV(a, n − 1)
```

b) Otra manera de hacerlo es empleando para ello *expomod*, que es una modificación del dado anterior *expoiter*:

```
funcion expomod (c, s, z: entero)
    { Calcula c^s mod z }
    i ← s; r ← 1; x ← c
    mientras i > 0 hacer
        si i es impar entonces r ← c mod z
        x ← x^2 mod z
        i ← i ÷ 2
    devolver r
```

NOTA DEL AUTOR: Entiendo que en la solución aportada de este ejercicio se nos dan dos partes, pero no me acaba de quedar claro realmente cual es la diferencia entre dos. Por ello, asumo que son dos maneras distintas de hacerlo, una de modo recursivo (*expoDV*) y otra de modo iterativo (*expomod*).

Febrero 2003-2ª (ejercicio 1) (igual a ejercicio 2 de Septiembre 2006-reserva y a ejercicio 2 de Diciembre 06)

Enunciado: Explica cómo pueden ordenarse n valores enteros positivos en tiempo lineal, sabiendo que el rango de valores es limitado. Explica las ventajas e inconvenientes de este método.

Respuesta: Suponiendo que el rango de los valores a ordenar es **limitado**, es decir, sabemos que, por ejemplo, el rango de números a ordenar va de 0 a 2000, podemos generar un vector de 2000 elementos del tipo entero que almacenara el número de ocurrencias de cada valor de la lista a ordenar en la posición del vector correspondiente. Seguiremos estos pasos:

1. El vector está inicializado a 0 $(O(n))$.
2. Recorremos la lista de valores a ordenar $(O(n))$.
    a. Por cada valor i extraído de la lista, incrementamos el valor de la posición y del vector.
3. Recorremos el vector generado mostrando los valores generados en el paso anterior (2) y omitiendo aquellas posiciones del vector que contengan un 0.

Septiembre 2003 (ejercicio 2)

Enunciado: Una matriz T contiene n elementos. Se pide encontrar los m elementos más pequeños de T (con $m \ll n$). Explicar cómo hacer esto de la manera más eficiente.

Respuesta:

Este ejercicio se ha sacado del libro *Estructuras de datos y métodos algorítmicos. Ejercicios resueltos* de Narciso Martí, Y. ortega, J.A. Verdejo, es el número 11.11, además de parecerse bastante al ejercicio 3 de Febrero 2001-1ª semana. Pasamos a verlo:

Un procedimiento parcial de ordenación por selección nos da los m elementos más pequeños con coste $O(m * n)$. Una ordenación eficiente, sin embargo, lo haría en $O(n * \log(n))$. Si $m \approx n$, el coste vendría a ser cuadrático, lo cual nos haría desechar el procedimiento de selección, sin embargo, el orden $O(m * n)$ se puede considerar lineal y, en este caso, el algoritmo de selección puede ser más eficiente.

Tendríamos estos métodos:

- El **primer método** es: Ordenar V[1..n] y coger los m primeros elementos. Tiene coste $\theta(n^2)$.
- El **segundo método** es: Utilizar un algoritmo de coste lineal para la selección. Tiene coste $\theta(m * n)$.
- El **tercer método** es:

    1. Mediante una llamada a selección2 (V, 1, n, m, v) devuelve el m-ésimo elemento menor de V.
    2. Llamamos a partición (V, 1, n, v, i, j), en la que se particionan los m-ésimos menores elementos para guardarlos en V.
    3. Devolvemos V[1..m].

    En este método, el coste es $\theta(n)$.

Evidentemente, tendremos que el tercer método, como vimos en el ejercicio anterior es el adecuado y el más eficiente para nuestro problema.

NOTA DEL AUTOR: Creo que no ha quedado claro este ejercicio al ser una copia literal del ejercicio, aunque se ha intentado. Sobre todo lo interesante es ver los métodos y estudiarlos.

Septiembre 2003-reserva (ejercicio 1)

Enunciado: Ordenar completamente el vector en orden no decreciente por el método quicksort indicando claramente cuál es el contenido del vector en cada paso.

3	4	5	1	3	7	6	2

Respuesta:

Este ejercicio está resuelto por el autor completamente. Ya hemos visto un ejemplo similar a éste, por lo que lo que haremos será ordenar el vector indicando que se intercambia. Lo curioso de este ejercicio es que piden orden no decreciente, es decir, orden creciente (es rizar el rizo). Por tanto, veremos los pasos:

3	4	5	1	3	7	6	2

3	4	5	1	3	7	6	2

3	*2*	5	1	3	7	6	*4*

3	2	*3*	1	*5*	7	6	4

1	2	3	**3**	5	7	6	4

1	2	3	3	**5**	7	6	4

1	2	3	3	**5**	*4*	6	*7*

1	2	3	3	4	**5**	6	7

El último intercambio ya sería el que ordenara completamente el vector, por lo que quedaría así:

1	2	3	3	4	5	6	7

Septiembre 2003-reserva (ejercicio 3)

Enunciado: ¿Se puede aplicar el procedimiento de búsqueda binaria sobre árboles con estructura de montículo? Razonar la respuesta.

Respuesta: Esta solución es del autor personalmente. La respuesta sería que **no**, porque el vector al tener estructura de montículo no está ordenado de modo creciente (aunque el montículo sea de mínimos), siendo una de las condiciones para realizar la búsqueda binaria. Al ser montículo puede haber hijos que sean iguales que los padres.

Un **ejemplo** será:

[3,4,3, ...]

No ordenado

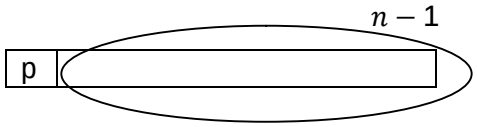
Diciembre 2003 (ejercicio 3)

Enunciado: ¿Cuáles son los casos mejor y peor para el algoritmo de ordenación rápida (quicksort)? ¿Cuál es el orden de complejidad en cada uno de ellos? Razona tu respuesta.

Respuesta: Este ejercicio lo hemos visto previamente en la teoría, por lo que lo recordaremos a continuación:

Vemos estos casos de colocación del pivote:

- **Cuando el pivote p queda en un extremo** (al inicio o al final, da igual): Tendremos una versión no equilibrada de ordenación rápida, en la que el tamaño del problema se reduce en una mitad. La situación tras colocar los elementos menores y mayores es:



La ecuación de recurrencia, por tanto, sería:

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

Recolocación del pivote

Las distintas variables al igual que hemos visto previamente es:

a: Número de llamadas recursivas = 1
b: Reducción del problema en cada llamada = 1
$\boldsymbol{c * n^k}$: Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 1$, al ser el tiempo extra lineal.

Recordemos que la resolución para la reducción de la **recurrencia por sustracción** es la siguiente:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta(a^{n \; div \; b}) & \text{si } a > 1 \end{cases}$$

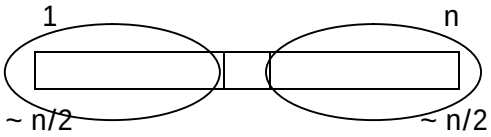
Por tanto, vemos que $a = 1$, por lo que estaremos en el segundo caso. Pasamos a resolverlo, siendo el tiempo $t(n) \in \theta(n^{k+1}) = \boldsymbol{\theta(n^2)}$.

El algoritmo se comporta muy mal en este caso, siendo éste el **caso peor**. Veremos un ejemplo de este caso peor, si *T* ya está ordenado antes de la llamada a quicksort obtenemos $l = i$ en todas las ocasiones, lo cual implica una llamada recursiva a un caso de tamaño 1 y otra a un caso cuyo tamaño se reduce en una unidad.

Este caso podremos compararlo con el peor de la ordenación por fusión (donde estaban descompensadas las particiones), en la que recordemos tenía coste cuadrático.

- Por otra parte, **si los elementos de la matriz que hay que ordenar se encuentran inicialmente en orden aleatorio**, tendremos que los subejemplares para ordenar estarán suficientemente bien equilibrados.

  En el caso peor, tendremos:



  El pivote está ya ordenado.

  Tendremos esta **recurrencia por división**:

$$t(n) = 2 * t(n/2) + O(n)$$

Recolocación del pivote

  De nuevo, los valores de las variables son:

  a: Número de llamadas recursivas = 2
  b: Reducción del problema en cada llamada = 2
  $\boldsymbol{c * n^k}$: Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 1$, al ser el tiempo extra lineal.

  Resolvemos la ecuación $a = b^k$, siendo éste $2 = 2^1$, que es el segundo caso:

$$(n) \in \theta\left(n^k * log(n)\right) = \boldsymbol{\theta(n * log(n))}.$$

  Este caso correspondería con el **mejor caso** de esta ordenación.

Febrero 2004-1ª (ejercicio 1)

Enunciado: Programar una función "potencia (n, m)" que halle n^m -se supone que no existe la operación potencia y que el coste de una operación de multiplicación es $O(1)$- mediante Divide y Vencerás con coste $\theta(\log(n))$.

Respuesta: La solución dada está hecho por un alumno, que es el autor. La función que nos piden programar no es ni más ni menos que la *expoDV* que la tenemos en el libro de Brassard, página 276 y en el resumen de la asignatura. Insistimos en la importancia de comprender bien estos códigos, ya que este ejercicio se parece mucho a la cuestión 3 de Febrero de 2008, que veremos más adelante con detenimiento. El algoritmo será, por tanto:

funcion potencia (n, m) dev entero
  si $m = 1$ entonces devolver n
  si m es par entonces devolver $[potencia(n, m/2)]^2$
  devolver $a * potencia(n, m - 1)$

Una curiosidad es que en la teoría, que es de donde hemos sacado este algoritmo no ponemos que devuelva ningún valor, no obstante, sí que lo devuelve, como se puede ver. Esto es otra pequeña característica que habría que tener en cuenta en los códigos, pero que a veces se salta.

Vimos en la teoría que la ecuación de recurrencia seria:

$$\underbrace{N(\lfloor n/2 \rfloor) + 1}_{n\ par} \leq N(n) \leq \underbrace{N(\lfloor n/2 \rfloor) + 2}_{n\ impar}$$

Las distintas variables serian:

a: Número de llamadas recursivas = 1, que seria 1 si es impar o par.
b: Reducción del problema en cada llamada = 2
$\boldsymbol{c * n^k}$: Coste de las operaciones extras a las llamadas recursivas. Al ser una constante la operación extra, tendremos que $k = 0$.

La resolución de la **recursividad por división** es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Por tanto, deducimos que el coste al sustituir los valores en $a = b^k$ es $1 = 2^0$, siendo el caso el segundo. Por tanto, de nuevo el coste $t(n) \in \theta\left(n^0 * log(n)\right) = \theta\left(log(n)\right)$, tal y como nos piden en el enunciado. Nos fijamos, además, que las operaciones son elementales, por ello, todas las multiplicaciones tienen coste constante, si no sería otro coste totalmente distinto, para eso habría que remontarse a la teoría.

Septiembre 2005 (ejercicio 1)

Enunciado: Suponga que N personas numeradas de 1 a N deben elegir por votación a una entre ellas. Sea V un vector en el que la componente $V[i]$ contiene el número de candidato que ha elegido el votante i. ¿Qué algoritmo utilizarías para determinar si una persona ha obtenido más de la mitad de los votos?

Respuesta: Podría utilizarse una estrategia similar a la que emplea para la ordenación el algoritmo de la *casilla* (página 80 del libro de Brassard), pero trasladado al conteo de elementos. La siguiente función almacena en el vector *votos* el número de votos que va sumando cada candidato votado. Devuelve un valor TRUE si hay alguno con mayoría indicando de qué candidato se trata:

```
funcion contar_votos (V[1..n]) dev (boolean, elegido)
    votos[1..n] = 0;
    para i = 1 hasta N hacer
        votos [V[i]] = votos [V[i]] + 1;
        si votos [V[i]] > N/2 entonces devolver (TRUE,  V[i]) fsi
    fpara
    devolver (FALSE, 0);
```

Observamos que tendremos dos vectores, uno donde almacenan los votos (V) y otro donde los cuentan (votos). Este ejercicio es parecido a aquellos de los "bombones pesados" y similar, por lo que lo veremos en la sección de problemas resueltos más ampliados. Aun así, como es habitual este algoritmo sirve para estos ejercicios.

Septiembre 2006 (ejercicio 3)

Enunciado: Dibuja como evolucionaría el siguiente vector al ordenarlo mediante el algoritmo de ordenación rápida (quicksort). Indica únicamente cada una de las modificaciones que sufriría el vector.

6	5	1	2	3	4	7	8	9

Respuesta:

Hemos visto numerosos ejercicios de este tipo, por tanto, omitiremos dar más detalles y literalmente copiaremos la solución dada. Nuestro criterio es tomar como **pivote** el *primer elemento*.

6	5	1	2	3	4	7	8	9

4	5	1	2	3	6	7	8	9

4	3	1	2	5	6	7	8	9

2	3	1	4	5	6	7	8	9

2	1	3	4	5	6	7	8	9

1	2	3	4	5	6	7	8	9

Ahora nuestro criterio es tomar como **pivote** el elemento que ocupa la *posición central*, que en este caso el primer elemento es el elemento que ocupa la posición quinta, que es el 3:

6	5	1	2	**3**	4	7	8	9

2	5	1	6	3	4	7	8	9

2	1	5	6	3	4	7	8	9

2	1	3	6	5	4	7	8	9

1	2	3	6	5	4	7	8	9

1	2	3	4	5	6	7	8	9

Observamos que hace el mismo número de pasos con ambos criterios.

Febrero 2007-1ª (ejercicio 2) (parecido a ejercicio 3 de Septiembre 2008)

Enunciado: Escribir en pseudocódigo una función recursiva de búsqueda binaria en un vector, suponiendo que en algunos casos el elemento no está en el vector. Demuestra el coste que tiene la función implementada.

Respuesta: Los códigos que se piden en la primera pregunta son los siguientes, recordando que el primero indicaba si el elemento estaba dentro del vector y el segundo es el propio de la búsqueda binaria:

```
funcion busquedabin (T[1..n], x)
    si n = 0 ó x > T[n] entonces devolver n + 1
    si no                          { Elemento dentro del vector }
        devolver binrec (T[1..n], x)


funcion binrec (T[i..j], x)
    { Búsqueda binaria de x en la submatriz T[i..j] con la seguridad de que
    T[i − 1] < x ≤ T[j] }
    si i = j entonces devolver i
    k ← (i + j) ÷ 2
    si x ≤ T[k] entonces devolver binrec (T[i..k], x)     { Mitad izquierda}
    si no devolver binrec (T[k + 1..j], x)                { Mitad derecha }
```

En cuanto a la segunda pregunta, no haría falta más que sustituir los valores en la ecuación de recurrencia y ver que de nuevo (es bastante importante) el coste del algoritmo sería $\boldsymbol{\theta(\log(n))}$.

Este ejercicio se parece mucho al 3 de Septiembre de 2008, en el que piden que al no existir el número devuelva -1, habría que sustituir en el algoritmo busquedabin $n + 1$ por -1, que es simplemente un formalismo. Por otro lado, se resuelve igual a como viene en el libro que es de donde se han sacado estos algoritmos anteriores.

Febrero 2007-1ª (ejercicio 3)

Enunciado: Dado un vector $C[1..n]$ de números enteros distintos y un número entero S, se pide plantear un programa de complejidad $\theta(n * \log(n))$ que determine si existen o no dos elementos de C tales que su suma sea exactamente S. En caso de utilizar algoritmos conocidos no es necesario codificarlos.

Respuesta: Hemos resuelto este problema tomando como base el ejercicio 11.16 del libro de Martí. Por tanto, podremos resolver el problema siguiendo la siguiente idea:

1. Ordenar el vector C.
2. Para cada elemento x de C buscar, utilizando búsqueda binaria, el entero $S - x$. Si alguna de las búsquedas tiene éxito, habremos encontrado dos elementos de C que sumen S.

El coste del primer paso está en $\theta(n * \log(n))$ y el del segundo igualmente en $\theta(n * \log(n))$, ya que en el caso peor, se hacen n búsquedas binarias de coste $\theta(n * \log(n))$ para cada una de ellas. Por tanto, el **coste total** del algoritmo está en $\boldsymbol{\theta(n * \log(n))}$.

Septiembre 2007 (ejercicio 2)

Enunciado: Supongamos que un polinomio se representa por un vector $v[0..n]$ de longitud $n + 1$ donde el valor $v[i]$ es el coeficiente de grado i. describir claramente los cálculos y la descomposición algorítmica necesaria para plantear un algoritmo, de orden mejor que cuadrático, que multiplique polinomios P(x) y Q(x) mediante divide y vencerás. NOTA: La solución típica trivial de orden cuadrático puntúa 0 puntos.

Respuesta: Es relativamente simple dar con la solución de orden cuadrático. En este caso, se descompone en mitades $P(x) = Ax^{n/2} + B$ y $Q(x) = Cx^{n/2} + D$ con *A*, *B*, *C*, *D* polinomios de grado n/2 sacando factor común $x^{n/2}$ como se detalla en las expresiones. De esta forma se ve claramente que $P * Q$ es $\left(Ax^{n/2} + B\right) * \left(Cx^{n/2} + D\right) = ACx^n + (AD + BC)x^{n/2} + BD$ lo que la solución conlleva 4 multiplicaciones de grado n/2. El coste sería en este caso cuadrático.

Sin embargo, hay una manera de organizar las operaciones mediante la cual, no es necesario calculas $AD + BC$ mediante 2 productos, sino solo con uno, aprovechando que ya tenemos realizados los productos BD y AC. En este último caso basta con observar que $(A + B) * (C + D) = AC + BC + AD + BD$ y que $BC + AD = (A + B) * (C + D) - AC - BD$ con lo que es posible realizar el cálculo con 3 productos en lugar de 4, ya que el coste de las sumas, si consideramos las multiplicaciones como lineales, sería constante. De manera que $(A + B) * (C + D)$ sería uno de los productos, y AC y BD los otros dos.

Este problema es similar al problema de la multiplicación de enteros muy grandes del texto base (ver resumen de la asignatura o libro de Brassard).

Septiembre 2007 (ejercicio 3)

Enunciado: Con respecto al algoritmo de ordenación por fusión (mergesort).

a) Escribe el algoritmo.
b) Dibuja la secuencia de llamadas del algoritmo y la evolución del siguiente vector al ordenarlo. Para ello ten en cuenta que hay que considerar que el problema es suficientemente pequeño cuando el array es de tamaño 2.

(2,0,2,1,9,6,2,3,5,8)

Respuesta:

a) El algoritmo lo hemos visto en la teoría, aunque lo veremos de nuevo poniendo el pseudocódigo siguiente:

   Emplearemos como centinela (una especie de posición auxiliar, para evitar realizar cálculos extras) la última posición en las matrices U y V.

```
procedimiento fusionar (U[1..m + 1], V[1..n + 1], T[1..m + n])
    { Fusiona las matrices ordenadas U[1..m] y V[1..n] almacenándolas en
    T[1..m + n], U[m + 1] y V[n + 1] se utilizan como centinelas }
    i, j ← 1;
    U[m + 1], V[n + 1] ← ∞
    para k ← 1 hasta m + n hacer
        si U[i] < V[j] entonces
            T[k] ← U[i]; i ← i + 1
        si no
            T[k] ← V[j]; j ← j + 1
```

El algoritmo de **ordenación por fusión** es como sigue, en donde utilizamos la ordenación por inserción (insertar) como subalgoritmo básico, que también añadiremos a continuación (tomándolo del tema 2).

```
procedimiento ordenarporfusion (T[1..n])
    si n es suficientemente pequeño entonces
        insertar (T)
    si no
        matriz U[1..1 + ⌊n/2⌋], V[1..1 + ⌊n/2⌋]
        U[1..⌊n/2⌋] ← T[1..⌊n/2⌋]
        V[1..⌊n/2⌋] ← T[1 + ⌊n/2⌋..n]
        ordenarporfusion (U[1..⌊n/2⌋])
        ordenarporfusion (V[1..⌊n/2⌋])
        fusionar (U, V, T)
```

Usaremos la función de *fusionar* anterior. Vamos a ver la función de *insertar* tal y como hemos comentado previamente:

```
procedimiento insertar T([1..n])
    para i ← 2 hasta n hacer
        x ← T[i]; j ← i − 1;
        mientras j > 0 y x < T[j] hacer
            T[j + 1] ← T[j];
            j ← j − 1;
        T[j + 1] ← x
```

b) Para variar, veremos una solución alternativa (o no visto previamente) de esta ordenación, como sigue:

OrdFusion (2, 0, 2, 1, 9, 6, 2, 3, 5, 8)
- OrdFusion (2, 0, 2, 1, 9)
  - OrdFusion (2, 0)
    - Insertar (2, 0) → (0, 2)
  - OrdFusion (2, 1, 9)
    - OrdFusion (2)
      - Insertar (2) → (2)
    - OrdFusion (1, 9)
      - Insertar (1, 9) → (1, 9)
    - Fusionar ((2), (1,9)) → (1, 2, 9)
  - Fusionar ((0,2), (1,2,9)) → (0, 1, 2, 2, 9)
- OrdFusion (6, 2, 3, 5, 8)
  - OrdFusion (6, 2)
    - Insertar (6, 2) → (2, 6)
  - OrdFusion (3, 5, 8)
    - OrdFusion (3)
      - Insertar (3) → (3)
    - OrdFusion (5, 8)
      - Insertar (5, 8) → (5, 8)
    - Fusionar ((3), (5,8)) → (3, 5, 8)
  - Fusionar ((2,6), (3,5,8)) → (2, 3, 5, 6, 8)
- Fusionar ((0,1,2,2,9), (2,3,5,6,8)) →(0, 1, 2, 2, 2, 3, 5, 6, 8, 9)

Septiembre 2007-reserva (ejercicio 2)

Enunciado: Dos amigos juegan a un sencillo juego de adivinación: uno de ellos piensa un número natural positivo y el otro debe adivinarlo solamente preguntado si es menor o igual que otros números. ¿Qué esquema utilizaría para adivinar el número en tiempo logarítmico? Diseñe el algoritmo y escríbalo en pseudocódigo.

Respuesta: No hay solución oficial a este ejercicio, aunque nos basaremos en el 11.1 del libro de Martí. Por tanto, pasamos a verlo, ya que es interesante este tipo de ejercicio.

Como el número a adivinar puede ser arbitrariamente grande, empezar preguntando por el 1 y seguir después en secuencia (búsqueda lineal) hasta alcanzar el número propuesto no es método práctico. En su lugar, necesitamos hacer una **búsqueda binaria**, que nos permita reducir de forma más rápida el conjunto de candidatos. Sin embargo, el algoritmo de búsqueda binaria funciona con vectores de tamaño fijo y conocido, por lo que el primero será adivinar una **cota superior** del número a adivinar (en principio una cota inferior es 1). Para encontrar dicha cota habrá que seguir un método que genere números cada vez más grandes y de modo que el incremento también aumente de forma rápida. A tal efecto, podremos utilizar las potencias de 2, por ejemplo.

Como anotación mía propia decir que en vez del código del citado libro lo sustituiremos y pondremos el que vemos en la asignatura, recordemos que el primero indicaba si el elemento estaba dentro del vector y el segundo es el propio de la búsqueda binaria:

```
funcion busquedabin (T[1..n], x)
    si n = 0 ó x > T[n] entonces devolver n + 1
    si no                          { Elemento dentro del vector }
        devolver binrec (T[1..n], x)


funcion binrec (T[i..j], x)
    { Búsqueda binaria de x en la submatriz T[i..j] con la seguridad de que
    T[i − 1] < x ≤ T[j] }
    si i = j entonces devolver i
    k ← (i + j) ÷ 2
    si x ≤ T[k] entonces devolver binrec (T[i..k], x)     { Mitad izquierda}
    si no devolver binrec (T[k + 1..j], x)                { Mitad derecha }
```

Decir que estos pseudocódigos los he copiado igualmente del compañero Antonio Rivera Cuesta, así que le agradezco personalmente la innegable aportación que siempre da.

Febrero 2008-2ª (ejercicio 3)

Enunciado: Dado n potencia de 2, escriba un algoritmo recursivo que calcule en tiempo logarítmico el valor de a^n suponiendo que sólo se puedan realizar multiplicaciones y que éstas tienen coste unitario. Demostrar el coste mediante la ecuación de recurrencia. No justifique el esquema usado, aplíquelo.

Respuesta: Este ejercicio ya lo pusimos anteriormente, así que nos evitaremos explicarlo, por lo que pondremos el pseudocódigo como sigue:

```
fun exp (a: entero; n: natural) dev entero
    si n = 1 entonces dev a
    si no
        si n = 0 entonces dev 1
        si no
            t ← exp(a, n DIV 2)
            dev t * t
        fsi
    fsi
ffun
```

Veríamos la ecuación de recurrencia, que es:

$$t(n) = 1 * t(n/2) + cte$$

De nuevo, podremos sacar las distintas variables de la **reducción por división** de la recursión como sigue:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Siendo dichas variables:

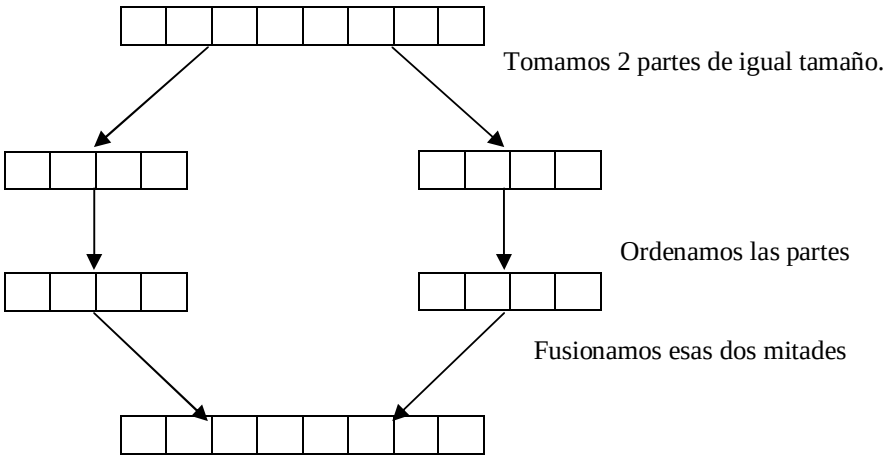
a: Número de llamadas recursivas = 1
b: Reducción del problema en cada llamada = 2
$\boldsymbol{c * n^k}$: Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 0$, al ser el tiempo extra constante.

De nuevo, sustituimos en la ecuación $a = b^k$, siendo, por tanto, $1 = 2^0$, el coste sería entonces el del segundo caso, por ello, $t(n) \in \theta\left(n^k * log(n)\right) = \theta\left(log(n)\right)$.

Septiembre 2007-reserva (ejercicio 2)

Enunciado: El algoritmo 'mergesort' posee una complejidad $T(n) \in \theta(n * \log(n))$, describa y demuestre un caso en el que 'mergesort' tiene una complejidad $T(n) \in \theta(n^2)$.

Respuesta: Recordemos que el algoritmo de ordenación por fusión (mergesort) realizaba lo siguiente:



En este caso, veíamos que estaban bien distribuidas las mitades, es decir, se dividían exactamente por dos. El peor caso, el caso en el que es cuadrático el coste corresponde cuando no lo están y queda un elemento solo para ordenar y n – 1 en la otra partición. Además, esto lo vimos en la teoría de la asignatura, que conviene estudiarla previo a hacer los ejercicios:

```
procedimiento ordenarporfusionmala (T[1..n])
    si n es suficientemente pequeño entonces
        insertar (T)
    si no
        matriz U[1..1 + ⌊n/2⌋], V[1..1 + ⌊n/2⌋]
        U[1..n − 1] ← T[1..n − 1]
        V[1] ← T[n]
        ordenarporfusion (U[1..n − 1])
        ordenarporfusion (V[1..1])
        fusionar (U, V, T)
```

Como se observa y ya se demostró en ese apartado al reducirse la recursividad en 1, tendríamos coste cuadrático. Sólo hay que sustituir los valores en la ecuación de la recurrencia y luego resolverlo con la fórmula adecuada.

2ª parte. Problemas de exámenes solucionados:

Tendremos en cuenta los pasos a seguir para resolver problemas:

- Elección del esquema (voraz, vuelta atrás, divide y vencerás).
- Identificación del problema con el esquema.
- Estructura de datos.
- Algoritmo completo (con pseudocódigo).
- Estudio del coste.

Antes de pasar a ver los problemas definiremos brevemente el preorden bien fundado, que es un preorden $\preccurlyeq$ (es un signo de precedencia) en D, se dice que es bien fundado si no existen en D sucesiones infinitas estrictamente decreciente, es decir:

$\{x_i\}$ tales que $\forall i \in N, x_{i+1} \prec x_i$

Un ejemplo de preorden bien fundado es la relación $\leq$ de los naturales o también el orden lexicográfico (es decir, el abecedario).

Ponemos esta definición en la segunda parte debido a que veremos en algunos problemas de exámenes que soliciten el preorden bien fundado, no haciéndolo en la primera parte o en general, en las cuestiones. Así mismo, dicha definición se ve mucho más ampliada en la asignatura de programación 2, por lo que para su ampliación se debería ver los temas de la citada asignatura.

Febrero 1996-1ª (problema 1) (igual a 3.3 libro de problemas resueltos)

Enunciado: El tiempo de ejecución de un algoritmo viene dado por $T(n) = 2^{n^2}$. Encontrar una forma eficiente de calcular $T(n)$, suponiendo que el coste de multiplicar dos enteros es proporcional a su tamaño en representación binaria.

Respuesta:

1. Elección razonada del esquema algorítmico

   Una forma eficiente de calcular exponenciaciones es aplicando la técnica de Divide y Vencerás y usando la relación

   $$a^n = \left(a^{n\ div\ 2}\right)^2 * a^{n\ mod\ 2}$$

   (es decir, $a^n = \left(a^{n\ div\ 2}\right)^2$ si *n* es par y $a^n = \left(a^{(n-1)\ div\ 2}\right)^2$ si *n* es impar)

   Tenemos que tener en cuenta que *div* significa el resultado de la división y *mod* el resto.

2. Descripción del esquema usado e identificación con el problema

El **esquema general** es:

```
fun divide-y-vencerás (problema)
    si suficientemente-simple (problema) entonces
        dev solucion-simple (problema)
    si no                                  { No es solución suficientemente simple }
        {p_1 .. p_k} ← decomposicion (problema)
        para cada p_i hacer
            s_i ← divide-y-vencerás (p_i)
        fpara
        dev combinacion (s_i ... s_k)
    fsi
ffun
```

Particularizaremos las siguientes funciones para este problema de la siguiente manera:

- **Tamaño umbral y solución-simple**: El preorden bien fundado para los problemas se deriva directamente del orden total entre los exponentes (n) como números naturales. Podemos tomar como tamaño umbral $n = 1$, caso en que la exponenciación es trivial y se devuelve como resultado el mismo dato de entrada.
- **Descomposición**: 2^{n^2} se descompone en un único subproblema, $2^{n^2/2}$ de la mitad de tamaño. Se trata, pues, de un problema de reducción más que descomposición. El convertir un problema de tamaño *n* en otro de tamaño $n/2$ es lo que nos proporcionará un algoritmo eficiente.
- **Combinación:** La función de combinación viene dada por la fórmula anterior para $a = 2$:

$$2^n = \left(2^{n\ div\ 2}\right)^2 * 2^{n\ mod\ 2}.$$

3. Estructuras de datos

La única estructura de datos necesaria en el problema son los **enteros**. Según las condiciones del problema, supondremos que se dispone de un algoritmo de multiplicación de enteros de coste proporcional al tamaño de éstos en representación binaria (0 ó 1). En particular, 2^n se representa en binario como un 1 seguido de n ceros, de forma que el coste de la operación $(2^n)^2$ es proporcional a *n* (por ejemplo, $2^5 = 10000$).

4. Algoritmo completo a partir del refinamiento del esquema general

Tendremos las siguientes funciones, siendo la *principal* la siguiente:

```
fun T (n: entero) dev entero
    m ← n * n
    dev exp (2, m)
ffun
```

La función *exp*, que será la que realmente realice la exponenciación (nuestro problema de divide y vencerás) es:

```
fun exp (a, n: entero) dev entero
    si n ≤ 1 entonces dev solución-simple (a, n)
    si no
        p ← n div 2              // Cociente
        r ← n mod 2              // Resto
        t ← exp (a, p)
        dev combinación (t, r, a)
    fsi
ffun
```

En los argumentos de *combinación* sólo el primero es un subproblema. Los otros dos son parámetros auxiliares que utiliza la función de *combinación*, la cual veremos a continuación:

```
fun combinación (t, r, a: entero) dev entero
    dev t * t * a^r
ffun
```

Por último, veremos la función solución-simple:

```
fun solución-simple (a, n: entero) dev entero
    si n = 1 entonces dev a
    si no dev 1
    fsi
ffun
```

5. Estudio del coste

El coste de calcular una exponenciación según este algoritmo cumple la siguiente relación:

$$T(n) = T(n/2) + cte * n$$

Es decir, calcular a^n se reduce a calcular $a^{n\ div\ 2}$ y multiplicar dos o tres enteros cuyo tamaño es del orden de *n*. Las operaciones *n div 2* y *n mod 2* tienen un coste constante sobre números en representación binaria.

La ecuación de recurrencia será:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \le n < b \\ a * T(n/b) + c * n^k & \text{si } n \ge b \end{cases}$$

La resolución de la recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Las variables de la ecuación de recurrencia correspondientes con nuestro problema serán:

a: Número de llamadas recursivas = 1
b: Reducción del problema en cada llamada = 2
$\boldsymbol{c * n^k}$: Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 1$, al ser el tiempo extra lineal.

Siguiendo el procedimiento habitual, sustituiremos en la ecuación $a = b^k$, siendo éste el caso primero, por lo que el coste será $\theta(n)$. Como la cantidad que necesitamos calcular es $T(n) = 2^{n^2}$, esa operación tendrá un coste $\boldsymbol{\theta(n^2)}$.

Si no empleáramos el algoritmo de divide y vencerás tendrá un coste $\theta(n^3)$.

Febrero 1996-2ª (problema 2) (igual a 3.1 libro de problemas resueltos)

Enunciado: Dada la sucesión definida como $f_n = af_{n-3} + bf_{n-2} + cf_{n-1}$ se pide diseñar un algoritmo que calcule un tiempo logarítmico en termino f_n.

Sugerencia: Utilizad la siguiente relación:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix} \cdot \begin{pmatrix} f_{n-3} \\ f_{n-2} \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix}$$

Respuesta:

1. Elección razonada del esquema algorítmico

   Cada término de la sucesión se calcula en función de las 3 anteriores. Por tanto, la forma trivial de calcular f_n consiste en calcular los *n* términos anteriores, lo que supone un coste lineal $O(n)$. Sin embargo, la ecuación que se nos proporciona nos da una forma más eficiente de resolver el problema, recurriendo a la técnica de **Divide y Vencerás**. Efectivamente, si llamamos:

   $$F = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$$

   Tendremos

   $$\begin{pmatrix} f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix} = F \cdot \begin{pmatrix} f_{n-3} \\ f_{n-2} \\ f_{n-1} \end{pmatrix} = F^2 \cdot \begin{pmatrix} f_{n-4} \\ f_{n-3} \\ f_{n-2} \end{pmatrix} = \cdots = F^{n-2} \cdot \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix}.$$

   donde f_0, f_1 y f_2 no pueden calcularse en término de los elementos anteriores y han de ser, por tanto, datos del problema.

   Para calcular f_n es suficiente, por tanto, con evaluar F^n y hacer una multiplicación de una matriz de 3 x 3 por un vector de 3 elementos (que tiene coste constante). Sabemos que la forma más eficiente de calcular exponenciaciones es mediante la técnica de divide y vencerás, aplicada mediante la igualdad:

   $$F^n = \left(F^{n\ div\ 2}\right)^2 * F^{n\ mod\ 2}$$

   (es decir, $F^n = \left(F^{n\ div\ 2}\right)^2$ si *n* es par y $F^n = \left(F^{(n-1)\ div\ 2}\right)^2$ si *n* es impar)

   Nos fijamos que es igual a la exponenciación del ejercicio anterior, casi sin diferencia.

2. Descripción del esquema e identificación con el problema

El esquema de Divide y Vencerás es una **técnica recursiva** que consiste en dividir un problema en varios subproblemas del mismo tipo. Las soluciones a estos subproblemas se combinan a continuación para dar la solución al problema original. Cuando los subproblemas son más pequeños que un umbral prefijado, se resuelven mediante un algoritmo especifico. Si su tamaño es mayor, se vuelven a componer.

El **esquema general** es el siguiente:

```
fun divide-y-vencerás (problema)
    si suficientemente-simple (problema) entonces
        dev solucion-simple (problema)
    si no                                   { No es solución suficientemente simple }
        {p_1 .. p_k} ← decomposicion (problema)
        para cada p_i hacer
            s_i ← divide-y-vencerás (p_i)
        fpara
        dev combinacion (s_i ... s_k)
    fsi
ffun
```

Los elementos de este esquema se particularizan así a nuestro caso:

- **Tamaño umbral y solución-simple**: El **preorden bien fundado** para los problemas se deriva directamente del orden total entre los exponentes (n) como números naturales. Podemos tomar como tamaño umbral $n = 1$, caso en que la exponenciación es trivial y se devuelve como resultado la matriz de entrada.
- **Descomposición**: F^n se descompone en un único subproblema, $F^{n/2}$ de la mitad de tamaño. Se trata, pues, de un problema de *reducción* más que descomposición. El convertir un problema de tamaño *n* en otro de tamaño *n/2* es lo que nos proporcionará un algoritmo eficiente.
- **Combinación:** La función de combinación es la ecuación mencionada más arriba.

Nos fijamos, de nuevo, que incluso las definiciones son muy parecidas a las vistas previamente en el problema anterior.

3. Estructuras de datos

En el problema intervienen solamente enteros y matrices de 3 x 3. Los datos de entrada y salida del algoritmo son enteros; las matrices se utilizan como resultados intermedios. Daremos por conocida la multiplicación de matrices.

4. Algoritmo completo

Suponemos conocidos *a*, *b*, *c*, f_0, f_1, f_2. Empezaremos viendo la <u>función principal</u>, como sigue:

fun f (n: entero) dev entero

$F \leftarrow \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$

caso $n = 0$: dev f_0;
[] $n = 1$: dev f_1;
[] $n = 2$: dev f_2;
[] verdadero: hacer
$S \leftarrow$ exp_mat (F, n/2)

$s \leftarrow S \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix}$

dev s[3]
fcaso
ffun

La siguiente función es la <u>exponencial</u>:

fun exp_mat (M: vector[3,3] de enteros, n: entero) dev vector[3,3] de enteros

si $n \leq 1$ entonces
devolver solución_simple (M, m)
si no
$p \leftarrow n\ div\ 2$
$r \leftarrow n\ mod\ 2$
$T \leftarrow$ exp_mat (M, p)
dev combinacion (T, r, M)
fsi
ffun

Se modifica algo la anterior función, ya que el vector es de enteros, tanto el del argumento como el que devuelve.

En los argumentos de <u>combinación</u> sólo el primero es un subproblema. Los otros dos son parámetros auxiliares que utiliza la función de combinación (recordemos que lo vimos en el ejercicio anterior):

fun combinación (T: vector[3,3] de enteros, r: entero, M: vector[3,3] de enteros) dev entero
dev $T * T * M^r$
ffun

donde $M^1 = M; M^0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$

Por último, veremos la función solución-simple:

```
fun solución-simple (M: vector[3,3] de enteros, n: entero) dev entero
    si n = 1 entonces dev M
    si no
        dev ( 1  0  0 )
            ( 0  1  0 )
            ( 0  0  1 )
    fsi
ffun
```

5. Estudio del coste

El coste del algoritmo recae sobre la exponenciación de la matriz F. Como se reduce un problema de tamaño n a otro de tamaño n/2, el coste cumple la **ecuación de recurrencia** $T(n) = T(n/2) + cte$, donde *cte* hace referencia al tiempo necesario para hacer una o dos multiplicaciones de matrices *3 x 3*. Seguiremos con nuestro planteamiento del ejercicio anterior, siendo ambos casi idénticos.

La ecuación de recurrencia de reducción por división será:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Las variables de la ecuación de recurrencia correspondientes con nuestro problema serán:

a: Número de llamadas recursivas = 1
b: Reducción del problema en cada llamada = 2
$\boldsymbol{c * n^k}$: Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 0$, al ser el tiempo extra constante, lo que vimos previamente.

Siguiendo el procedimiento habitual, sustituiremos en la ecuación $a = b^k$, siendo éste el caso segundo, por lo que el coste será $\theta(\log(n))$. Hemos visto ejercicios anteriores éste sería el coste de este algoritmo, como puede ser el de búsqueda binaria.

NOTA DEL AUTOR: Este problema se parece bastante al de Febrero de 2003-2ª semana, la única diferencia es que la sucesión definida es $f_n = af_{n-1} + bf_{n-2}$, por lo que hay que hacer 2 sumas en vez de 3 que nos solicitan en este ejercicio. Tenía idea ponerlo aparte, pero me he dado cuenta que es repetir el mismo ejercicio de manera absurda.

Septiembre 1996-reserva (problema 1) (igual a 3.2 libro de problemas resueltos)

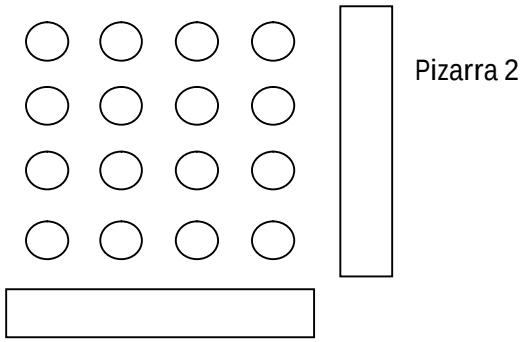
Enunciado: En una clase hay f filas y c columnas de pupitres. Delante de la primera fila se encuentra la pizarra:

a) Diseñar un algoritmo que reparta los $f * c$ alumnos de forma que, al mirar hacia la pizarra, ninguno se vea estorbado por otro alumno más alto que él.
b) A mitad del curso se coloca una pizarra adicional en una de las paredes adyacentes con la primera pizarra. Diseñar un algoritmo que coloque a los alumnos de forma que pueden mirar también a esa segunda pizarra sin estorbarse. Este algoritmo debe sacar partido de la colocación anterior.

Respuesta:

1. Elección razonada del esquema algorítmico

   Para que nadie tenga un alumno más alto que él al mirar la pizarra, es necesario que, dentro de cada columna, los alumnos están ordenados según su altura de mayor a menor:



   Para resolver el apartado a) de forma eficiente basta con **dividir** los $f * c$ alumnos en *c* subconjuntos de *f* elementos escogidos al azar y, a continuación, debe ordenarse cada uno de esos subconjuntos. Cada uno de ellos será una columna en la clase. Como algoritmo de ordenación puede escogerse cualquiera de los estudiados; nosotros utilizaremos el algoritmo de Divide y Vencerás de *fusión*, por ser más eficiente asintóticamente en el caso peor.

   Al colocar una segunda pizarra adyacente a la primera, los alumnos de cada fila deben estar, a su vez, ordenados entre sí. Para que estén ordenadas las columnas y las filas, es necesario ordenar a todos los alumnos de menor a mayor, y colocarlos en la clase de forma que el más bajito ocupe el pupitre que está en la intersección de las dos pizarras, y el más alto en el vértice opuesto de la clase. Por lo tanto, para obtener la disposición final de los alumnos en el apartado b) debe hacerse una ordenación de $f * c$ elementos. Pero si aprovechamos la disposición anterior no es necesario, esta vez, aplicar ningún algoritmo de ordenación: basta con realizar una fusión de *c* subconjuntos ya ordenados (equivaldría al último paso de un algoritmo de ordenación por fusión en el que el factor de división fuera *c*. Así, la ordenación final puede obtenerse en **tiempo lineal**.

2. Descripción del esquema usado e identificación con el problema

Dado el esquema de divide y vencerás:

```
fun divide-y-vencerás (problema)
    si suficientemente-simple (problema) entonces
        dev solucion-simple (problema)
    si no                                   { No es solución suficientemente simple }
        {p_1 .. p_k} ← decomposicion (problema)
        para cada p_i hacer
            s_i ← divide-y-vencerás (p_i)
        fpara
        dev combinacion (s_i ... s_k)
    fsi
ffun
```

Se particulariza para nuestro problema de la siguiente manera:

- **Tamaño umbral y solución-simple**: El preorden bien fundado para los problemas se deriva directamente del orden total entre el tamaño de los subconjuntos problema. Podemos tomar como tamaño umbral $n = 1$, caso en que la exponenciación es trivial y consiste simplemente en devolver el elemento.
- **Descomposición**: Dividiremos el conjunto problema en dos subconjuntos formados por los n *div* 2 primeros elementos por un lado y el resto por otro.
- **Combinación:** Es necesario fundir los dos subconjuntos ordenados, mediante un bucle que toma cada vez el menor elemento de entre los primeros de uno y otro subconjunto todavía sin seleccionar.

3. Estructuras de datos

La única estructura de datos que necesitamos es una matriz de enteros de tamaño $f * c$ que almacene las alturas de los alumnos. También podremos utilizar un *vector* de tamaño $f * c$ sabiendo que cada *f* elementos representa una columna, aunque es menos natural.

4. Algoritmo completo a partir del refinamiento del esquema general

Llamaremos *a* a la función que obtiene la ordenación requerida en el apartado a) y *b* a la que soluciona en el apartado b), es decir, obtiene una ordenación total a partir de la que se tiene en a).

La función *a* es la siguiente:

```
fun a (clase: vector[1..f, 1..c] de reales) dev vector[1..f, 1..c] de enteros
    para i ← 1 hasta c hacer
        clase[1..f, i] ← ordenacion (clase[1..f, i])
    fpara
    dev clase
ffun
```

La función ordenación equivale a la función general divide y vencerás, que modificaremos ligeramente para incluir como argumento vectores que sean siempre del mismo tamaño:

```
fun ordenación (v: vector[1..n]; i, j: entero)
    si i ≠ j entonces
        k ← (i + j) div 2
        ordenación (v, i, k)
        ordenación (v, k + 1, j)
        fusión (v, i, j, 2)
    fsi
ffun
```

La función de fusión tiene **cuatro parámetros**: el vector, el principio y final del tramo que contiene a los dos subvectores que hay que fusionar, y el factor de división empleado. Para el apartado a) podríamos ahorrarnos este último argumento (es 2), pero esa generalización nos permitirá usar la misma función que el apartado b).

```
proc fusión (v: vector [1..n]; inicial, final, factor: entero)
    { Inicializa el vector solución }
    v' ← vector [1..n]
    { Inicializa punteros al comienzo de los vectores por fusionar }
    para k ← 1 hasta factor hacer
        i_k ← inicio + factor * (k − 1)
    fpara
    I ← {i_1 ... i_factor}
    { Selecciona el menor elemento de entre los principios de vector para
    incluirlo en la solución, y a continuación lo borra para que no vuelva a ser
    considerado }
    para h ← inicio hasta final hacer
        i_x ← elemento que maximiza v[i_x]
        v'[i_x] ← v[i_x]                    // Guarda en vector solución
        si i_x < inicio + factor * x − 1 entonces
            i_x ← i_x + 1                   // Incrementa valor del puntero
        si no
            I ← I\{i_x}
        fsi
    fpara
    v ← v';
fproc
```

La función *b* debe conseguir una ordenación completa del conjunto de alumnos, pero debe tener en cuenta que ya exista una ordenación parcial entre ellos: los alumnos de cada columna están ordenados entre sí. Si representamos el conjunto de los alumnos como un vector $f * c$ elementos, en el que los *f* primeros elementos corresponden a la primera columna, los *f* siguientes a la segunda, etc. El problema queda solucionado llamando a la función de *fusión* definida anteriormente, pero utilizando un factor de división *c* en lugar de 2:

```
fun b (v: vector[1..c * f]; c: entero) dev vector[1..c * f]
    fusión (v, 1, c * f, c)
ffun
```

5. Estudio del coste

Apartado a

La ordenación por fusión tiene un coste que cumple:

$$T(n) = 2 * T(n/2) + cte * n$$

De nuevo estamos ante un caso de reducción de la **recursividad por división**. El algoritmo de fusión tiene un coste $O(n)$ (consta de dos bucles consecutivos). De esa igualdad se obtiene un coste $O(n * \log(n))$. Como se realizan c combinaciones de f elementos cada una, el coste total es $\boldsymbol{O(c(f * \log(f)))}$ (estimo que es una ordenación parcial, de ahí ese coste).

Mediante una ordenación total habríamos resuelto también el problema, pero con un coste $O(n * \log(n)) \rightarrow \boldsymbol{O(cf * log(cf))}$.

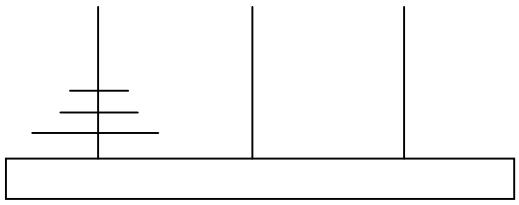
Apartado b

Se resuelve mediante una llamada al algoritmo de fusión, que tiene un coste lineal; como el tamaño del problema es $c * f$, el coste es $\boldsymbol{O(c * f)}$. El coste es mucho menor que en el caso que no aprovechemos la ordenación parcial que se obtiene en el apartado a.

NOTA DEL AUTOR: Se observa que los vectores, salvo algunos añadidos míos no se saben de qué tipo de son, poniendo sólo $\mathrm{vector}[1..n]$, por no estar seguro del mismo. Se dejará como ejercicio el hacerlos, ya que desconozco de que tipos pueden ser, si reales o enteros.

Problema 3.4 del libro de problemas resueltos (sin correspondencia con ningún ejercicio de examen)

Enunciado: Se tienen 3 varillas y n discos agujereados por el centro. Los discos son todos de diferente tamaño y en la posición inicial se tienen los discos insertados en el primer palo y ordenados en tamaños decrecientes desde la base hasta la punta de la varilla. El problema consiste en pasar los discos de la primera a la tercera varilla observando las siguientes reglas: se mueven los discos de uno en uno y nunca un disco puede colocarse encima de otro menor que éste.



Respuesta:

1. Elección razonada del algoritmo

   Aparentemente el problema parece solucionarse mediante una búsqueda en un árbol de búsqueda como vimos con los juegos en el tema 9 (de vuelta atrás), sin embargo, hay una forma que nos asegura que utilizamos el mínimo número de movimientos mediante Divide y Vencerás. La justificación de la elección se basa, por tanto, en la **eficiencia** más que en la imposibilidad de utilizar el esquema de Vuelta Atrás.

   Por otra parte, hay que indicar que un esquema voraz es imposible de aplicar, ya que la solución no consta de unos elementos elegidos de un conjunto de candidatos, sino de los movimientos efectuados sobre ellos. Para que un conjunto de candidatos de tal naturaleza

fuera posible, habría que añadir a cada movimiento información acerca del número de discos que descansa sobre cada varilla, lo cual es tanto como enumerar todas las posibles combinaciones del juego.

2. Descripción del esquema usado e identificación con el problema

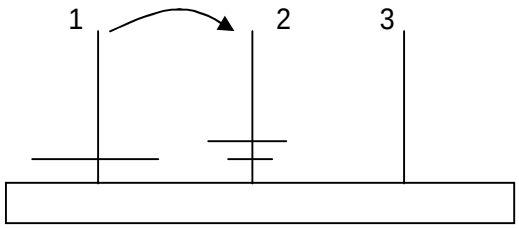
El **esquema general** es:

```
fun divide-y-vencerás (problema)
    si suficientemente-simple (problema) entonces
        dev solucion-simple (problema)
    si no                                  { No es solución suficientemente simple }
        {p_1 .. p_k} ← decomposicion (problema)
        para cada p_i hacer
            s_i ← divide-y-vencerás (p_i)
        fpara
        dev combinacion (s_i ... s_k)
    fsi
ffun
```

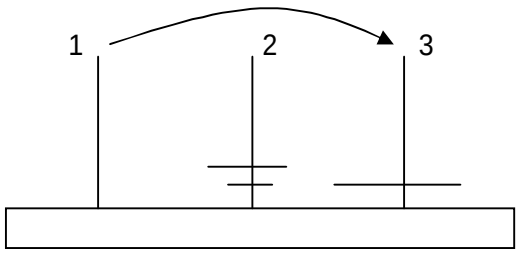
y se particulariza de la siguiente forma:

- **Tamaño umbral y solución-simple**: El preorden bien fundado se establece sobre los tamaños de los sucesivos problemas en la cadena de llamadas recursivas. El único problema verdaderamente simple es el de solucionar el pasatiempo para un único disco, por lo que el tamaño umbral es $n = 1$.
- **Descomposición**: Cualquier solución para pasar n discos del poste 1 al 3 pasa por:

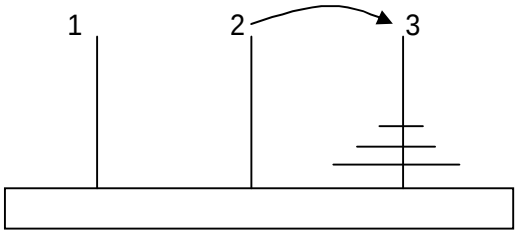
    1. Conseguir pasar $n - 1$ discos del poste 1 al 2. Siguiendo las condiciones del enunciado sería algo así:



    2. Pasar el disco que queda (que además será el de mayor radio de los que habrá en el poste 1) del 1 al 3:



3. Pasar, por último, los $n-1$ discos que habíamos puesto en el poste auxiliar 2, de dicho poste al 3:



En general, si tenemos *k* discos en *i* y queremos pasarlo a *j*, siendo $i, j \in \{1,2,3\}$ habrá que calcular $6-i-j$ para hallar el número del poste restante que se utilizará como *auxiliar* y se siguen los pasos anteriores.

NOTA DEL AUTOR: He hecho los dibujos para que se vea gráficamente este ejercicio paso a paso. Creo que así queda más claro.

- **Combinación:** No hay función de combinación. Las soluciones parciales son validas en sí mismas. En el caso que nos ocupa, éstas (las soluciones) son movimientos expresados mediante origen y destino.

3. Estructuras de datos

No hay estructuras complejas de datos. Se utilizarán únicamente tipos de datos **estándar**.

4. Algoritmo completo a partir del refinamiento del esquema general

Los parámetros del algoritmo serán la varilla origen, la varilla destino y el número de discos que deben ser movidos del origen al destino. La llamada inicial tendrá como origen la varilla 1, como destino la 3 y con un número *n* de discos.

El **algoritmo refinado** se expone a continuación:

```
fun hanoi (origen, destino, n: entero)
    si n = 1 entonces
        escribe "Mover disco de poste origen a poste destino"
    si no
        hanoi (origen, 6 – destino – origen, n – 1)
        hanoi (origen, destino, 1)
        hanoi (6 – origen – destino, destino, n – 1)
    fsi
ffun
```

La función de combinación **no** es necesaria, ya que cada descomposición del problema en un subproblema produce un movimiento y dos llamadas a la función.

5. Estudio del coste

Debemos plantear la ecuación de recurrencia. En el caso que nos ocupa, al entrar a la función tenemos una instrucción *si … entonces* con coste 1 seguida de 3 llamadas a la función, de manera que $T(n) = 2 * T(n-1) + T(1) + 1$. Simplificando $T(1)$ tenemos finalmente que $T(n) = 2 * T(n-1) + 2$ y resolviendo con las formulas antes puestas tiene coste del orden de $O(2^n)$.

Problema 3.5 del libro de problemas resueltos (sin correspondencia con ningún ejercicio de examen)

Enunciado: Se tiene un vector de enteros no repetidos y ordenados de menor a mayor. Diseñar un algoritmo que compruebe en tiempo logarítmico si existe algún elemento del vector que coincida con su índice.

Respuesta:

1. Elección razonada del esquema algorítmico

   Se trata del esquema de **Divide y Vencerás** por dos motivos:

   - Se exige coste logarítmico y esto solo es posible si el problema se reduce (al menos) a la mitad en cada paso (mucho ojito con esto, en muchas ocasiones es determinante).
   - Se puede descomponer la estructura de datos en partes de su misma naturaleza.

   Un esquema de búsqueda ciega en el vector carece por otra parte, de sentido (no es vuelta atrás) y tampoco se trata de un esquema voraz, ya que no hay candidatos que haya que escoger ningún tipo de conjunto (no es voraz).

2. Descripción del esquema usado e identificación con el problema

   El **esquema general** es:

```
fun divide-y-vencerás (problema)
    si suficientemente-simple (problema) entonces
        dev solucion-simple (problema)
    si no                                  { No es solución suficientemente simple }
        {p_1 .. p_k} ← decomposicion (problema)
        para cada p_i hacer
            s_i ← divide-y-vencerás (p_i)
        fpara
        dev combinacion (s_i ... s_k)
    fsi
ffun
```

   Se trata de un problema en el que no hay que efectuar una combinación posterior de las soluciones. A este tipo de problemas se les denomina de **reducción o simplificación** dentro del esquema de Divide y Vencerás.

   Lo más importante de este tipo de problemas de simplificación es que hay que tener en cuenta con qué mitad del vector nos quedamos para seguir buscando. Este criterio es prácticamente lo único que tenemos que decidir, ya que el algoritmo debe sólo devolver falso o dar un índice del vector.

   Supongamos que es *i* es el elemento del vector *v*, o bien *i*, *j* los elementos centrales en caso de que tengan un número par de elementos. Si $v[i] < i$ sabemos que podemos descartar la parte izquierda del vector como susceptible de contener en ella algún elemento tal que $v[i] = i$. Lo mismo le ocurre a la derecha para la condición contraria. Esto ocurre por dos motivos:

   - Porque se trata de números enteros.
   - Porque se prohíben las repeticiones.

Se debería hacer la comprobación de que son dos condiciones imprescindibles para que el algoritmo pueda ser resuelto con los requisitos de coste del enunciado.

Lo único que resta es comprobar las condiciones para vectores con un número par e impar de elementos. Para dividir el vector se utilizará un índice sobre él que se pasa como argumento.

3. Estructuras de datos

No hay más estructura de datos que el **vector de enteros**.

4. Algoritmo completo a partir del refinamiento del esquema general

La *condición trivial* se aplica a vectores de un solo elemento:

```
fun solución-trivial (v: vector; i: natural) dev boolean
    dev (v[i] = i)
ffun
```

El *esquema refinado* será el siguiente:

```
fun indice (v: vector; i, j: natural)
    si i = j entonces solución-trivial (v, i)
    si no
        k ← (i + 1) div 2
        si v[k] ≥ k entonces índice (v, i, k)
        si no
            índice (v, i, k)
        fsi
    fsi
ffun
```

NOTA DEL AUTOR: Esta parte de problemas del libro de problemas resueltos por norma general están bastante bien hechos y claros, sólo que este ejercicio tiene una errata en la condición del "si" más interior, es decir, en la expresión $v[k] \geq k$, que hemos tratado de subsanarlo. Como es habitual, está hecho por un alumno, así que no se sabe si está realmente bien, aunque se intenta. Nos fijamos, además, que es de nuevo una *búsqueda binaria*, con las implicaciones de coste que tanto hacemos hincapié.

5. Estudio del coste

La ecuación de recurrencia del algoritmo es $T(n) = T(n/2) + 1$. Su resolución es, como esperábamos (en este caso, no lo haremos por haberlo hecho en innumerables ocasiones), $T(n) \in \log(n)$.

Febrero 2002-1ª (problema)

Enunciado: una caja con n bombones se considera "aburrida" si se repite un mismo tipo de bombón (denominado bombón "pesado") más de $n/2$ veces. Programar el pseudocódigo un algoritmo que decida si una caja es "aburrida" y devuelva (en su caso) el tipo de bombón que le confiere dicha propiedad. El coste debe ser a lo sumo $\theta(\text{n} * \log(n))$.

NOTA: Si una caja tiene un bombón "pesado", entonces necesariamente también lo es de al menos una de sus dos mitades.

Respuesta:

Este ejercicio no está resuelto por el equipo docente (si no estoy equivocada), por lo que se toma esta solución de otro sitio distinto. Notaremos que todos los ejercicios de este mismo tipo (de contar votos o similares) se parecen entre sí mucho, y que posteriormente veremos.

1. Elección razonada del esquema algorítmico

   Se trata del esquema de Divide y Vencerás, ya que se puede descomponer la estructura de datos en partes de su misma naturaleza, resolviendo el problema con el coste requerido. Además, la propiedad de la nota del enunciado hace referencia a las mitades del problema.

   El tamaño del problema viene dado por el número de bombones que es un natural. En los naturales no hay sucesiones decrecientes infinitas, por tanto, tendremos un **preorden bien fundado**.

2. Descripción del esquema usado e identificación con el problema

   El esquema de Divide y Vencerás es una técnica recursiva que consiste en **dividir** el problema en varios subproblemas del mismo tipo. Las soluciones a estos subproblemas se **combinan** a continuación para dar la solución al problema original. Cuando los subproblemas son más pequeños que un umbral prefijado, se resuelven mediante un algoritmo especifico. Si su tamaño es mayor, se vuelven a descomponer.

   El **esquema general**, por tanto, es el siguiente:

```
fun divide-y-vencerás (problema)
    si suficientemente-simple (problema) entonces
        dev solucion-simple (problema)
    si no                                  { No es solución suficientemente simple }
        {p_1 .. p_k} ← decomposicion (problema)
        para cada p_i hacer
            s_i ← divide-y-vencerás (p_i)
        fpara
        dev combinacion (s_i ... s_k)
    fsi
ffun
```

   Se particulariza estos elementos a nuestro caso:

   - **Tamaño umbral y solución-simple**: El preorden se establece sobre los tamaños de los sucesivos problemas en la cadena de llamadas recursivas. El único problema verdaderamente simple es el de solucionar el problema para una caja de bombón, por lo que el tamaño umbral es $n = 1$.
   - **Descomposición**: Dividiremos el conjunto problema en dos subproblemas de tamaño mitad $(n/2)$.

- **Combinación:** Las dos llamadas devuelven:

  $\langle b_1, e_1 \rangle$ $\langle b_2, e_2 \rangle$

  Siendo b_1 y b_2 dos booleanos que nos informan de la existencia de "bombón pesado" en cada una de las dos mitades respectivas, y e_1 y e_2 los "bombones pesados", en caso de que alguno o ambos existan.

  Tendremos que resolver el problema para que nos dé la solución del "bombón pesado" en todo el vector, es decir, hallar sabiendo las mitades cuál es la solución $\langle b, e \rangle$. Para ello, veremos el enunciado:

  Si e es pesado en el vector $[1..n]$, lo será en la primera mitad o en la segunda. Como ejemplo, podrá ser:

A	A		A	B

  Sin embargo, la implicación en el otro sentido no es cierta. Por ejemplo:

A	A		B	B

  Por tanto, un "bombón pesado" en alguna de las mitades sólo será candidato a "bombón pesado" en todo el vector, requiriendo un conteo posterior del mismo.

  Tendremos estos **casos** a distinguir:

  1. Si $b_1 = falso$ y $b_2 = falso$, entonces significa que no hay ningún candidato y, por tanto, deducimos que $b = falso$.
  2. Si $b_1 = verdadero$ y $b_2 = falso$, entonces se ha encontrado un candidato que es el e_1. Por ello, hay que contar en todo el vector si supera la mitad de los elementos de dicho candidato. Es decir, verificamos esta condición:

     contar $(e_1, v[1..n]) > n/2$

     dándose estos casos posibles:

     - SI: Existe "bombón pesado" y es el candidato e_1, es decir, $b = verdadero, e = e_1$.
     - NO: No existe "bombón pesado", o lo que es igual, $b = falso$.
  3. Si $b_1 = falso$ y $b_2 = verdadero$, entonces se ha encontrado un candidato que es el e_2. Por ello, hay que contar en todo el vector si supera la mitad de los elementos de dicho candidato. Es decir, verificamos esta condición:

     contar $(e_2, v[1..n]) > n/2$

     dándose estos casos posibles:

     - SI: Existe "bombón pesado" y es el candidato e_2, es decir, $b = verdadero, e = e_2$.
     - NO: No existe "bombón pesado", o lo que es igual, $b = falso$.

4. El último caso es el más especial y es aquél en el que $b_1 = verdadero$ y $b_2 = verdadero$, es decir, ha encontrado dos candidatos distintos en ambas partes, por lo que se tiene que verificar si en todo el vector sale el candidato e_1. Al igual que antes tendríamos que comprobar está condición:

   contar $(e_1, v[1..n]) > n/2$

   dándose estos casos posibles:

   - SI: Existe "bombón pesado" y es el candidato e_1, es decir, $b = verdadero, e = e_1$.
   - NO: Esto significa que podría ser que el otro candidato posible fuera "bombón pesado" por lo que tendríamos de nuevo que contar, así que de nuevo verificaríamos está condición:

     contar $(e_2, v[1..n]) > n/2$

     de nuevo, tendremos otros subcasos, que serian los siguientes:

     - SI: Existe "bombón pesado" y es el candidato e_2, es decir, $b = verdadero, e = e_2$.
     - NO: No existe "bombón pesado", o lo que es igual, $b = falso$.

3. Estructuras de datos

   La única estructura de datos que necesitamos es un **vector de *n* componentes** para almacenar los bombones; para representar un bombón se puede utilizar un tipo estándar, teniendo siempre en cuenta que los bombones en principio no tienen porque admitir ninguna ordenación.

4. Algoritmo completo a partir del refinamiento del esquema general

Siguiendo los casos previos es tan fácil como transcribirlos a pseudocódigo como sigue:

```
fun pesado (v[1..n] de Tbombon, c, f: natural) dev <b: boolean, e: Tbombon >
    si c = f entonces
        b ← cierto
        e ← v[c]
    si no
        m ← (c + f) div 2
        ⟨b1, e1⟩ ← pesado (v, c, m)
        ⟨b2, e2⟩ ← pesado (v, m + 1, f)
        casos
            b1 = falso ∧ b2 = falso: b← cierto
            b1 = falso ∧ b2 =cierto: num ← contar (e2, v, c, f)
                si num > (f − c + 1) div 2 entonces
                    b ← cierto; e ← e2
                si no
                    b ← falso
                fsi
            b1 = cierto ∧ b2 = falso: num ← contar (e1, v, c, f)
                si num > (f − c + 1) div 2 entonces
                    b ← cierto; e ← e1
                si no
                    b ← falso
                fsi
            b1 = cierto ∧ b2 = cierto: num ← contar (e1, v, c, f)
                si num > (f − c + 1) div 2 entonces
                    b ← cierto; e ← e1
                si no
                    num ← contar (e2, v, c, f)
                    si num > (f − c + 1) div 2 entonces
                        b ← cierto; e ← e1
                    si no
                        b ← falso
                    fsi
                fsi
        fcasos
    fsi
ffun
```

5. Estudio del coste

   Veremos, como en ocasiones anteriores la **ecuación de recurrencia** será:

$$T(n) = \begin{cases} c * n & \text{si } n = 1 \\ 2 * T(n/2) + c * n & \text{si } n \geq 1 \end{cases}$$

   La resolución de la recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

   Las variables de la ecuación de recurrencia correspondientes con nuestro problema serán:

   a: Número de llamadas recursivas = 2
   b: Reducción del problema en cada llamada = 2
   $\boldsymbol{c * n^k}$: Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 1$, al ser el tiempo extra lineal.

   Siguiendo el procedimiento habitual, sustituiremos en la ecuación $a = b^k$, siendo éste el caso segundo, por lo que el coste será $\boldsymbol{\theta(n * log(n))}$.

   NOTA (del ejercicio): El coste de contar es lineal.

Febrero 2005-2ª (problema)

Enunciado: Utiliza el esquema de divide y vencerás para implementar una función que tome un vector de enteros y le dé estructura de montículo con el menor coste posible.

Respuesta:

Lo interesante de este problema es que emplean cambios de variables para resolverlo. Nunca antes hemos visto hecho así ningún problema en este tema. Para ello, convendría repasar el tema de costes (tema 4) de Brassard y el de estructuras de datos (tema 5), para así entender mejor el problema. Pasamos a explicarlo a continuación.

1. Elección del esquema

   No es necesario razonar la elección del esquema puesto que viene impuesto en el problema.

2. Esquema general

   El esquema general de **divide y vencerás** consiste en:

   - **Descomponer** el ejemplar en subejemplares del mismo tipo que el original.
   - **Resolver** independientemente cada subejemplar (subproblemas).
   - **Combinar** los resultados para construir la solución del ejemplar original.

En los problemas de los últimos años se nos da un esquema alternativo que, por curiosidad pasamos a escribirlo, aunque como es habitual tomaremos el nuestro empleado en todos los ejercicios hasta el momento:

```
fun divide-y-vencerás (X)
    si suficientemente-pequeño (X) entonces
        dev subalgoritmo_basico (X)
    si no
        decomponer (X, X1 .. Xn)
        para i = 1 hasta n hacer
            Yi := divide-y-vencerás (p_i)
        fpara
        recombinar (Y1 .. Yn, Y)
        dev Y
    fsi
ffun
```

El **esquema general** es el siguiente:

```
fun divide-y-vencerás (problema)
    si suficientemente-simple (problema) entonces
        dev solucion-simple (problema)
    si no                                  { No es solución suficientemente simple }
        {p_1 .. p_k} ← decomposicion (problema)
        para cada p_i hacer
            s_i ← divide-y-vencerás (p_i)
        fpara
        dev combinacion (s_i ... s_k)
    fsi
ffun
```

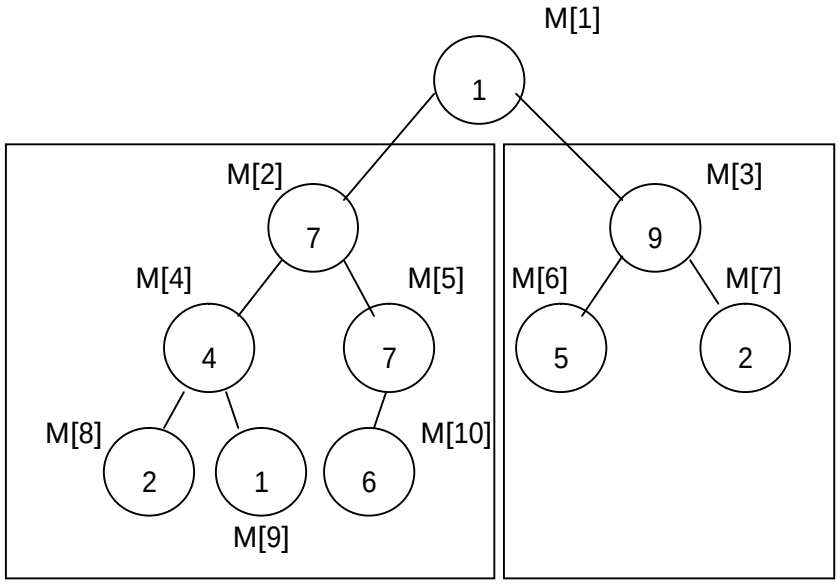
3. Estructuras de datos

No se necesita ninguna estructura adicional a partir del vector de entrada.

4. Algoritmo completo

Como se trata de un árbol binario descomponemos el problema en los dos subárboles bajo el nodo *i*, es decir, el que tiene por raíz $2 * i$ y el que tiene por raíz $2 * i + 1$. Cada subárbol a su vez debe tener estructura de montículo, lo que resulta un problema del mismo tipo que el original y, por tanto, corresponde a sendas llamadas recursivas, una para $2 * i$ y otra para $2 * i + 1$. Por último, hay que colocar la raíz *i* en su lugar. Para ello hay que *hundirla*.

Ejemplo:



El algoritmo será, por tanto, el siguiente:

```
proc crear-montículo (M[1..m], i)
    si 2 * i < m entonces
        crear-montículo (M, 2 * i)
    fsi
    si (2 * i + 1) < m entonces
        crear-montículo (M, 2 * i + 1)
    fsi
    si 2 * i ≤ m entonces
        hundir (M, i)
    fsi
fproc
```

se ha hecho una versión iterativa del crear montículo algo distinta a como lo vimos en el tema 5, pero igualmente válida aunque pondremos para refrescar la memoria la versión recursiva:

```
procedimiento crear-montículo (T[1..n])
    { Este procedimiento transforma la matriz T[1..n] en un montículo }
    para i ← ⌊n/2⌋ bajando hasta 1 hundir (T, i)
```

Como puede observarse, dividir el vector en dos mitades $M[1..m/2]$, $M[(m/2) + 1..m]$ y tratar de resolver cada una de ellas supone un error. Una solución que recorra completamente el vector y proceda a *hundir* o *flotar* cada elemento siempre tendrá mayor coste.

Ya que estamos en un esquema de divide y vencerás, podemos plantear el procedimiento hundir como un problema de reducción. Únicamente, se tiene que decidir si el nodo debe intercambiarse por alguno de sus hijos y, en caso afirmativo, intercambiarse con el mayor de ellos y realizar la correspondiente llamada recursiva para seguir *hundiéndose*.

El procedimiento hundir es el siguiente:

```
proc hundir (T[1..n], i)
    hmayor ← i;
    { Buscar el hijo mayor del nodo i }
    si (2 * i ≤ n) y (T[2 * i] > T[hmayor]) entonces
        hmayor = 2 * i;
    fsi
    si (2 * i + 1 ≤ n)  y (T[2 * i + 1] > T[hmayor]) entonces
        hmayor = 2 * i + 1;
    fsi
    { Si cualquier hijo es estrictamente mayor que el padre }
    si (hmayor > i) entonces
        intercambiar T[i] y T[hmayor]
        hundir-rec (T, hmayor)
    fsi
fproc
```

5. Estudio del coste

Esta parte es muy importante, ya que si fuera mayor el coste se puntuaría como 0 el ejercicio. El coste del procedimiento *hundir* se puede plantear mediante una r**ecurrencia con reducción del problema por división**, ya que *hundir* prosigue por uno de los dos subárboles y, por tanto, el problema se ha reducido a la mitad, siendo ésta la ecuación de recurrencia:

$$T(n) = \begin{cases} c * n^k & \text{si } 1 \leq n < b \\ a * T(n/b) + c * n^k & \text{si } n \geq b \end{cases}$$

La resolución de la ecuación de recurrencia es:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Las distintas variables son:

a: Número de llamadas recursivas = 1

b: Reducción del problema en cada llamada = 2

$\boldsymbol{c * n^k}$: Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 0$, al ser el tiempo extra constante.

Como en ocasiones anteriores y evitando el cálculo de la resolución de la recurrencia, tendremos que el coste $\theta(\log(n))$.

El coste de crear-montículo puede expresarse mediante la siguiente **ecuación de recurrencia**:

$T(n) = 2 * T(n/2) + \log(n)$

Sin embargo, esta recurrencia no se puede resolver con la fórmula anterior, puesto que la parte no recursiva no tiene una complejidad polinomial. Para resolverlo, vamos a utilizar un **cambio de variable**:

Sea h la altura del montículo de n nodos: $h = \log_2(n)$. En cada llamada recursiva bajamos un nivel por lo que el problema se puede expresar mediante **una recurrencia con reducción del problema por sustracción**. Es decir, si en cada paso se baja un nivel, el problema se reduce a uno. Igualmente tenemos la ecuación de recurrencia y su resolución, como vimos anteriormente es la siguiente:

$$T(n) = \begin{cases} c * n^k & \text{si } 0 \leq n < b \\ a * T(n-b) + c * n^k & \text{si } n \geq b \end{cases}$$

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta\left(a^{n \; div \; b}\right) & \text{si } a > 1 \end{cases}$$

En el caso de *hundir* $T(h) = T(h-1) + c$, con $a = 1$ y $b = 1$, luego $T(h) \in \theta(h)$ que deshaciendo el cambio de variable lleva un tiempo *en función de n*:

$T(n) \in \theta(\log(n))$

Como habíamos demostrado previamente.

Sin embargo, ahora ya podemos plantear la recurrencia para *crear-montículo*:

$T(h) = 2 * T(h-1) + h$, donde $a = 2$ y $b = 1$ y, por tanto, $T(h) \in \theta(2^h)$.

Deshaciendo el cambio de variable:

$2^h = 2^{\log_2(n)} = n$ y $T(n) \in \theta(n)$, que es el menor coste posible para dar estructura de montículo a un vector.

Septiembre 2005 (problema) (parecido a problema Septiembre 2000)

Enunciado: Sea $V[1..n]$ un vector con la votación de una elecciones. La componente $V[1..n]$ contiene el nombre del candidato que ha elegido el votante i. Implementa un programa cuya función principal siga el esquema de divide y vencerás, que decida si algún candidato aparece en más de la mitad de los componentes (tiene mayoría absoluta) y que devuelva su nombre. Sirva como ayuda que para que un candidato tenga mayoría absoluta considerando todo el vector (al menos $N/2 + 1$ de los N votos), es condición necesaria pero no suficiente que tenga mayoría absoluta en alguna de las mitades del vector. La resolución del problema debe incluir, por este orden:

1. Descripción del esquema de divide y vencerás y su aplicación al problema
2. Algoritmo completo a partir del refinamiento del esquema general.
3. Estudio del coste del algoritmo desarrollado.

Respuesta:

En este ejercicio al igual que en algunos otros, veremos los 5 pasos habituales aunque de modo condensado en 3.

1. Descripción del esquema de divide y vencerás y su aplicación al problema

   El esquema general de **divide y vencerás** consiste en:

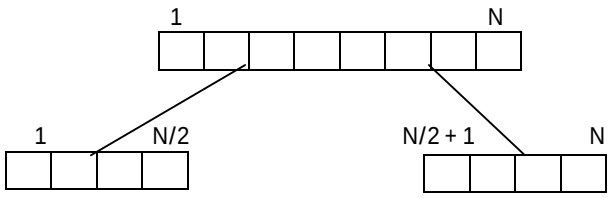
   - **Descomponer** el ejemplar en subejemplares del mismo tipo que el original.
   - **Resolver** independientemente cada subejemplar (subproblemas).
   - **Combinar** los resultados para construir la solución del ejemplar original.

   Más formalmente:

```
fun divide-y-vencerás (problema)
    si suficientemente-simple (problema) entonces
        dev solucion-simple (problema)
    si no                                  { No es solución suficientemente simple }
        {p_1 .. p_k} ← decomposicion (problema)
        para cada p_i hacer
            s_i ← divide-y-vencerás (p_i)
        fpara
        dev combinacion (s_i ... s_k)
    fsi
ffun
```

   Vamos a hacer una ampliación del problema propuesto, por lo que añadiremos en este apartado las funciones que se particularizan habitualmente. Realmente este problema es bastante parecido al hecho de los "bombones pesados", es casi idéntico, ya veremos en qué se diferencian más adelante:

   - El problema **suficientemente simple** es 1 voto, por lo que habría mayoría absoluta, seria $V[1]$.
   - Según el enunciado para tener mayoría absoluta en N es condición necesaria pero no suficiente que tenga mayoría absoluta en alguna de las mitades del vector (recordemos ese ejercicio, es igual el planteamiento).
     La descomposición será, por tanto:



     De nuevo, al igual que pasaba antes se pueden dar 4 casos distintos. En todo caso, necesitaremos saber si hay alguien con mayoría absoluta, qué candidato es y además con cuántos votos. Veremos estos casos posibles:

     1. **No** hay mayoría absoluta en ninguna mitad, por tanto, no hay mayoría absoluta en el total (es condición del enunciado).
     2. Existe un **candidato1 con mayoría absoluta** en la mitad *izquierda* y no hay nadie con mayoría en la mitad derecha (condición necesaria pero no suficiente para que sea mayoría absoluta en el vector). Para ello, hay que hacer lo siguiente:

- Contar los votos de candidato1 en la mitad derecha y sumarlos a los que tiene en la izquierda. Tendremos que averiguar si num_votos1 $\geq \lfloor N/2 \rfloor + 1$.

3. Existe un **candidato2 con mayoría absoluta** en la mitad *derecha* y no hay nadie con mayoría en la mitad derecha (condición necesaria pero no suficiente para que sea mayoría absoluta en el vector). Para ello, hay que hacer lo siguiente:
   - Contar los votos de candidato1 en la mitad izquierda y sumarlos a los que tiene en la derecha. Tendremos que averiguar si num_votos2 $\geq \lfloor N/2 \rfloor + 1$.
4. El **candidato1 tiene mayoría absoluta en la derecha y el candidato2 lo tiene a la izquierda**, por lo que habría que ver cuál de los dos es el que gana (o ninguno).

2. Algoritmo completo a partir del refinamiento del esquema general

El esquema siguiendo los casos antes expuestos es:

```
fun pesado (v[1..n] de Tbombon, i, j: natural)
                                    dev (tiene_mayoria, candidato, num_votos)
    si (i = j)entonces
        dev (cierto, v[i], 1)
    si no
        /* Descomponer */
        (tiene_mayoria1, candidato1, num_votos1)=contar (V, i, (i + j) ÷ 2)
        (tiene_mayoria2, candidato2, num_votos2)=contar (V, (i + j) ÷ 2 + 1, j)
        /* Combinar */
        si tiene_mayoria1 entonces                    // Si mayoría en parte izq.
            para k desde (i + j) ÷ 2 + 1 hasta j hacer    // Cuenta en parte der.
                si es_igual (v[k], candidato1) entonces
                    num_votos1 = num_votos1 + 1
                fsi
            fpara
            si (num_votos1 > (j − i + 1) ÷ 2) entonces
                devolver (cierto, candidato1, num_votos1)
            fsi
        fsi
        si tiene_mayoria2 entonces                    // Si mayoría en parte der.
            para k desde 1 hasta (i + j) ÷ 2 hacer        // Cuenta en parte izq.
                si es_igual (v[k], candidato2) entonces
                    num_votos2 = num_votos2 + 1
                fsi
            fpara
            si (num_votos2 > (j − i + 1) ÷ 2) entonces
                devolver (cierto, candidato2, num_votos2)
            fsi
        fsi
        devolver (falso, " ", 0)
    fsi
ffun
```

La **llamada inicial** es contar (V, 1, N).

NOTA DEL AUTOR: Como hemos comentado previamente tenemos una serie de características propias a este código, aunque en el fondo sigue siendo igual al del "bombón pesado" y, por extensión, a los ejercicios de este tipo, que se harían casi iguales. En este caso, hemos añadido una nueva variable que cuenta el número de votos, que como antes vimos lo hacía la función auxiliar *contar*.

3. Estudio del coste del algoritmo desarrollado

Plantearemos la siguiente ecuación de recurrencia:

$$T(n) = 2 * T(n/2) + \frac{n}{2} + \frac{n}{2} = 2 * T(n/2) + n$$

Está ecuación se deduce de esta manera, ya que recorre ambas mitades al contar los distintos candidatos. Es por ello, que al final acaba **recorriendo todo el vector** encontrando al candidato con mayoría absoluta.

Por tanto, la reducción del problema se realiza mediante **división**, cuyos casos son los siguientes:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < b^k \\ \theta\left(n^k * log(n)\right) & \text{si } a = b^k \\ \theta\left(n^{\log_b a}\right) & \text{si } a > b^k \end{cases}$$

Las distintas variables son:

a: Número de llamadas recursivas = 2
b: Reducción del problema en cada llamada = 2
$\boldsymbol{c * n^k}$: Coste de las operaciones extras a las llamadas recursivas. Tendremos que el valor de $k = 1$, al ser el tiempo extra lineal con respecto a n.

Por ello y de nuevo, es más de lo mismo, tendremos que el coste es $T(n) = \theta(n * \log(n))$.

3ª parte. Problemas de exámenes sin solución o planteados:

Febrero 1999-2ª (problema 2)

Enunciado: Una liga de n equipos e_i juega a un juego donde solo se pierde o se gana, pero no se empata. Diseñar un algoritmo que realice una *pseudo-ordenacion* $(e_1, e_2, \dots, e_n)$ a partir de la tabla de resultados de los equipos de manera que e_1 haya ganado a e_2, e_2 a e_3, etc. El coste debe ser a lo sumo $O(n * \log(n))$

Respuesta: Este ejercicio no sé muy bien como se haría, entiendo que al ser una pseudo-ordenacion se debería plantear un esquema de divide y vencerás, pero estoy con la duda con respecto a los voraces, debido a que el coste puede ser también correspondiente a una exploración de grafos (recordemos algoritmo de Prim, Kruskal, …). Esto último al escribir el enunciado en el editor de texto me he dado cuenta que podría llegar a ser eso. Para concluir y en resumen decir que no estoy segura personalmente ni del esquema, aunque lo pondremos en divide y vencerás (que es lo que más se asemeja).

Diciembre 2002 (problema)

Enunciado: Se tienen dos polinomios de grado n representados por dos vectores $[0..n]$ de enteros, siendo la posición i de ambos vectores la correspondiente al término x^i del polinomio. Se pide diseñar un algoritmo que multiplique ambos polinomios, valorándose especialmente que sea con un coste más eficiente que $O(n^2)$ (En concreto, hay una solución fácil de hallar con coste $O\left(n^{\log_2 3}\right)$).

Respuesta: Este ejercicio se asemeja bastante al ejercicio 2 de Septiembre de 2007, en el que pedían exactamente lo mismo e igualmente lo hemos visto en la teoría, recordando, por tanto, que para tener menos coste resolvemos el problema usando 3 multiplicaciones en vez de 4. Lo único que se ha escrito este enunciado, ya que la solución más exhaustiva (la que vemos en los problemas) no la daremos, aunque sí que lo plantearemos adecuadamente.

Ejercicios resueltos de programación 3

Tema 9. Exploración de grafos.

Alumna: Alicia Sánchez
Centro: UNED-Las Rozas (Madrid)

Estos ejercicios son bastante extensos, por lo que lo separaremos en dos apartados distintos, que serán todos los ejercicios correspondientes a las búsquedas en anchura, profundidad y vuelta atrás y aparte debido a la importancia que tienen los de ramificación y poda. Decir que las cuestiones de esta segunda parte no son muy importantes, teniendo pocos de ellos, no obstante los distinguiremos. Por tanto, el *índice* de este tema será:

Búsqueda en anchura, profundidad y vuelta atrás

Introducción teórica:

Antes de resolver las cuestiones y preguntas vamos a dar unas nociones básicas de teoría, que en este caso pondremos los esquemas correspondientes a este apartado (recordemos que es recorrido en profundidad, anchura y vuelta atrás, siendo este último el más importante). En todo caso, como en ejercicios anteriores si hace falta especificar algún concepto o esquema se hará en el propio ejercicio. Pasamos a verlos:

- Recorrido en profundidad:

  Para el **recorrido en profundidad** se siguen estos pasos:

  - Se selecciona cualquier nodo $v \in N$ como punto de partida.
  - Se marca este nodo para mostrar que ya ha sido visitado.
  - Si hay un nodo adyacente a v que no haya sido visitado todavía, se toma este nodo como punto de partida y se invoca recursivamente al procedimiento en profundidad. Al volver de la llamada recursiva, si hay otro nodo adyacente a v que no haya sido visitado se toma este nodo como punto de partida siguiente, se llama recursivamente al procedimiento y, así sucesivamente.
  - Cuando están marcados todos los nodos adyacentes a v el recorrido que comenzó en v ha finalizado. Si queda algún nodo de G que no haya sido visitado tomamos cualquiera de ellos como nuevo punto de partida y (como en los grafos no conexos), volvemos a invocar al procedimiento. Se sigue así hasta que estén marcados todos los nodos de G.

  El procedimiento de **inicialización y arranque** será:

  ```
  procedimiento recorridop (G)
      para cada v ∈ N hacer marca[v] ← no visitado
      para cada v ∈ N hacer
          si marca[v] ≠ visitado entonces rp (v)
  ```

  El algoritmo de recorrido en profundidad siguiendo los pasos anteriores es:

  ```
  procedimiento rp (v)
      { El nodo v no ha sido visitado anteriormente }
      marca[v] ← visitado
      para cada nodo w adyacente a v hacer
          si marca[v] ≠ visitado entonces rp (v)
  ```

  Sea **pila** un tipo de datos que admite dos valores *apilar* y *desapilar*. Se pretende que este tipo represente una lista de elementos que hay que manejar por el orden "primero en llegar, primero en salir". La función *cima* denota el elemento que se encuentra en la parte superior de la pila.

El algoritmo de recorrido en profundidad ya modificado es:

```
procedimiento rp2 (v)
    P ← pila-vacía
    apilar w en P
    mientras P no esté vacía hacer
        mientras exista un nodo w adyacente a cima (P)
            tal que marca [w] ≠ visitado hacer
            marca [w] ← visitado
            apilar w en P              { w es la nueva cima (P) }
    desapilar P
```

- Recorrido en anchura:

En cuanto al **recorrido en anchura** seguiremos estos pasos:

- Se toma cualquier nodo $v \in N$ como punto de partida.
- Se marca este nodo como visitado.
- Después se visita a todos los adyacentes antes de seguir con nodos más profundos.

El procedimiento de **inicialización y arranque** será:

```
procedimiento recorrido (G)
    para cada v ∈ N hacer marca [v] ← no visitado
    para cada v ∈ N hacer
        si marca [v] ≠ visitado entonces {rp2 o ra} (v)
```

Para el algoritmo de recorrido en anchura necesitamos un tipo **cola** que admite las dos operaciones *poner* o *quitar*. Este tipo representa una lista de elementos que hay que manejar por el orden "primero en llegar, primero en salir". La función *primero* denota el elemento que ocupa la primera posición en la cola.

El recorrido en anchura no es naturalmente recursivo, por lo que el algoritmo será:

```
procedimiento ra (v)
    Q ← cola-vacía
    poner v en Q
    mientras Q no esté vacía hacer
        v ← primero (Q)
        quitar u de Q
        para cada nodo w adyacente a u hacer
            si marca [w] ≠ visitado entonces
            marca [w] ← visitado
            poner w en Q
```

- Vuelta atrás:

  Hay problemas que son inabordables mediante grafos abstractos (almacenados en memoria). Si el grafo contiene un número elevado de nodos y es infinito, puede resultar inútil construirlo implícitamente en memoria. En tales situaciones emplearemos un **grafo implícito**, que será aquél para el cual se dispone de una descripción de sus nodos y aristas, de tal manera que se pueden construir partes relevantes del grafo a medida que progresa el recorrido.

  En su forma básica, la vuelta atrás se asemeja a un recorrido en profundidad dentro de un grafo dirigido. Esto se consigue construyendo soluciones parciales a medida que progresa el recorrido; estas soluciones parciales limitan las regiones en las que se puede encontrar una solución completa. Se nos darán estos casos:

  - El recorrido **tendrá éxito** si se puede definir por completo una solución. En este caso, el algoritmo puede o detenerse (si sólo necesita una solución al problema) o bien seguir buscando soluciones alternativas (si deseamos examinarlas todas).
  - Por otra parte, el recorrido **no tiene éxito** si en alguna etapa de la solución parcial construida hasta el momento no se puede completar, lo cual denominaremos condición de poda (no se construye esa parte del árbol). Cuando vuelve a un nodo que tiene uno o más vecinos sin explorar, prosigue el recorrido de una solución.

  Este **primer esquema** de vuelta atrás es el general, en nuestro caso, será el básico que tengamos que saber para la asignatura:

```
fun vuelta-atrás (ensayo)
    si valido (ensayo) entonces
        devolver ensayo
    si no
        para cada hijo en compleciones (ensayo) hacer
            si condiciones-de-poda (hijo) entonces
                vuelta-atrás (hijo)
            fsi
        fpara
    fsi
ffun
```

  Necesitaremos especificar lo siguiente:

  1. **Ensayo**: Es el nodo del árbol.
  2. **Función valido**: Determina si un nodo es solución al problema o no.
  3. **Función compleciones**: Genera los hijos de un nodo dado.
  4. **Función condiciones-de-poda**: Verifica si se puede descartar de antemano una rama del árbol, aplicando los criterios de poda sobre el nodo origen de esa rama. Adoptaremos el convenio de que la función condiciones-de-poda devuelve *cierto* si ha de explorarse el nodo, y *falso* si puede abandonarse.

Para el problema de las ocho reinas (y otros problemas) a tener en cuenta, esta técnica tiene dos **ventajas** con respecto a las anteriores:

1. El número de nodos del árbol es menor que $8! = 40.320$. Bastaría con explorar 114 nodos para obtener la primera solución.
2. Para decidir si un vector es k-prometedor, sabiendo que es una extensión de un vector (k-1)-prometedor, sólo necesitamos comprobar la última reina que haya que añadir.

Los algoritmos de vuelta atrás se pueden utilizar aun cuando las soluciones buscadas no tengan todas necesariamente la misma longitud. Siguiendo con el planteamiento anterior de los k-prometedores tendremos este **cuarto esquema**, que será:

```
fun vueltaatrás (v[1..k])
    { v es un vector k-prometedor }
    si v es una solución entonces escribir v
    si no
        para cada vector (k+1)-prometedor w
            tal que w[1..k] = v[1..k] hacer
                vueltaatrás (w[1..k + 1])
```

1ª parte. Cuestiones de exámenes:

Febrero 2000-2ª (ejercicio 3)

Enunciado: En una exploración de un árbol de juego con dos oponentes a nivel de profundidad 4 (contando la raíz como nivel 1) y siguiendo la estrategia MINIMAX nos encontramos con la siguiente puntuación:

$$\Big[[[-7{,}5][-3][-10,-20,0]][[-5,-10][-15,20]][[1][6,-8,14][-30,0][-8,-9]]\Big]$$

donde los corchetes indican agrupación de nodos por nodo padre común. Se pide propagar las puntuaciones y elegir la rama más adecuada a partir de la raíz sabiendo que el nodo raíz corresponde al jugador A, a mayor valor, mejor jugada para A, y que se alternan en mover. ¿En qué se diferencia esta estrategia de juego de la utilizada para resolver el problema?

Respuesta:

NOTA DEL AUTOR: Este algoritmo no lo hemos visto en la sección de teoría, por lo que no indagaremos más sobre el mismo. Lo dejaremos como curiosidad, aunque este mismo algoritmo se dará en otras asignaturas de la UNED de informática de sistemas tales como Introducción a la Inteligencia Artificial.

Ahora veremos la respuesta dada para este ejercicio. El nodo raíz corresponde al jugador A, el nodo 2 al B, el nodo 3 al A y consecuentemente, el nodo 4 al B, según lo cual los valores corresponderán, por tanto, a una jugada de B. Como anteriormente juega A y los mejores valores para A son los mayores, cogerá, por tanto, los mayores valores de las jugadas posibles. Cuando juegue B es lógico que éste tome aquellas jugadas que sean peores para A, luego cogerá los valores más pequeños.

Dibujaremos el árbol (de forma invertida) para verlo más claro y realizar la propagación:

Juega
B $\min\ [-7{,}5][-3][-10,-20,0][-5,-10][-15,20][1][6,-8,14][-30,0][-8,-9]$
A $\max\ [5,-3,0][-5,20][1,14,0,-8]$
B $\min\ [-3,-5,8]$
A $\max\ [-3]$

Septiembre 2000 (ejercicio 1)

Enunciado: A la hora de hacer una exploración ciega en un grafo, ¿qué criterios nos pueden decidir por una búsqueda en profundidad y una en anchura?

Respuesta: A la hora de escoger un criterio u otro, fundamentaremos nuestra decisión según el problema en cuestión y el tamaño del grado generado. Tendremos estos casos:

- Si el problema consiste en encontrar el camino más corto desde un punto del grafo hasta otro, la **exploración en anchura** es la más adecuada.
- Del mismo modo, si el grafo a tratar es de un tamaño muy grande, de forma que no resulte manejable, o infinito, utilizaremos una **exploración en anchura** para realizar una exploración parcial de dicho grafo.
- En otro caso, nos decidiremos por la **exploración en profundidad**.

Febrero 2001-1ª (ejercicio 2)

Enunciado: Suponemos que para el juego del ajedrez hemos programado una función estática perfecta de evaluación eval (u) para cada situación del juego u. Se supone conocida otra función, compleciones (u) que devuelve la lista de las jugadas legales a partir de la posición del tablero u. Explica, incluyendo pseudocódigo si es preciso, como programar un algoritmo que juegue al ajedrez, ¿qué pasa si (como ocurre en la realidad) la función eval (u) no existe? ¿Qué papel desempeña entonces la estrategia MINIMAX?

Respuesta: Gracias a la información perfecta suministrada por la función eval (u) podremos saber en cada momento cuál es la mejor jugada posible. De este modo, si jugaran, por ejemplo, las blancas dado un estado u, tomaríamos como mejor movimiento el que maximice el valor de eval (u) (suponiendo que éste nos devuelve el valor más grande cuanto mejor sea la jugada para las blancas), sin importarnos qué decisión puede tomar las negras tras nuestro movimiento, ya que eval (u) nos asegura que nuestro movimiento es el más adecuado.

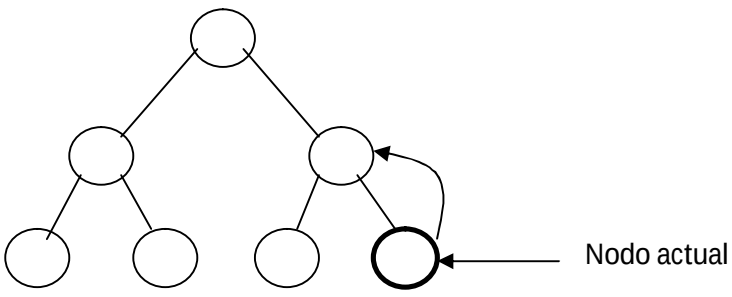
Sin embargo, no existe una función eval (u) perfecta en el caso del ajedrez, deberemos entonces buscar una función eval (u) aproximada con una relación coste/precisión lo mejor posible. En este caso, eval (u) no nos confirma que el mejor movimiento considerado por ella sea en realidad el más adecuado. Y es aquí donde entre la estrategia MINIMAX, considerando que si para las blancas será la mejor jugada la marcada como mayor valor de eval (u), las negras ejecutarán aquella que minimice dicho valor. De este modo, si queremos anticipar un número determinado de jugadas, tomaremos el anterior criterio, minimizar el valor de eval (u) para las negras y maximizarlas para las blancas.

NOTA DEL AUTOR: Este ejercicio se ha copiado literalmente de la respuesta dada (no se sabe si es oficial la solución), aunque como en el anterior es una parte del temario que no hemos resumido y en ninguna otra pregunta posterior hace referencia, como ocurría con el ejercicio anterior.

Septiembre 2001-reserva (ejercicio 3) (igual que el ejercicio 2 de Febrero 2002-1ª semana)

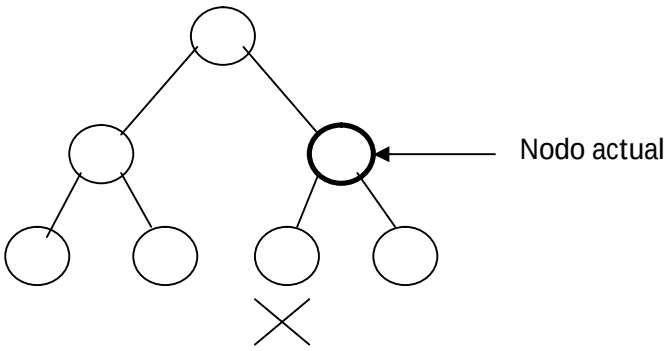
Enunciado: En los algoritmos de vuelta atrás explica la diferencia entre una condición de poda y una condición de retroceso.

Respuesta: El algoritmo de vuelta atrás básico consiste en una búsqueda exhaustiva en el árbol, si el recorrido del árbol no tiene éxito porque la solución parcial hasta ese momento no puede ser completada se produce un retroceso (**condición de retroceso**), igualmente si ya existiera una solución mejor a la que está siendo actualmente calculado volveríamos atrás. Podríamos decir que una condición de retroceso es aquella en la que no podemos seguir explorando o bien porque no hay más ramas para explorar o bien porque no podemos explorar más. Gráficamente, sería algo así:



Vemos en el grafico que no hay modo de continuar por ninguna otra rama, por lo que se iría al nivel superior, siendo éstas las condiciones de retroceso.

Si queremos limitar más aun nuestro espacio de búsqueda podemos utilizar **condiciones de poda**, dado un determinado problema intentamos encontrar información que nos permita detener nuestra búsqueda (volviendo consecuentemente atrás) si hay claros indicios de que el estado actual no nos conducirá a una solución. Una condición de poda verifica si se puede descartar de antemano una rama del árbol, aplicando los criterios de poda sobre el nodo origen de esa rama. Gráficamente, sería algo así:



Marcamos con una cruz la rama del árbol por donde no se puede continuar debido a las condiciones de poda.

NOTA DEL AUTOR: Ambos gráficos son añadidos del autor, por lo que no se asegura que esté bien. Lo he visto en varios libros y pienso que debe ser así.

Febrero 2004-1ª (ejercicio 2)

Enunciado: Se tienen 4 productos infraccionables p1, p2, p3, p4 en cantidades ilimitadas cuyo beneficio es respectivamente 23, 12, 21, 11 € y cuyos pesos son respectivamente 2, 4, 3, 5 kgs. Tenemos un camión que carga un máximo de 55 kgs. El problema consiste en llenar un camión con la carga de mayor valor. ¿Qué algoritmo nos permite resolver el problema? Aplicarlo al enunciado y detallarlo paso por paso.

Respuesta: En este problema se nos dan 4 productos infraccionables, de tal manera que hay que llenar un camión maximizando el valor de los productos con carga máxima de 55 kgs. Tendremos, por tanto, dos posibles maneras de resolverlo:

- **Esquema voraz**: **No** podremos aplicarla, ya que no existe ninguna función de selección que garantice la solución óptima, por ser infraccionables.
- **Vuelta atrás**: Podremos emplear este esquema para resolver el problema, justamente por la misma razón que descartamos el anterior. Para ello, emplearemos una exploración de grafos.

Este ejercicio se ha resuelto íntegramente por un alumno, por lo que seguramente falten algunas razones por las que puede ser vuelta atrás, aunque no lo sabría explicar adecuadamente.

Con respecto a la segunda pregunta se nos pide que apliquemos el enunciado y lo detallemos paso a paso. Como vimos antes, realizaremos una exploración de grafos, empleando para ello vuelta atrás, de tal modo que al expandir el grafo implícito si en algún nivel se sobrepasa del peso máximo se poda la rama.

Vemos, además, que el máximo de peso del camión es de 55 kgs y que el peso máximo de los productos es de 4 kgs, esto quiere decir que tendríamos que hacer un mínimo de 12 niveles del

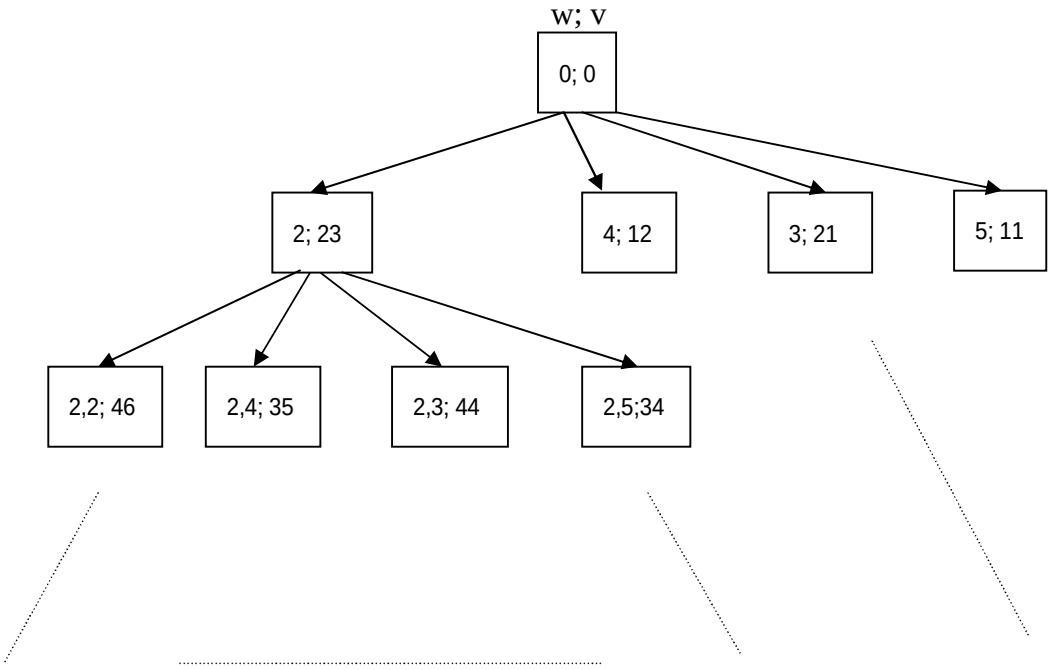
árbol, por lo que sólo lo haremos de un par de niveles. Se puede ver en el resumen de este tema un ejercicio similar a este (problema de la mochila), dando resultados posibles y podando ramas. Por tanto, para completar este ejercicio, se puede ver dicho árbol.

Del enunciado tenemos:

i	1	2	3	4
v_i	23	12	21	11
w_i	2	4	3	5

PMAX = 55 kgs.

El grafo entonces sería el siguiente:



Febrero 2004-2ª (ejercicio 3)

Enunciado: ¿Cuándo resulta más apropiado utilizar una exploración en anchura que en profundidad? ¿En qué casos puede que la exploración en anchura no encuentre una solución aunque ésta exista? ¿Y la exploración en profundidad? Razona tus respuestas

Respuesta:

En cuanto a la *primera pregunta* existen tres **situaciones** en las que resulta más adecuado utilizar una exploración en anchura que en profundidad:

- Cuando el grafo tiene ramas infinitas.
- Cuando se busca el camino de menor número de nodos o aristas.
- Cuando los nodos finales se encuentran cerca del nodo raíz (creo que es igual que la segunda situación, pero en la solución la separan).

De la *segunda pregunta* tenemos que:

- Que exista una solución quiere decir que existe un camino desde el nodo inicial al final. No tiene sentido, entonces, hablar de que la "solución" esté en una componente conexa distinta. Por otro lado, el caso en que un nodo genere **infinitos hijos** (no se suele dar en la práctica) es un caso excepcional que no debería darse. Por todo ello y salvo esta última excepción, la exploración en anchura siempre encuentra solución, si existe.

De la *última pregunta* tendremos un caso en el que no encuentra solución la exploración en profundidad es si existen **ramas infinitas**, quedándose atrapada en esa rama sin encontrar la solución.

Febrero 2005-1ª (ejercicio 3)

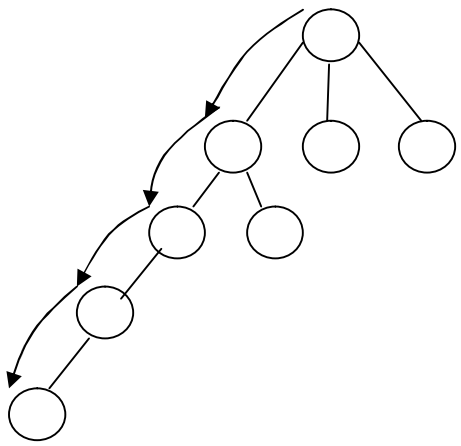
Enunciado: Explique las diferencias entre un recorrido en anchura y un recorrido en profundidad. Exponga un algoritmo iterativo para cada uno de los recorridos explicando las estructuras de datos asociadas, así como su coste.

Respuesta:

Para el **recorrido en profundidad** se siguen estos pasos:

- Se selecciona cualquier nodo $v \in N$ como punto de partida.
- Se marca este nodo para mostrar que ya ha sido visitado.
- Si hay un nodo adyacente a v que no haya sido visitado todavía, se toma este nodo como punto de partida y se invoca recursivamente al procedimiento en profundidad. Al volver de la llamada recursiva, si hay otro nodo adyacente a v que no haya sido visitado se toma este nodo como punto de partida siguiente, se llama recursivamente al procedimiento y, así sucesivamente.
- Cuando están marcados todos los nodos adyacentes a v el recorrido que comenzó en v ha finalizado. Si queda algún nodo de G que no haya sido visitado tomamos cualquiera de ellos como nuevo punto de partida y (como en los grafos no conexos), volvemos a invocar al procedimiento. Se sigue así hasta que estén marcados todos los nodos de G.

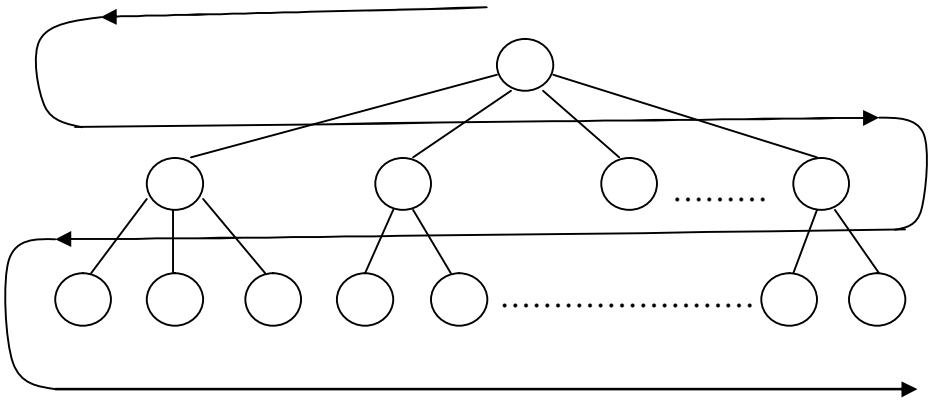
Gráficamente, será algo así:



En cuanto al **recorrido en anchura** seguiremos estos pasos:

- Se toma cualquier nodo $v \in N$ como punto de partida.
- Se marca este nodo como visitado.
- Después se visita a todos los adyacentes antes de seguir con nodos más profundos.

Gráficamente, sería algo así:



La diferencia más significativa entre ambos recorridos es que se usan *pilas* en la **búsqueda en profundidad** y *colas* en la **búsqueda en anchura**.

La *pila* es una estructura de datos que maneja los elementos por el orden "primero en entrar, último en salir". En la cima estará el elemento en la parte superior de la pila. Las operaciones que usaremos serán la de *apilar* y *desapilar*. Tendremos este esquema para la búsqueda (o recorrido, es similar) en profundidad:

```
procedimiento rp2 (v)
    P ← pila-vacía
    apilar w en P
    mientras P no esté vacía hacer
        mientras exista un nodo w adyacente a cima (P)
        tal que marca [w] ≠ visitado hacer
        marca [w] ← visitado
        apilar w en P              { w es la nueva cima (P) }
    desapilar P
```

Como nota del autor decir que hay otro algoritmo de recorrido en profundidad, pero que no usa las estructuras de datos *pila*, por lo que ponemos este último por ser más completo.

Con referencia a la *cola* decir que es una estructura de datos que maneja los elementos por el orden "primero en entrar, primero en salir". La función *primero* denotará el elemento que ocupa la primera posición de la *cola*. El esquema de búsqueda en anchura será el siguiente:

```
procedimiento ra (v)
    Q ← cola-vacía
    poner v en Q
    mientras Q no esté vacía hacer
        v ← primero (Q)
        quitar u de Q
        para cada nodo w adyacente a u hacer
            si marca [w] ≠ visitado entonces marca [w] ← visitado
            poner w en Q
```

Estas son las estructuras de datos que usaremos. En cuanto al coste están ambas en $\boldsymbol{\theta(max(a, n))}$, siendo *a* el número de aristas y *n* el número de nodos.

Febrero 2005-2ª (ejercicio 2)

Enunciado: ¿En qué se diferencia una búsqueda ciega en profundidad y un esquema de vuelta atrás? Pon un ejemplo.

Respuesta: El **esquema de vuelta atrás** es una búsqueda en profundidad, pero en la que se articula un mecanismo para detener la búsqueda en una determinada rama (poda). Para alcanzar una solución final es necesario que los pasos intermedios sean soluciones parciales al problema. Si un determinado nodo no es una solución parcial, entonces la solución final no se puede alcanzar a partir de dicho nodo y la rama se poda. Esto se implementa en el esquema de vuelta atrás mediante la función condiciones-de-poda o función de factibilidad (siempre lo he visto como la primera función).

En el caso de **la búsqueda ciega en profundidad** no habrá esta función de condiciones-de-poda, por lo que no hay restricciones posibles a la hora de realizar la búsqueda. Se podría decir que es la diferencia más importante.

El problema de colocar N reinas en un tablero de N x N sin que se amenacen entre sí un problema que se puede resolver mediante vuelta atrás, puesto que en cada nivel de la búsqueda se establece un problema parcial: en el nivel i se trata de colocar i reinas en un tablero de N x N sin que se amenacen entre sí. Si un nodo no cumple está condición, por muchas más reinas que se conocen a continuación nunca se va a encontrar una solución final. Recordemos además que se usaban vectores k-prometedores para resolverlo.

Septiembre 2005 (ejercicio 2)

Enunciado: En el contexto de elegir un esquema algorítmico para resolver un problema de optimización con restricciones, ¿cuándo se puede resolver mediante un esquema voraz y en qué casos sería necesario utilizar un esquema de ramificación y poda?

Respuesta: Para resolverlo con un **esquema voraz** es necesario que exista una función de selección de candidatos y una función de factibilidad que decida si se acepta o rechaza el candidato, de manera que la decisión es irreversible. Si no es posible encontrar ninguna de las funciones anteriores entonces optaríamos por otro esquema como el de **ramificación y poda**, que este último lo escogeremos por ser problema de optimización.

Febrero 2006-1ª (ejercicio 1)

Enunciado: ¿En qué se diferencia una búsqueda ciega en profundidad y un esquema de vuelta atrás? En el espacio de búsqueda, ¿qué significa que un nodo sea k-prometedor? ¿Qué hay que hacer para decidir si un vector es k-prometedor sabiendo que extensión de un vector (k-1)-prometedor?

Respuesta:

En cuanto a la primera pregunta, la **búsqueda ciega** explora todas las ramas alternativas mientras que en un esquema de **vuelta atrás** se establecen condiciones de poda que determinan si una rama puede alcanzar o no una solución final. Si se determina que no es posible, entonces no se prosigue la búsqueda por dicha rama (se poda).

Los problemas aptos para un esquema de vuelta atrás permiten expresar los nodos intermedios como soluciones parciales al problema. Aquellos nodos que no sean soluciones parciales no permiten alcanzar una solución final y, por tanto, su rama se poda. Por ejemplo, el problema de "poner N reinas en un tablero de N x N sin que se amenacen entre sí" requiere ir solucionando la siguiente secuencia de soluciones parciales:

Poner 1 reina en un tablero de N x N sin que esté amenazada (trivial).
Poner 2 reinas en un tablero de N x N sin que se amenacen entre sí.
...
Poner k reinas en un tablero de N x N sin que se amenacen entre sí.
...
Poner N reinas en un tablero de N x N sin que se amenacen entre sí.

Si por una rama no se puede resolver el problema para k, entonces evidentemente no se podrá resolver para N, por muchos intentos que hagamos de añadir reinas.

Con respecto a la segunda pregunta, un nodo k-prometedor significa que es solución parcial para las k primeras componentes de la solución y, por tanto, todavía es posible encontrar una solución final (incluyendo las N componentes). En el problema de las N reinas habríamos colocado k reinas sin que se amenacen entre sí.

Por último, en cuanto a la tercera pregunta si es (k-1)-prometedor quiere decir que es una solución parcial para las primeras k – 1 componentes y que, por tanto, éstas cumplen las restricciones necesarias entre sí y no hay que volver a verificarlas. Entonces, para decidir si una extensión considerando la siguiente componente k conforma un nodo *k-prometedor*, lo único que hay que hacer es verificar si esta nueva componente k cumple las restricciones respecto a las otras k – 1.

NOTA DEL AUTOR: Esta última pregunta es importante tenerla en cuenta y completarla con la teoría, por lo que se tiene que hacer especial hincapié al estudiar esta pregunta. Además, entiendo que es fácil el comprenderlo.

2ª parte. Problemas de exámenes solucionados:

Tendremos en cuenta los pasos a seguir para resolver problemas:

- Elección del esquema (voraz, vuelta atrás, divide y vencerás).
- Identificación del problema con el esquema.
- Estructura de datos.
- Algoritmo completo (con pseudocódigo).
- Estudio del coste.

Vamos a ver los problemas resueltos de exploración en profundidad, anchura y vuelta atrás, siendo estos últimos los más importantes y más comúnmente puestos en ejercicios de exámenes. Separaremos, por tanto, los esquemas de ramificación y poda.

Representaremos las funciones en pseudocódigo empleando cualquiera de las dos técnicas en los parámetros: escribiendo como una variable de la estructura de datos (por ejemplo, e:ensayo) o bien como la propia estructura de datos (por ejemplo, ensayo). Se verán ejemplos más adelante donde usaremos ambas técnicas (o casos) de modo indiferente, ya que la solución es siempre la misma, siendo, por tanto, su uso indiferente.

Otro asunto que es importante destacarlo es que no podremos punto y coma (;) en todas las sentencias, aunque lo ideal es que se ponga. Es importante saberlo, ya que nuestro propósito es didáctico y siendo estrictos habría que ponerlo siempre.

Febrero 1996-2ª (problema 1) (igual a 4.2 libro de problemas resueltos)

Enunciado: Las 28 dichas del dominó son de la forma $\{i, j\}$ con $i, j = 1 \dots 6$. Una ficha de dominó puede colocarse a continuación de la anterior si coinciden los valores de los extremos que se tocan. Por ejemplo, a continuación de la ficha (1,2) puede colocarse (2,4). Diseñar un algoritmo que produzca todas las cadenas permisibles que contengan todas las filas del dominó.

Respuesta:

1. Elección razonada del esquema algorítmico

   Se justifica la técnica de algoritmo por **retroceso o backtracking** porque:

   - No hay criterio de elección de la pieza a colocar que conduzca con certeza a la solución correcta (búsqueda ciega).
   - Se pide dar todas las soluciones permitidas (no es voraz).
   - No es descomponible en subproblemas idénticos cuyas soluciones pueden combinarse para dar la solución global (no es divide y vencerás).

2. Descripción del esquema usado e identificación con el problema

El esquema general de los algoritmos de retroceso o vuelta atrás es:

```
fun vuelta-atrás (ensayo)
    si valido (ensayo) entonces
        devolver ensayo
    si no
        para cada hijo en compleciones (ensayo) hacer
            si condiciones-de-poda (hijo) entonces
                vuelta-atrás (hijo)
            fsi
        fpara
    fsi
ffun
```

En nuestro caso, las funciones que tendremos que especificar es:

- **Válido**: Cuando se hayan colocado todas las piezas será solución.
- **Compleciones**: Se crea un nuevo ensayo por cada pieza que se pueda añadir en ese momento.
- **Condiciones de poda**: No son necesarias.

3. Estructuras de datos

Un juego (ensayo) va a ser una tupla compuesta de las siguientes estructuras:

- Una caja con las fichas del dominó.
- La cadena de fichas colocadas.
- El número de fichas colocadas, es decir, la longitud de la cadena.

Para implementar dicha tupla se pueden utilizar los siguientes tipos de datos:

- Caja de fichas: Es una matriz booleana simétrica. Si una ficha está en la caja, en la posición (i, j) y (j, i) habrá un valor *cierto*. La ficha (i, j) está en la caja si $caja[i, j] = cierto$.
- Cadena de fichas: Un vector con rango 1 … 28 que contenga fichas. Una ficha será un registro con dos valores numéricos i y j con rango 0 … 6.

La estructura quedaría así:

```
ficha = tupla
    i,j: arreglo[0..6] de entero
ftupla

juego = tupla
    caja: arreglo[0..6,0..6]de boolean
    cadena: arreglo[1..28] de ficha
    última: entero
ftupla
```

Hemos añadido la primera tupla, debido a que se puede sacar de la propia solución del ejercicio y así queda más claro.

4. Refinamiento del problema (algoritmo completo)

Una vez detalladas las estructuras de datos, el algoritmo se puede refinar instanciando el esquema planteado.

```
fun domino (juego)
    si valido (juego) entonces
        dev juego.cadena
    si no
        lista-c ← compleciones (juego)
        mientras no vacía (lista-c) hacer
            hijo ← primero (lista-c)
            lista-c ← resto (lista-c)
            si condiciones de poda (hijo) entonces
                domino (hijo)
            fsi
        fmientras
    fsi
ffun
```

siendo:

lista-c: lista de compleciones.

NOTA DEL AUTOR: Se ha modificado esta función, porque hacía una llamada recursiva a otra función llamada vuelta atrás, cuando observamos que se llama *domino*. Se podría haber modificado la función *domino* y llamarla *vuelta atrás* o bien la modificación hecha arriba. Pongo este detalle porque es otra errata más del libro de problemas.

Una vez instanciado el esquema, detallamos la función **compleciones**, que comprueba la ultima ficha colocada y saca de la caja todas las que pueden casar con ella. Por cada ficha crea un juego nuevo y lo añade a la lista de compleciones.
NO hay **condiciones de poda** en este problema, por lo que se elimina la referencia a esta función.

Por ello, un 'croquis' de la función compleciones será:

```
fun compleciones (juego) dev lista-c: lista
    lista-c ← lista vacía
    obtener última ficha colocada
    para cada ficha de la caja que case con ella hacer
        quitar ficha de la caja
        añadir ficha a la cadena
        crear un juego con los datos anteriores
        añadir juego a lista-c
    fpara
    devolver lista-c
ffun
```

Refinando un poco más obtenemos el algoritmo definitivo en pseudocódigo:

```
fun compleciones (juego) dev lista-c: lista
    lista-c ← lista vacía
    última-ficha ← juego.cadena[última]
    j ← ultima-ficha.j
    para i ← 0 hasta 6 hacer
        si caja[j,i] = cierto entonces
            juego-nuevo ← juego
            ficha ← crear-ficha (j, i)
            juego-nuevo.caja[i, j] ← falso
            juego-nuevo.caja[j, i] ← falso
            juego-nuevo.cadena[j + 1] ← ficha
            juego-nuevo.última ← juego-nuevo.última + 1
            lista-c ← añadir (juego, lista-c)
    fpara
    devolver lista-c
ffun
```

La función crear-ficha toma dos valores y crea una tupla del tipo indicado.

NOTA DEL AUTOR: No acabo de ver claro lo que significa la variable *última*, es decir, esa variable que 'aparece de la nada'. Conceptualmente (y personalmente) lo veo claro, que es tomar la última ficha colocada, pero implementado en pseudocódigo es donde tengo esa duda. Por tanto, creo que es otra errata más del libro de problemas, pero no estoy segura.

5. Estudio del coste

El coste del algoritmo de vuelta atrás depende del número de nodos recorridos. En sentido estricto, el coste del algoritmo no depende del tamaño del problema puesto a que éste es constante, pero sí que podemos estimar el coste analizando cómo es el árbol que recorremos.

Cada nodo tiene a lo sumo 6 ramas y además el último nodo tiene sólo una alternativa, al igual que el penúltimo, de manera que el coste se **puede estimar** muy por encima como de 6^{26}.

Septiembre 1996 (problema 2) (igual a 4.5 libro de problemas resueltos)

Enunciado: Se consideran las funciones $m(x) = 3 * x$ y $d(x) = x\ div\ 2$ (donde 'div' representa la división entera). Diseñar un algoritmo que, dados dos números *a* y *b*, encuentre una forma de llegar de *a* a *b* mediante aplicaciones sucesivas de $m(x)$ y $d(x)$. Por ejemplo, se puede pasar de 7 a 2 mediante

$$2 = d\left(d\left(m(d(7))\right)\right)$$

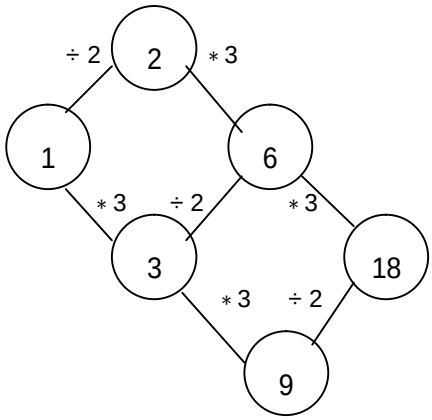
Respuesta:

1. Elección razonada del esquema algorítmico

   El esquema voraz no es aplicable, ya que no se trata de un problema de optimización. El esquema de divide y vencerás tampoco parece aplicable, ya que no parece que se pueda descomponer el problema en subproblemas idénticos y más sencillos. Sin embargo, sí podemos utilizar el esquema de **vuelta atrás**: cada rama del árbol estará formada por el número al que se llega a través de la aplicación de $m\ (x)$ o $d\ (x)$.

   El nodo inicial será el origen *a* y deberemos explorar el grafo hasta encontrar el número de llegada, *b*. sin embargo, habremos de tener cuidado al explorar este tipo de árbol, pues es **potencialmente infinito**; la vuelta atrás debe producirse cuando un camino no es prometedor, pero no puede esperarse a llegar a una hoja: en este árbol no hay hojas, es decir, no hay nodos terminales debido a que se aplica la división y multiplicación constantemente.

   El árbol de búsqueda será:



2. Descripción del esquema usado e identificación con el problema

   El **esquema general** de vuelta atrás es:

```
fun vuelta-atrás (ensayo)
    si valido (ensayo) entonces
        devolver ensayo
    si no
        para cada hijo en compleciones (ensayo) hacer
            si condiciones-de-poda (hijo) entonces
                vuelta-atrás (hijo)
            fsi
        fpara
    fsi
ffun
```

En nuestro caso, cada ensayo estará constituido fundamentalmente por una serie de aplicaciones de $m\ (x)$ y $d\ (x)$. Habrá, por tanto, dos compleciones posibles:

- Aplicar m (multiplicación).
- Aplicar d (división).

Tendremos que tener en cuenta lo siguiente:

- Para no caer en ciclos debemos anotar en una lista los **nodos ya visitados** y desechar una exploración que no nos lleve a alguno de ellos de nuevo. Esta variable debe ser global, ya que aunque afecta a la transparencia del programa su uso evita repetir búsquedas hechas en otras ramas.
- Debemos tener cuidado de **eliminar una rama** si se llega a cero.
- Hay que evitar las **búsquedas infinitas no cíclicas**. Para ello, situaremos el ensayo (pienso personalmente que es la compleción del ensayo) que consiste en dividir en el primer lugar de la lista de compleciones para cada nodo.

3. Estructuras de datos

Manejaremos enteros y listas de enteros (para tomar nota de los números ya visitados). Para recoger el camino podemos usar listas de caracteres, que contengan caracteres "d" y "m".

Un ensayo será, por tanto, una estructura con los siguientes registros:

Dos **enteros** que representan el número que nos encontramos y el número objetivo.
Una **lista de caracteres** con las funciones (o dividir, carácter "d" o multiplicar, carácter "m") que hemos aplicado hasta llegar a ese ensayo.

```
ensayo = tupla
    origen: entero                  { Valor inicial del árbol de búsqueda }
    destino: entero                 { Valor a alcanzar }
    camino: lista de caracteres     { Para luego recoger el camino realizado }
```

En nuestro ejemplo anterior hemos podríamos decir que el origen es el nodo con valor 2 puede ser el origen y el de valor 18 el destino. Trataremos de ver el camino que se ha recorrido hasta llegar a la misma, como hemos puesto en el enunciado.

4. Algoritmo completo a partir del refinamiento del esquema general

Para dar el algoritmo completo, especificamos cada una de las funciones sin definir en el esquema general (válido, compleciones y condiciones-de-poda). La primera de ellas (**válido**) es:

```
fun válido (e: ensayo) dev boolean
    si e.origen = e.destino entonces
        dev cierto
    si no
        dev falso
    fsi
ffun
```

En este caso en especial podremos reescribir esta función como sigue (pienso que además es más fácil en todos los sentidos), aunque la primera es la que viene en la solución del ejercicio. Se queda, por tanto, como añadido del autor:

```
fun válido (e: ensayo) dev boolean
    dev (e.origen = e.destino)
ffun
```

La función de **compleciones** será:

```
fun compleciones (e: ensayo) dev lista de ensayos
    lista-compleciones ← lista vacía
    { Compleción 1: multiplicación. Crea nueva estructura de datos }
    hijo1 ← e
    hijo1.origen ← e.origen * 3
    hijo1.camino ← añadir ("m", hijo1.camino)
    añadir (hijo1, lista-compleciones)
    añadir (hijo1.origen, nodos-visitados)

    { Compleción 2: división. Crea nueva estructura de datos }
    hijo1 ← e
    hijo1.origen ← e.origen div 2
    hijo1.camino ← añadir ("d", hijo1.camino)
    añadir (hijo1, lista-compleciones)
    añadir (hijo1.origen, nodos-visitados)

    dev lista-compleciones
ffun
```

Por último, la función **condiciones-de-poda** del ensayo será:

```
fun condiciones-de-poda (e: ensayo) dev boolean
    si e.origen en nodos-visitados V e.origen = 0 entonces
        dev falso
    si no
        dev cierto
    fsi
ffun
```

De nuevo, se puede reescribir esta función como hemos visto previamente. Esta función intentamos averiguar si exploramos un nodo si ya es visitado o el nodo origen es 0. En este caso no seguiremos por esa rama, en caso contrario sí.

NOTA: Se ha rehecho este ejercicio ya que el argumento sólo ponía ensayo y se ha añadido la variable *e* perteneciente a la tupla ensayo, como en la función válido. Así mismo, se ha añadido la variable que devuelve dicha función.

Necesitamos definir una función adicional que inicialice la **variable global** nodos-inicializados y el **primer ensayo** antes de llamar a la función recursiva vuelta-atrás:

```
fun conecta (a, b: natural) dev boolean
    e ← inicializar-ensayo
    e.origen ← a
    e.destino ← b
    e.camino ← lista-vacía
    nodos-visitados ← lista-vacia
ffun
```

Esta definición supone que a y b son mayores que cero.
La función añadir es una función estándar para añadir un elemento al final de una lista. Se obvia su implementación, debido a que es una operación propia de la estructura de datos lista.

5. Estudio del coste

   No es posible deducir de forma sencilla cual es el coste de este algoritmo, ya que el árbol es potencialmente infinito. Desconocemos igualmente el tamaño del problema. Este algoritmo halla una solución, pero no la más corta, de forma que sus resultados pueden ser arbitrariamente largos.

Problema 4.1 del libro de problemas resueltos (sin correspondencia con ningún ejercicio de examen)

Enunciado: Dado el conjunto de caracteres alfabéticos, se quieren generar todas las palabras de cuatro letras que cumplen las siguientes condiciones:

a) La primera letra debe ser vocal.
b) Sólo pueden aparecer dos vocales seguidas si son diferentes.
c) No pueden haber ni 3 vocales ni 3 consonantes seguidas.
d) Existe un conjunto C de parejas de consonantes que no pueden aparecer seguidos.

Respuesta:

1. Elección razonada del esquema algorítmico

   Se nos está pidiendo buscar en el conjunto formado por las palabras de cuatro letras, aquellas que cumplan determinadas condiciones. Se plantea dicha búsqueda de forma exhaustiva y se imponen algunas restricciones. Todo lo anterior encaja dentro de los algoritmos de **vuelta atrás**, ya que la búsqueda no se guía por ningún criterio y además no es posible descomponer el problema en subproblemas de sí mismo.

2. Descripción del esquema usado e identificación con el problema

   El esquema de **vuelta atrás o backtracking** es una técnica algorítmica para realizar búsquedas en grafos o árboles. Si en algún punto del recorrido se dan condiciones que hagan inútiles según el camino tomado, se retrocede a la última decisión y se sigue la siguiente opción.

El **esquema general** es:

```
fun vuelta-atrás (ensayo)
    si valido (ensayo) entonces
        devolver ensayo
    si no
        para cada hijo en compleciones (ensayo) hacer
            si condiciones-de-poda (hijo) entonces
                vuelta-atrás (hijo)
            fsi
        fpara
    fsi
ffun
```

En el caso que nos ocupa, un **ensayo** es una palabra de 4 letras, el cual será válido cuando se completen las 4 letras de la palabra formada. El árbol de búsqueda tiene como nodo raíz la palabra vacía (sin ningún carácter) y cada nodo tiene como hijos el resultado de añadir una letra, para cada una de las letras disponibles.

La función compleciones puede generar un nodo por cada letra del alfabeto y entonces la función condiciones-de-poda verificará que la palabra así generada cumple las condiciones del enunciado: que la primera letra sea vocal, que no haya 3 vocales seguidas, que no aparezca ninguna pareja de consonantes de entre la lista de parejas prohibidas, … Así pues, cada nodo se ramificará en un máximo de 28 y el árbol tendrá 4 niveles.

A continuación, se describen las estructuras de datos necesarias y se refina el algoritmo anterior.

3. Estructuras de datos

Para representar una **palabra (ensayo)** se utilizará un vector de 4 caracteres. Nos interesa añadir, además, un campo que indique el número de letras incorporadas para no tener que hacer una comprobación lineal cada vez que se quiera incorporar una letra. Por tanto, utilizaremos un registro con dos campos: **palabra e índice**.

Por tanto, la tupla se puede definir así:

```
ensayo = tupla
    palabra: arreglo [1..4] de a..z
    indice: entero
ftupla
```

4. Algoritmo completo a partir del refinamiento del esquema general

Una vez establecida la forma de representar las palabras pasamos a detallar el refinamiento del algoritmo dado anteriormente. Consideramos que la función **compleciones** devuelve una lista de palabras resultado de añadir una letra nueva a la palabra que se le pasa como argumento. Si la palabra está completa se considera válida y se devuelve como resultado.

La función compleciones sólo genera palabras posibles, es decir, que es necesario que posteriormente cumplan las restricciones impuestas en el enunciado. En el caso que nos ocupa tenemos como condiciones de poda las siguientes:

- Primera letra vocal.
- Dos vocales seguidas sólo si son distintas.
- Ni tres vocales ni consonantes seguidas.
- No pertenencia al conjunto de pares C.

La función **compleciones** puede escribirse así:

```
fun compleciones (e: ensayo) dev lista de ensayos
    lista-compleciones ← lista vacía
    para cada letra en {a, b, c, ... , z} hacer
        hijo ← añadir-letra (e, letra)
        lista-compleciones ← añadir (hijo, lista-compleciones)
    fpara
    dev lista-compleciones
ffun
```

NOTA DEL AUTOR: Se ha modificado la línea de añadir, porque en esta función el primer argumento es el nodo y el segundo la lista donde se añade el nodo. Simplemente es un detalle insignificante en este caso, pero lo suyo es hacerlo bien.

La función auxiliar *añadir-letra* es la única función de **modificación** de la estructura de datos ensayo que necesitamos, y puede definirse así:

```
fun añadir-letra (e: ensayo; l: letra) dev ensayo
    nuevo-ensayo ← e
    nuevo-ensayo.palabra ← añadir (l, nuevo-ensayo.palabra)
    nuevo-ensayo.indice ← nuevo-ensayo.indice + 1
    dev nuevo-ensayo
ffun
```

NOTA DEL AUTOR: En esta función hay determinadas cosas que no son muy comprensibles. La primera de ellas es que ahora la función añadir añade la letra en un vector de 4 elementos y sin embargo previamente se añadía en una lista. Por lo que mi conclusión personal es que se usa añadir para cualquier estructura de datos (lista, vector,...), lo cual no acabo de comprender. La otra cosa que no veo clara es en la línea siguiente:

```
nuevo-ensayo.indice ← nuevo-ensayo.indice + 1
```

Siguiendo el planteamiento anterior, estimo que debería ser como hemos visto en compleciones, algo así:

```
nuevo-ensayo.indice ← e.indice + 1
```

No se ha modificado dicha línea por ser una estimación mía.

Con respecto a la función **condiciones-de-poda**, ésta queda como sigue:

```
fun condiciones-de-poda (e: ensayo) dev boolean
    dev condicion1 (e) ∧ condicion2 (e) ∧ condicion3 (e) ∧ condicion4 (e)
ffun
```

Ahora veremos las condiciones de poda del enunciado. La **primera condición** dice que la primera letra ha de ser una vocal:

```
fun condicion1 (e: ensayo) dev boolean
    dev vocal (e.palabra[1])
ffun
```

donde hemos utilizado una función auxiliar *vocal* que nos será útil para implementar las siguientes condiciones:

```
fun vocal (l: letra) dev boolean
    dev pertenece (l, {a, e, i, o, u})
ffun
```

La **segunda condición** dice que sólo pueden aparecer dos vocales seguidas si son diferentes. Podemos implementarla cumpliendo cualquiera de estos casos:

- La palabra tiene al menos dos letras
- O si la última no es vocal
- O si la última no es igual a la penúltima

Por tanto, sólo comprobaremos sobre las dos últimas letras porque el resto de la palabra ha debido de pasar ya las condiciones de poda en el momento de ser generado. La función quedaría así:

```
fun condicion2 (e: ensayo) dev boolean
    dev e.indice < 2 ∨
        consonante (e.palabra[e.indice]) ∨
        e.palabra[e.indice] ≠ e.palabra[e.indice − 1]
ffun
```

En cuanto a la función auxiliar *consonante* tendremos la siguiente implementación:

```
fun consonante (l: letra) dev boolean
    dev ¬vocal (l)
ffun
```

La **tercera condición** consiste en que no haya 3 letras seguidas del mismo tipo (vocales o consonantes). De nuevo teniendo en cuenta que sólo debe comprobarse que la última letra introducida no hace que se viole ninguna condición, puede cumplirse para cualquiera de estos casos:

- La palabra tiene menos de 3 letras
- O la última y la penúltima son de distinto tipo
- O la última y la antepenúltima son de distinto tipo

Por lo que la función sería:

```
fun condicion3 (e: ensayo) dev boolean
    i ← e.indice
    dev i < 2 ∨
        vocal (e.palabra[i − 1]) ≠ (e.palabra[i]) ∨
        vocal (e.palabra[i − 2]) ≠ (e.palabra[i])
ffun
```

Por último, la **cuarta y última condición** nos dice que no debe aparecer ninguna pareja de entre las de una lista C. De nuevo, hay que comprobarlo sólo para las dos últimas letras:

```
fun condicion4 (e: ensayo) dev boolean
    i ← e.indice
    dev no pertenece ((e.palabra[i − 1], e.palabra[i]), C)
ffun
```

Sólo nos queda escribir la función principal, que debe llamar a la vuelta-atrás tomando como argumento ensayo-vacio:

```
fun caracteres () dev lista de ensayo
    i ← e.indice
    dev vuelta-atrás (ensayo-vacio)
ffun
```

NOTA DEL AUTOR: En la solución del ejercicio esta función no tenia paréntesis, por lo que en la particular se ha añadido, de tal modo que se vea que realmente es una función sin argumentos.

5. Estudio del coste

El problema tiene un **tamaño acotado** y es, por tanto, de **coste constante**. Si lo generalizamos al problema de generar palabras de *n* letras con *m* restricciones, el coste estar en relación directa con el tamaño del árbol de búsqueda, que está acotado por 28^n, siempre que la verificación de las condiciones de poda se pueda realizar en tiempo constante, como en las 4 condiciones que teníamos. Este factor exponencial quiere decir que el problema es irresoluble, en la práctica, para valores grandes de *n*.

Problema 4.3 del libro de problemas resueltos (sin correspondencia con ningún ejercicio de examen)

Enunciado: Sobre un tablero de ajedrez de tamaño N (con $N > 5$) se coloca un caballo. Determinar una secuencia de movimientos del caballo que pase por todas las casillas del tablero sin pasar dos veces por la misma casilla. La posición inicial podrá ser cualquiera.

Un ejemplo de movimientos del caballo puede ser:

	M		M	
M				M
		C		
M				M
	M		M	

Donde los M en las casillas indica los alcanzables por el movimiento del caballo desde la casilla C.

NOTA DEL AUTOR: Antes de empezar a resolver el ejercicio, decir que originalmente las casillas marcadas con M eran casillas grises, sólo que no sabía muy bien hacer la casilla negra con el editor de texto. Sin más, pasamos a ver el ejercicio resuelto.

Respuesta:

1. Elección razonada del algoritmo

   Hay varias razones para **descartar tanto un esquema voraz como el de divide y vencerás**.

   En el primero de los casos el conjunto de candidatos tendría necesariamente que ser el de las casillas del tablero. En tal caso, un movimiento en falso nos haría finalizar el algoritmo sin éxito u obligarnos a deshacer un movimiento ya realizado. No hay, por tanto, función de selección, por lo que el esquema voraz es incorrecto.

   Con respecto al esquema de divide y vencerás se observa que no hay forma de descomponer el problema en otros de menos tamaño sin que se desvirtúe el mismo.

2. Descripción del esquema usado e identificación con el problema

   El esquema de vuelta atrás utiliza el retroceso como técnica de exploración de grafos. En este caso, el grafo es un árbol donde los nodos son tableros. No es pertinente utilizar ramificación y poda, puesto que no existe ningún parámetro que deba ser optimizado.

   El **esquema general** de vuelta atrás es el siguiente:

```
fun vuelta-atrás (ensayo)
    si valido (ensayo) entonces
        devolver ensayo
    si no
        para cada hijo en compleciones (ensayo) hacer
            si condiciones-de-poda (hijo) entonces
                vuelta-atrás (hijo)
            fsi
        fpara
    fsi
ffun
```

   La particularización del esquema al problema del caballo puede hacerse así:

   - **Válido**: Un ensayo será válido (solución) si no queda casilla alguna por visitar.
   - **Compleciones**: Se generarán tantos ensayos como casillas libres haya alcanzables directamente desde la última casilla visitada.
   - **Condiciones de poda**: Si sólo se generan ensayos sobre casillas alcanzables no hay ninguna condición de poda que añadir.

3. Estructuras de datos

   Necesitamos manejar los **siguientes elementos**:

   - La última posición que ha alcanzado el caballo.
   - El tablero con los movimientos efectuados.
   - El número de movimientos efectuados.

Para representar estos elementos se puede usar:

- Un par de enteros: *i* y *j*, que son las coordenadas de la última posición que ha alcanzado el caballo.
- Una matriz de enteros donde cada posición representa una casilla y el valor indica en qué movimiento ha pasado el caballo por la casilla. El valor 0 indica que esa casilla está libre.
- Un entero, que indica el número de movimientos efectuados, siendo la solución al alcanzar todos ellos.

4. Algoritmo completo a partir del refinamiento del esquema general

Como hemos indicado antes, un **ensayo** consta de 3 elementos.

La función **válido** utiliza únicamente uno de ellos para comprobar que el número de casillas ocupadas corresponde con el de casillas en el tablero. Ya veremos más adelante que esta comprobación es suficiente gracias a que las posiciones exploradas del árbol son únicamente *aquellas legales* y que, por tanto, son susceptibles de conducir a una solución.

La función **compleciones** tiene el esquema general siguiente:

```
fun compleciones (e: ensayo) dev lista de ensayos
    lista ← lista vacía
    si condiciones de poda (ensayo) entonces
        para cada rama hacer
            w ← generar ensayo (rama)
            lista ← insertar (w, lista)
        fpara
    fsi
    dev lista
ffun
```

NOTA: Se ha hecho una pequeña modificación en la función insertar (recordemos que previamente vimos algo parecido con añadir) en la que se intercambian los parámetros para indicar el primero de ellos el elemento a añadir (en este caso w) y el segundo donde se añade. Además, se ha modificado está función para que devuelva una lista de ensayos, que no estaba incluida en la solución de dicho ejercicio.

La única **condición de retroceso** posible es la del que el caballo haya alcanzado una posición tal que se encuentre rodeado de casillas ya recorridas y no pueda, por tanto, efectuar movimiento alguno sin repetir casilla. En este caso, el árbol no se sigue explorando y el algoritmo debe retroceder y buscar otra rama.

Estrictamente hablando lo anterior **no** es una **condición de poda**. En el caso expuesto bastaría con generar todas las posibles compleciones (que serian 0 puestos que deben ser validas) y devolver una lista vacía para que la función retroceda.

NOTA (del ejercicio): Al hablar de condiciones de poda nos referimos a que anticipamos que no hay solución por ese camino, no que hayamos llegado a un punto muerto. En ese sentido la condición de "caballo sin casillas libres a su alrededor" no es una condición de poda, ya que no es un criterio objetivo que anticipe a un camino erróneo. En el caso de este problema no hay en realidad condición de poda si no *condición de retroceso* y, por tanto, esta función no tiene sentido en esta implementación.

Los movimientos posibles del caballo pueden verse en el dibujo anterior. Se puede ver que, si el caballo ocupa una posición (a, b), las casillas alcanzables son $(a + i, b + j)$ en las que $i, j \in \{-2, -1, 1, 2\}$ y tales que $|i| + |j| = 3$. Con esta consideración, la generación de las casillas validas se puede hacer mediante un bucle con la condición anterior para formar los 8 nuevos movimientos posibles.

Para crear los nuevos tableros a partir de un movimiento válido se utiliza la función de **generar ensayo**. Esta función crea una nueva variable ensayo y genera:

1. Un nuevo registro de última casilla ocupada con la posición obtenida del bucle.
2. Un nuevo tablero copiado del ensayo anterior al que se le añade la nueva posición en la que escribimos el correspondiente valor del número de movimientos efectuados.
3. Incrementamos en 1 el valor del número de movimientos efectuados que contenía al anterior ensayo.

La función de **compleciones** adaptada a nuestro problema será:

```
fun compleciones (e: ensayo) dev lista de ensayos
    lista ← lista vacía
    (i, j) ←e.ultima-posición
    N ← e.numero-movs
    último-tablero ← e.tablero
    /* Generamos todas las posibles ramas */
    para i ← −2 hasta 2 hacer
        para j ← −2 hasta 2 hacer
            si (abs(i) + abs(j) = 3) ∧ (e.tablero[i, j] = 0) entonces
                nuevo-tablero ← último-tablero
                nuevo-tablero[i, j] ← N + 1
                nueva-posición ← (i, j)
                nuevo-ensayo ← <nuevo-tablero, nueva-posición, N + 1>
                lista ← insertar (nuevo-ensayo, lista)
            fsi
        fpara
    fpara
    dev lista
ffun
```

NOTA: De nuevo hacemos hincapié en que cada sentencia acaba en punto y coma, pero por evitar líos innecesarios no se hace en ninguna de las funciones anteriores. Por otro lado, algunas funciones podremos una variable (por ejemplo, e) como argumento o bien la propia estructura de datos (por ejemplo, ensayo), siendo indiferente su uso, debido a que la solución en todos los casos es idéntica. Digo esto, porque en la solución del ejercicio lo han puesto con el nombre de la estructura de datos en todas las funciones y en la solución escrita por mi es justamente el primer modo de poner los argumentos.

Por último, un tablero es **válido** (solución) si hemos realizado N^2 movimientos, con lo que la función quedaría así:

```
fun válido (e: ensayo) dev boolean
    dev (e.numero-movs = N^2)
ffun
```

En la función principal, para recorrer la lista de compleciones, el esquema principal debe utilizar un bucle mientras … hacer. La **función principal** queda como sigue:

```
fun saltocaballo (e: ensayo)
    si válido (e) entonces
        escribe ensayo
    si no
        lista ← compleciones (e)
        mientras ¬vacía (lista) hacer
            w ← primer elemento (lista)
            saltocaballo (w)
            lista ← resto (lista)
        fmientras
    fsi
ffun
```

5. Estudio del coste

El estudio del coste en los algoritmos de búsqueda con retroceso es difícil de calcular con exactitud, ya que se desconoce el tamaño de lo explorado y sólo es posible en la mayoría de los casos dar una **cota superior** al tamaño de la búsqueda. Sin tener en cuenta las reglas de colocación del caballo en un tablero, N^2 casillas pueden rellenarse con números entre 1 y n^2 de $(n^2)!$ maneras posibles.

La anterior aproximación es demasiado burda, ya que tenemos la información del número máximo de ramificaciones del árbol que es 8. Considerando esto el tamaño del árbol se puede acotar en $\mathbf{8^{n^2}}$. Además, se puede precisar que no hay 8 posibles movimientos más que desde una parte del tablero. De cada n^2 casillas, sólo desde $n^2 - 8 * (n-2)$ es posible realizar 8 movimientos y además los últimos movimientos no tendrán prácticamente ninguna alternativa.

Problema 4.4 del libro de problemas resueltos (sin correspondencia con ningún ejercicio de examen)

Enunciado: Sobre una cuadrícula de 3x3 casillas se quieren colocar los dígitos del 1 al sin repetir ninguno y de manera que sumen 15 en todas las direcciones, incluidas las diagonales. Diseñar un algoritmo que busque todas las soluciones posibles.

Una posible solución para el problema puede ser:

2	7	6
9	5	1
4	3	8

Respuesta:

1. Elección razonada del esquema algorítmico

   El pasatiempo requiere tomar números y colocarlos de forma que encajen en la cuadricula de manera que cumplan con las condiciones del enunciado. Podemos ver una solución en la figura de antes. Aunque haya claramente un conjunto de elementos candidatos, no hay sin embargo forma de prever que podemos cometer un error en la elección y, por tanto, vernos obligados a deshacer una decisión ya tomada. El esquema correcto es el de **vuelta atrás**, ya que por otra parte no es posible descomponerlo en subproblemas de su misma naturaleza.

2. Descripción del esquema usado e identificación con el problema

   El **esquema general** de vuelta atrás es el siguiente:

```
fun vuelta-atrás (ensayo)
    si valido (ensayo) entonces
        devolver ensayo
    si no
        para cada hijo en compleciones (ensayo) hacer
            si condiciones-de-poda (hijo) entonces
                vuelta-atrás (hijo)
            fsi
        fpara
    fsi
ffun
```

   Condiciones de poda: La búsqueda de una combinación correcta de números en las casillas no es totalmente ciega. Tenemos varios criterios (condiciones de poda) y así evitaremos explorar ramas del árbol que recorre el algoritmo:

   1. Uno de los más claros es el de no considerar aquellas combinaciones en las que al haber puesto 2 números en una fila, éstos sumen una cantidad tal que $s < 6$ o bien $s > 14$, ya que en estos casos ninguna combinación posterior encontrará solución.
   2. Otras condiciones pueden limitar la presencia en la casilla central de determinados números. Por ejemplo, el 1 o el 9. No interesa, en general, realizar suposiciones poco fundadas o intuitivas. Las condiciones de poda deben ser rigurosas y demostrables, ya que de lo contrario podremos estar descartando soluciones válidas. En este problema, no se impondrán condiciones al número ubicado en la casilla central.

   NOTA (del ejercicio): Es importante establecer criterios de poda que nos ayuden lo más posible a reducir el tamaño del árbol de búsqueda, es decir, a podar.

   Compleciones: Debe generar todos los posibles hijos de un nodo. A partir de una cuadricula y de un conjunto de números sin usar hasta entonces se forman los hijos de dicho nodo. Cada uno de estos nuevos nodos tendrá con el nodo padre la diferencia de tener un nuevo número, pero considerando que:

   1. Hay que crear un nuevo nodo por cada uno de los números disponibles.
   2. Hay que crear un nuevo nodo por cada una de las casillas libres donde puede colocarse un número.

3. Estructura de datos

Para representar las casillas y su contenido se utilizará una matriz de números naturales. La lista de los números que ya han sido colocados puede estar recogida en una estructura de datos de tipo conjunto.

Un **ensayo** constará de:

- Una matriz de 3x3 de números naturales. Por ejemplo, cuadricula.
- Un conjunto de números disponibles. Por ejemplo, candidatos o disponibles.
- Un número natural n que indique cuantos valores hay colocados en la cuadricula. Por ejemplo, num-ocupadas.

Se define tipoCuadrado como un *registro* de lo anterior.

4. Algoritmo completo a partir del refinamiento del esquema general

Una vez vistos estos conceptos de **condiciones de poda** nos queda decir que dicha función devolverá *cierto* si un determinado nodo es susceptible de llegar a alcanzar una solución válida, por lo que evitaremos explorar ramas, siguiendo el primer criterio, sobre todo (el segundo hemos visto que al no ser riguroso no se impondrá esta condición).

Para comprobar las condiciones expuestas creamos unas funciones auxiliares que verifiquen las sumas en horizontal, vertical y diagonal. Basta con utilizar la misma función que diseñaremos para comprobar que la suma es 15.

```
fun suma-fila (matriz: vector[1..3,1..3] de natural, fila: natural) dev natural
    s ← 0
    para i ← 1 hasta 3 hacer
        s ← s + matriz[fila, i]
    fpara
    dev s
ffun
```

Para las columnas se utiliza una función análoga. Asimismo se construyen otras dos funciones que compruebe el número de elementos distinto de cero en una fila y en una columna dada:

```
fun num-f (matriz: vector[1..3,1..3] de natural, fila: natural) dev natural
fun num-c (matriz: vector[1..3,1..3] de natural, columna: natural) dev natural
```

La función general de **compleciones** será:

```
fun compleciones (e: ensayo) dev lista de ensayos
    lista ← lista vacía
    si condiciones de poda (ensayo) entonces
        para cada rama hacer
            w ← generar ensayo (rama)
            lista ← insertar (w, lista)
        fpara
    fsi
    dev lista
ffun
```

La función de compleciones debe generar una lista de compleciones que tenga en cuenta lo anterior. El esquema general un poco más refinado es:

```
fun compleciones (cuadrado: tipoCuadrado) dev lista de ensayos
    lista ← lista vacía
    para i ← 1 hasta 9 hacer
        si i en cuadrado.disponibles entonces
            w ← generar ensayo (cuadrado, i)
            lista ← insertar (w, lista)
        fsi
    fpara
    dev lista
ffun
```

NOTA DEL AUTOR: Como es habitual intercambiamos los parámetros en la función insertar y además nos fijamos que a diferencia del esquema anterior a la hora de generar ensayo toma dos parámetros. Conceptualmente, la variable i es un número disponible del cuadrado, por lo que creo que esta generación de ensayo corresponde a crear el hijo, siendo "cuadrado" el padre del mismo.

Faltan por especificar las funciones de **condiciones de poda** y de **generar ensayo**. La parte de la función compleciones que genera las ramas consta de una serie de bucles anidados para contemplar todas las ramificaciones a partir de un nodo. También se comprueba que se generen sólo ramas validas, es decir, cualquier combinación de una cuadricula en la que hayamos puesto ya 3 números y en la que estos no sumen 15 es descartada, las cuales serian las condiciones de poda.

La función generar ensayo se limita a tomar la información necesaria y generar la estructura de datos resultante de integrarla. Con una cuadrícula, una posición dentro de ella y el valor numérico que debe insertar forma un nuevo nodo.

```
fun generar ensayo (cuadrado: tipoCuadrado; i, j, n: natural) dev ensayo
    var nodo: tipoCuadrado
    nodo ← cuadrado
    nodo.cuadricula[i, j] ← n
    nodo.num-ocupadas ← nodo.num-ocupadas + 1
    nodo.candidatos ← nodo.candidatos - {n}
    dev nodo
ffun
```

NOTA DEL AUTOR: En la siguiente línea ocurre lo mismo exactamente que se vió en el problema 4.1, en la que la línea siguiente:

```
nodo.num-ocupadas ← nodo.num-ocupadas + 1
```

no le veo sentido, por lo que la siguiente estimo que es la correcta

```
nodo.num-ocupadas ← cuadrado.num-ocupadas + 1
```

Por último, en la siguiente línea a ésta entiendo que puede ser que no esté correctamente escrita, porque modifica el nodo eliminando el valor de la cuadricula dada (n) y, a continuación, devuelve el nodo tras esa modificación. Estimo que no es lo suyo hacerlo así, ya que al final no hemos creado ningún nodo. En mi opinión eliminaría esta misma línea. No obstante, dejamos la solución planteada en el libro de problemas, tal y como se muestra.

La implementación de la función de **condiciones de poda** queda como sigue:

```
fun condiciones de poda (cuadrado: tipoCuadrado) dev boolean
    para f ←1 hasta 3 hacer
        si suma-fila (cuadrado.cuadricula, f) < 6 entonces
            dev falso
        fsi
        si num-f (cuadrado.cuadricula, f) < 3 ∧
          suma-fila (cuadrado.cuadricula, f) > 14 entonces
            dev falso
        fsi
    fpara
    para c ←1 hasta 3 hacer
        si suma-fila (cuadrado.cuadricula, c) < 6 entonces
            dev falso
        fsi
        si num-f (cuadrado.cuadricula, c) < 3 ∧
          suma-fila (cuadrado.cuadricula, c) > 14 entonces
            dev falso
        fsi
    fpara
    dev cierto
ffun
```

Teniendo en cuenta lo anteriormente visto tendremos que la implementación final de la función **compleciones** queda:

```
fun compleciones (cuadrado: tipoCuadrado) dev lista de ensayos
    lista ← lista vacía
    si condiciones de poda (cuadrado) hacer
        para n ← 1 hasta 9 hacer
            para i ← 1 hasta 3 hacer
                para j ← 1 hasta 3 hacer
                    si n en cuadrado.disponibles  ∧
                      cuadrado[i,j] = vacio  ∧
                      parcialmente válido(cuadrado.cuadrícula) entonces
                        w ← generar ensayo (cuadrado, i, j, n)
                        lista ← insertar (w, lista)
                    fsi
                fpara
            fpara
        fpara
    fsi
    dev lista
ffun
```

NOTA DEL AUTOR: Se modifica la línea de generar ensayo, debido a que según la solución del problema tiene el mismo número de argumentos que la que vimos anteriormente (la refinada de la función general). Hemos visto previamente, como dicha función tenía 4 argumentos distintos, por eso se ha modificado.

Como hemos indicado antes, la función **parcialmente válido** comprueba que todas aquella filas, columnas o diagonales con 3 valores sumen 15.

Como último *comentario del autor* decir que se ha modificado la estructura del ejercicio debido a que había código en el punto 2 (que sólo es descriptivo) y se ha trasladado al punto 4 (refinamiento del esquema general). He tratado de hacerlo lo más lógico posible sin modificar gran código de la solución aportada en el ejercicio.

5. Estudio del coste

El coste depende del tamaño del árbol. Por lo general basta con estimar éste último para saber la complejidad. En casi todos los casos se trata de problemas con un coste exponencial.

En este problema el **tamaño es fijo**. El cuadrado es siempre de 3 x 3 y no hay un "orden" para el algoritmo. En estos casos trataremos, en la medida de lo posible, de acotar superiormente el número máximo de operaciones. Al principio hay 9 números y 9 casillas, por lo que las posibilidades para el primer número son 9^2. Para el siguiente número hay 8^2 y así sucesivamente. De nuevo tenemos con esto una cota irreal por estas razones:

1. Primero porque estamos descartando un enorme número de ramas que no cumplen con las condiciones exigidas de suma 15, y
2. Segundo porque en realidad no estamos teniendo en cuenta la simetría de la cuadricula.

Febrero 2003-1ª (problema)

Enunciado: Teseo se adentra en el laberinto en busca de un minotauro que no sabe dónde está. Se trata de implementar una función *ariadna* que le ayude a encontrar el minotauro y a salir después del laberinto. El laberinto debe representarse como una matriz de entrada a la función cuyas casillas contienen uno de los tres valores:

0 para "camino libre"
1 para "pared" (no se puede ocupar)
2 para "minotauro"

Teseo sale de la casilla (1,1) y debe encontrar la casilla ocupada por el minotauro. En cada punto, Teseo puede tomar la dirección Norte, Sur, Este u Oeste, siempre que no haya una pared. La función *ariadna* debe devolver la secuencia de casillas que componen el camino de regreso desde la casilla ocupada por el minotauro hasta la casilla (1,1).

Respuesta:

1. Elección razonada del esquema algorítmico

Como no se indica nada al respecto de la distancia entre casillas adyacentes, y ya que se sugiere únicamente una matriz, es lícito suponer que la distancia entre casillas adyacentes es siempre la misma (1, sin pérdida de generalidad). Por otra parte, no se exige hallar el camino más corto entre la entrada y el minotauro, sino que el enunciado sugiere, en todo caso, las posibles soluciones (y ayudar a salir a Teseo cuanto antes).

Tras estas consideraciones previas, ya es posible elegir el esquema algorítmico más adecuado. El tablero puede verse como un grafo en el que los nodos son las casillas y en el

que como máximo surgen cuatro aristas (N, S, E, O). Todas las aristas tienen el mismo valor asociado (por ejemplo, 1).

En primer lugar, el algoritmo de Dijkstra queda descartado. No se pide el camino más corto y si se hiciera, las particularidades del problema hacen que el camino más corto coincida con el camino de menos nodos y, por tanto, una exploración en anchura tendrá un coste menor: siempre que no se visiten nodos ya explorados, como mucho se recorrerá todo el tablero una vez (coste lineal con respecto al número de nodos versus coste cuadrático para Dijkstra).
En segundo lugar, es previsible esperar que el minotauro no esté cerca de la entrada (estará en un nivel profundo del árbol de búsqueda) por lo que los posibles caminos solución serían largos. Como no es necesario encontrar el camino más corto, sino encontrar un camino lo antes posible, una búsqueda en profundidad resulta más adecuada que una en anchura. En el peor de los casos en ambas habrá que recorrer todo el tablero una vez, pero ya que buscamos un nodo profundo se puede esperar que una búsqueda en profundidad requiera explorar menos nodos que una en anchura.
En tercer lugar, es posible que una casilla no tenga salida por lo que es necesario habilitar n mecanismo de retroceso.
Por último, es necesario que no se exploren por segunda vez casillas ya exploradas anteriormente.

Por estos motivos, se ha elegido el esquema de **vuelta atrás**.

2. Descripción del esquema usado

Vamos a utilizar el esquema de vuelta atrás modificado para que la búsqueda se detenga en la primera solución y para que devuelva la secuencia de ensayos que han llevado a la solución en orden inverso (es decir, la secuencia de casillas desde el minotauro hasta la salida).

```
fun vuelta-atrás (ensayo) dev (es_solucion, solución)
    si valido (ensayo) entonces
        solución ← crear-lista()
        solución ← añadir (ensayo, solución)
        devolver (verdadero, solución)
    si no
        hijos ← crear-lista()
        hijos ← compleciones (ensayo)
        es_solucion ← false
        mientras no vacía (hijos) y no es_solución hacer
            hijo ← primero (hijos)
            hijos ← resto (hijos)
            si cumple-poda (hijo) entonces
                (es_solucion, solución) ← vuelta-atrás (hijo)
            fsi
        fmientras
        si es_solucion entonces
            solución ← añadir (ensayo, solución)
        fsi
        devolver (es_solucion, solución)
    fsi
ffun
```

3. Estructuras de datos

Un ensayo constará de dos enteros *x* e *y* para almacenar las casillas

Además, utilizaremos:

- Una **lista de casillas**, para almacenar la solución y otra para las compleciones.
- Una **matriz de enteros** del tamaño del laberinto inicializada con la configuración del laberinto.
- Una **matriz de booleanos** de igual tamaño que el laberinto, para llevar control de los nodos visitados.

La estructura de datos será, por tanto, la siguiente:

```
tipoCasilla = registro
    x, y: entero
    fregistro

tipoLista                    // Lista de casillas
```

Será necesario implementar en nuestro algoritmo las siguientes funciones de lista:

1. crear-lista
2. vacía
3. añadir
4. primero

NOTA DEL AUTOR: Se ha reescrito esta parte del ejercicio para que tenga el mismo formato que anteriores ejercicios, sin modificar la solución dada.

4. Algoritmo completo

Modificaremos el esquema general de vuelta atrás quedando:

```
fun vuelta-atrás (laberinto: vector[1..LARGO, 1..ANCHO] de entero;
                  casilla: tipoCasilla;
                  visitados: vector[1..LARGO, 1..ANCHO] de boolean)
                  dev (es_solucion: boolean, solución: tipoLista)
    visitados[casilla.x, casilla.y] ← verdadero
    si laberinto[casilla.x, casilla.y] == 2 entonces
        solución ← crear-lista()
        solución ← añadir (casilla, solución)
        devolver (verdadero, solución)
    si no
        hijos ← crear-lista()
        hijos ← compleciones (laberinto, casilla)
        es_solucion ← false
        mientras no vacía (hijos) y no es_solución hacer
            hijo ← primero (hijos)
            hijos ← resto (hijos)
            si no visitados[hijo.x, hijo.y]entonces
                (es_solucion, solución) ← vuelta-atrás (laberinto, hijo, visitados)
            fsi
        fmientras
        si es_solucion entonces
            solución ← añadir (casilla, solución)
        fsi
        devolver (es_solucion, solución)
    fsi
ffun
```

En el caso de encontrar al minotauro, es decir, cuando el valor de la casilla es 2 se detiene la exploración en profundidad y al deshacer las llamadas recursivas se van añadiendo a la solución de las casillas que se han recorrido. Como se añaden al final de la lista, la primera será la del minotauro y la última la casilla (1,1), tal como pedía el enunciado.

La función **compleciones** comprobará que la casilla no es una pared (recordemos que el valor es 1) y que no esté fuera del laberinto, siendo:

```
fun compleciones (laberinto: vector [1..LARGO, 1..ANCHO] de entero;
                  casilla: tipoCasilla) dev tipoLista
    hijos ← crear-lista()
    si casilla.x+1 ≤ LARGO entonces          /* Movimiento a derecha */
        si laberinto[casilla.x + 1, casilla.y] <> 1 entonces
            casilla_aux.x = casilla.x + 1
            casilla_aux.y = casilla.y
            hijos ← añadir (casilla_aux, solución)
        fsi
    fsi
    si casilla.x-1 ≥ 1 entonces              /* Movimiento a izquierda */
        si laberinto[casilla.x − 1, casilla.y] <> 1 entonces
            casilla_aux.x = casilla.x − 1
            casilla_aux.y = casilla.y
            hijos ← añadir (casilla_aux, solución)
        fsi
    fsi
    si casilla.y+1 ≤ ANCHO entonces          /* Movimiento a abajo */
        si laberinto[casilla.x, casilla.y + 1] <> 1 entonces
            casilla_aux.x = casilla.x
            casilla_aux.y = casilla.y + 1
            hijos ← añadir (casilla_aux, solución)
        fsi
    fsi
    si casilla.y-1 ≥ 1 entonces              /* Movimiento a arriba */
        si laberinto[casilla.x, casilla.y − 1] <> 1 entonces
            casilla_aux.x = casilla.x
            casilla_aux.y = casilla.y − 1
            hijos ← añadir (casilla_aux, solución)
        fsi
    fsi
ffun
```

5. Análisis del coste

Todas las operaciones son constantes salvo la llamada recursiva a vuelta-atrás. En cada nivel, pueden realizarse hasta 4 llamadas. Sin embargo, las llamadas no se realizan si la casilla ya ha sido visitada. Esto quiere decir que, en el caso peor, sólo se visitará una vez cada casilla. Como las operaciones para una casilla son de complejidad constante, la complejidad será de $\boldsymbol{O(ANCHO * LARGO)}$, lineal con respecto al número de casillas.

Febrero 2004-2ª (problema)

Enunciado: Dado un grafo finito, con ciclos y con todas las aristas de coste unitario, implementar un algoritmo que devuelva el número de nodos que contiene el camino más largo sin ciclos que se puede recorrer desde un determinado nodo.

Respuesta:

1. Elección del esquema

   Hallar el camino más largo desde un nodo en un grafo finito puede verse como un problema de determinar cuál es la profundidad máxima del árbol. Para ello, será necesario recorrer todos los caminos sin ciclos del grafo, almacenando la longitud del más largo encontrado hasta el momento. El esquema de **búsqueda exhaustiva en profundidad** nos permite resolver este problema.

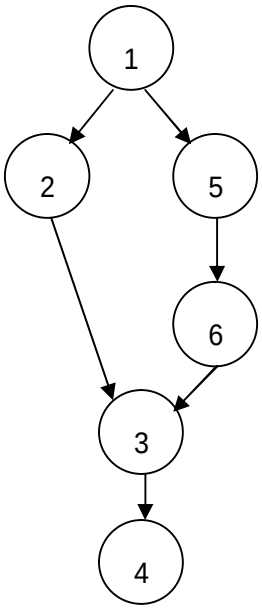
2. Descripción del esquema e identificación con el problema

   El esquema de **recorrido en profundidad** será:

```
proc rp (nodo)
    { El nodo no ha sido visitado anteriormente }
    visitado[nodo] := cierto
    para cada hijo en adyacentes[nodo] hacer
        si visitado[nodo] = falso entonces
            rp (hijo)
        fsi
    fpara
fproc
```

   NOTA DEL AUTOR: Este procedimiento es parecido al dado en el resumen del tema, pero modificado en la solución dada al problema, aun así conceptualmente es similar. Habrá en ocasiones, como en este caso, en la que representaremos la asignación (←) como :=, siendo ambas representaciones equivalentes.

Así planteado, un nodo **no** se exploraría dos veces aunque llegáramos a él desde una rama distinta. Esto es incorrecto y se debe a que el vector de nodos visitados se plantea de forma global. Veamos un ejemplo:



Una exploración llega al nodo y lo marca como *visitado*. Análogamente, sigue marcando como *visitados* los nodos 2, 3 y 4, hallando un camino de longitud 4. Cuando el control regresa al nodo 1 para explorar ramas alternativas, se explora el nodo 5, el 6, pero cuando llega al nodo 3, se detiene la exploración porque 3 está marcado como *visitado*. El nuevo camino encontrado tiene 4 nodos en vez de 5, que sería lo correcto.

Este problema lo podemos corregir si el vector de visitados tiene un ámbito local en vez de global y se pasa una copia de padres a hijos (paso de parámetro por valor).

3. Estructuras de datos

Necesitaremos una estructura para grafo. La más compacta y sencilla de usar es un **vector** en el que la componente i tiene un puntero a la lista de nodos adyacentes al nodo i. por tanto, también necesitaremos implementar una **lista de nodos** que puede ser, simplemente, una *lista de enteros*.

Además, necesitaremos un **vector de nodos visitados** por rama.

4. Algoritmo completo

Daremos un **algoritmo completo** al problema:

```
fun longitud (nodo, visitados) dev long
    { El nodo no ha sido visitado anteriormente }
    visitados [nodo] := cierto
    long_max := 0
    para cada hijo en adyacentes [nodo] hacer
        si visitados [nodo] = falso entonces
            long := longitud (hijo, visitados)
            si long > long_max entonces
                long_max = long
            fsi
        fsi
    fpara
    devolver long_max + 1
ffun
```

La **llamada inicial** se realizar con el nodo inicial y los elementos del vector de visitados inicializado a falso.

NOTA DEL AUTOR: Recalcamos de nuevo que todas las sentencias acaban en punto y coma, además que este algoritmo sería una función (y no un procedimiento) por devolver un valor, lo cual está modificado.

5. Estudio del coste

Sea n el número de nodos del grafo. Si suponemos un grafo denso (recordemos de los algoritmos voraces, en el que todos o casi todos los nodos están conectados entre sí) tenemos que la longitud del camino máximo será n. Cada llamada recursiva, entonces, supondrá una reducción en uno del problema. Así mismo, el número de llamadas recursivas por nivel también irá disminuyendo en uno puesto que en cada nivel hay un nodo adyacente más, que ya ha sido visitado. Por último, dentro del bucle se realizan al menos n comparaciones. Por tanto, podemos establecer la siguiente **ecuación de recurrencia**:

$$T(n) = (n-1) * T(n-1) + n$$

Como hacíamos en los ejercicios anteriores veremos las siguientes variables para la recursión por reducción:

a: Número de llamadas recursivas = $n - 1$
b: Reducción del problema en cada llamada recursiva = 1
$\mathbf{c} * \mathbf{n^k}$: Todas aquellas operaciones que hacen falta además de la recursividad. Tendremos, por tanto, $c * n^k = n \Rightarrow n = 1$.

Aplicando la formula siguiente para dicha recursión de **reducción por sustracción** tendremos:

$$T(n) = \begin{cases} \theta(n^k) & \text{si } a < 1 \\ \theta(n^{k+1}) & \text{si } a = 1 \\ \theta\left(a^{n \; div \; b}\right) & \text{si } a > 1 \end{cases}$$

Como hemos visto previamente el caso será el de $a > 1$, por el que resolviéndolo tendremos que $T(n) \in \theta(a^{n\ div\ b}) = \theta(n^n)$.

Se puede llegar a esta conclusión razonando sobre la forma del árbol de exploración.

Febrero 2005-1ª (problema)

Enunciado: Partiendo de un conjunto $N = \{n_1, n_2, \ldots, n_m\}$ compuesto por m números positivos y de un conjunto $O = \{+, -, *, /\}$ con las operaciones aritméticas básicas. Se pide obtener una secuencia de operaciones factible para conseguir un número objetivo P. como restricciones al problema, debe tenerse en cuenta que:

a) Los números del conjunto N pueden utilizarse en la secuencia de operaciones 0 ó 1 vez.
b) Los resultados parciales de las operaciones pueden utilizarse como candidatos en operaciones siguientes
c) Las operaciones que den como resultado valores negativos o números no enteros NO deberán tenerse en cuenta como secuencia válida para obtener una solución.

Diseñe un algoritmo que obtenga una solución al problema propuesto, mostrando la secuencia de operaciones para obtener el número objetivo P. en caso de no existir solución alguna, el algoritmo deberá mostrar la secuencia de operaciones que dé como resultado el valor más próximo por debajo del número objetivo P.

Por ejemplo, siendo $P = 960$ y $N = \{1,2,3,4,5,6\}$, la secuencia de operaciones que obtiene la solución exacta es:

$$\Big(\big((6 * 5) * 4\big) * 2\Big) * (3 + 1)\Big) = 960.$$

Si $P = 970$, el algoritmo encontraría la solución exacta con el conjunto de números inicial y la secuencia más próxima por debajo de P seria la dada anteriormente.

Respuesta:

1. Elección razonada del esquema algorítmico

   Obviamente **no** se trata de un problema que pueda ser resuelto por divide y vencerás, puesto que no es posible descomponer el problema en subproblemas iguales, pero de menor tamaño que con algún tipo de combinación nos permita encontrar la solución del problema.

   Tampoco se trata de un algoritmo voraz, puesto que no existe ninguna manera de atacar el problema de manera directa que nos lleve a la solución sin necesidad de, en algún momento, deshacer alguna de las decisiones tomadas.

   Por tanto, se trata de un problema de exploración de grafos donde deberemos construir el grafo implícito al conjunto de operaciones posibles con el fin de encontrar una solución al problema. Descartamos una exploración ciega en profundidad o anchura, puesto que, como en la mayor parte de los casos va a existir por lo menos una solución y además el enunciado del problema una solución al problema y no todas, por lo que es deseable que la exploración se detenga en el momento en el que encuentre alguna de ellas. Sólo en el caso de que no exista solución al problema, a partir del conjunto N inicial, el recorrido deberá ser completo. De acuerdo a este razonamiento, la estrategia más apropiada parece la de aplicar un esquema de tipo **backtracking o vuelta atrás**.

Como ya se conoce previamente, este tipo de algoritmo se basa en un recorrido en profundidad o en anchura, que no construye el grafo implícito de manera exhaustiva, puesto que dispone de condición de poda que lo detiene en cuanto se encuentra una solución. En nuestro caso, además, basaremos el algoritmo en un *recorrido en profundidad* y no en anchura, puesto que en la mayor parte de las ocasiones es necesario combinar prácticamente la totalidad de los elementos de N para obtener la solución.

La alternativa de aplicar un algoritmo de ramificación y poda **no** es válida en este caso, pues este tipo de algoritmo se caracteriza por obtener la solución óptima a un problema concreto. En este caso, no es necesario optimizar absolutamente nada, pues la solución es el número objetivo P que nos solicitan y no van a existir, por tanto, soluciones mejores o peores. Sólo en el caso de que no exista solución, el problema nos pide la más aproximada, por debajo, lo cual implica una exploración exhaustiva del grafo implícito y, por tanto, el conocimiento por parte del algoritmo de cuál ha sido la operación más cercana realizada hasta el momento.

2. Descripción del esquema algorítmico

El **esquema general** de vuelta atrás es:

```
fun vuelta-atrás (e: ensayo) dev resultado
    si valido (e) entonces
        devolver e
    si no
        listaensayos ← compleciones (e)
        mientras no vacía (listaensayos) ∧ no resultado hacer
            hijo ← primero (listaensayos)
            listaensayos ← resto (listaensayos)
            si condiciones-de-poda (hijo) entonces
                resultado ← vuelta-atrás (hijo)
            fsi
        fmientras
        dev resultado
    fsi
ffun
```

Se observa que el algoritmo dado es parecido al segundo esquema dado en el resumen del tema.

3. Estructuras de datos

La **estructura de datos principal** para llevar a cabo nuestro algoritmo va a ser aquélla encargada de almacenar la información relativa a cada uno de los ensayos generados:

```
tipo Tensayo = tupla
    candidatos: vector de enteros
    operaciones: vector de TPilaOperaciones
    solucion: boolean
    vacio: boolean
ftupla
```

La parte que viene a continuación es opcional y personalmente no apropiada para el examen, ya que no da tiempo material a hacerlo. Por tanto, aunque lo añadamos al apartado estimo que no debería entrar todo en el examen.

Las operaciones asociadas de la tupla anterior son:

getCandidatos (): vector
    Devuelve el vector de candidatos para operar con él
getOperaciones (): vector
    Devuelve el vector de operaciones (pilas) para operar con él
getCandidato (indexCand int): int
    Devuelve el candidato que se encuentra en la posición indexCand
removeCandidato (indexCand int): void
    Elimina el candidato que se encuentra en la posición indexCand
setCandidato (candidato int, indexCand int): void
    Inserta el candidato *candidato* en la posición indexCand del vector de candidatos
removeOperacion (indexOp int): TPilaOperaciones
    Devuelve la operación (pila) que se encuentra en la posición indexOp
setOperacion (operación TPilaOperaciones, indexOp int): void
    Inserta la pila de operaciones operación en la posición indexOp del vector de operaciones
setSolucion (solución boolean): void
    Marca el ensayo como solución válida
isSolucion (): boolean
    Devuelve un booleano que indica si el ensayo es o no solución
isVacio (): boolean
    Devuelve un booleano que indica si el ensayo es o no vacio
setVacio (vacio boolean): void
    Marca el ensayo como vacio

En cuanto a la otra estructura de datos, tenemos lo siguiente:

tipo TPilaOperaciones = tupla
    pila: pila de String

De nuevo, esta parte la considero opcional, pero lo pondremos para completar el ejercicio. Las operaciones asociadas son:

pushNumber (value int): void
    Añade un número a la pila. La lógica de la operación transforma el entero de entrada en una cadena de caracteres para poder insertarlo en la pila.
pushOperator (oper char): void
    Añade un operador a la pila. La lógica de la operación transforma el entero de entrada en una cadena de caracteres para poder insertarlo en la pila.

El **principal problema** del ejercicio se encuentra en determinar cómo vamos a ir construyendo el grafo implícito y cómo vamos a representar las operaciones que ya han sido llevadas a cabo. Para ello, vamos a tener en cuenta lo siguiente:

1. Nuestro algoritmo siempre va a trabajar sobre un conjunto de candidatos que recogerá inicialmente los valores asociados y, posteriormente, los valores obtenidos al ir realizando cada una de las operaciones. Esta es la función del elemento **candidatos** de Tensayo.

2. Para poder recordar qué operaciones se han llevado a cabo y mostrárselas al usuario al fin del algoritmo, es necesario desplegar un almacén de información paralelo a candidatos que, para cada uno de los candidatos recoja las operaciones que éste ha soportado hasta el momento. Con este fin se crea el elemento **operaciones** que no es más que un vector donde cada elemento representa una pila de operaciones y operandos que, en notación postfija, representan el historial de operaciones de dicho elemento.
3. Finalmente, el elemento **solución** marca el ensayo como solución o no, dependiendo de si alberga la solución al problema.

Vamos con un ejemplo el *funcionamiento* de esta estructura de datos. Supongamos que nuestro conjunto inicial es $N = \{1,2,3,4\}$. El **ensayo** correspondiente a esta situación inicial vendría descrito por:

```
tipo Tensayo = tupla
    candidatos: vector de int
    operaciones: vector de TPilaOperaciones
    solucion: boolean
ftupla
```

ENSAYO

candidatos: ⟨1,2,3,4⟩

operaciones: < [1] [2] [3] [4] >

solucion: false

Supongamos ahora que operamos el candidato 2 y el 4 con el operador '+'. El nuevo formato del ensayo sería el siguiente:

ENSAYO

candidatos: ⟨1,6,3⟩

operaciones: < [1] [2 / 4 / +] [3] >

solucion: false

vacio: false

Nótese que:

1. Desaparece el elemento que ocupa la posición 3 del vector y su pila asociada.
2. El vector de candidatos ahora almacena el valor 6 allí donde se ha realizado la operación.

3. El vector de operaciones ha actualizado la pila, reflejando la nueva situación en notación postfija (2,4,+).

Supongamos que ahora operamos el candidato 6 con el 3 utilizando el operador '-'. El nuevo ensayo quedaría:

ENSAYO

candidatos: ⟨1,3⟩

operaciones: < 1 | 2 4 + 3 - >

solucion: false

vacio: false

Finalmente, si operamos el candidato 1 con el 3 utilizamos el operador '*'. El ensayo obtenido sería:

ENSAYO

candidatos: ⟨3⟩

operaciones: < * | 2 4 + 3 - 1 >

solucion: false

vacio: false

Como podemos observar, el valor final obtenido combinando todos los valores iniciales, y de acuerdo a las operaciones descritas, sería 3 y el historial completo de todas las operaciones realizadas podría mostrarse deshaciendo la pila en formato postfijo generada $(2,4,+,3,-,1,*)$.

4. Algoritmo completo

El **algoritmo completo** será el siguiente:

```
fun vuelta-atrás (e: Tensayo): Tensayo
    si valido (e) entonces
        solucion ← e
        solución.setSolucion(true)
        dev solucion
    si no
        listaensayos ← compleciones (e)
        mientras no vacía (listaensayos) ∧ no solución.isSolucion() hacer
            hijo ← primero (listaensayos)
            listaensayos ← resto (listaensayos)
            si  no podar (hijo) entonces
                solucion ← vuelta-atrás (hijo)
            si no
                solucion ← mejor (e)
            fsi
        fmientras
        dev solucion
    fsi
ffun
```

Nótese que **solución** es un ensayo, que inicialmente es vacio, y que contendrá la solución en caso de existir o la serie de operaciones que más se aproximen en caso de que esto no ocurra (recordemos el enunciado que era lo que se solicitaba). Solución se define externamente a la función vuelta-atrás y, por eso, puede manipularse desde cualquiera de las funciones sin ser necesario enviarla a éstas como parámetro. Es, por tanto, una variable global.

Igualmente, el valor objetivo P, también es utilizado globalmente por la función vuelta-atrás.

Esta parte que veremos a continuación, al igual que la de las operaciones asociadas de las estructuras de datos lo considero inviable a la hora de hacer el examen. Pienso personalmente que se podrían tomar estas mismas funciones y simplificarlas. Como antes se ha puesto, se escribe la solución aportada por el ejercicio. Veremos a continuación las *funciones asociadas* al algoritmo:

La primera de ellas es la función **válido**:

```
fun válido (e: Tensayo): boolean
{ Función que devuelve true si el ensayo que recibe como parámetro es solución
al problema, es decir, si contiene algún candidato cuyo valor sea P. Devuelve
false en caso contrario }
    para c desde 0 hasta numCandidatos hacer
        candidato ← e.getCandidatos (c)
        si candidato = P entonces
            dev true
        fsi
    fpara
    dev false
ffun
```

El siguiente que veremos es la función **compleciones**, la más importante de los algoritmos de vuelta atrás:

```
fun compleciones (e: Tensayo): lista
{ Función que devuelve la lista de hijos correspondientes a un ensayo concreto.
La política de generación de hijos que seguiremos será la siguiente: para cada
candidato, compleciones genera todas las combinaciones posibles de éste con
cada uno de los demás, haciendo uso del conjunto de operaciones posibles }
    para c1 desde 0 hasta numCandidatos hacer
        para c2 desde c1 + 1 hasta numCandidatos hacer
            hijo = obtieneHijo (e, '+', c1, c2)
            si (no hijo.esVacio ()) entonces
                vhijos.addElement (hijo)
            fsi
            hijo = obtieneHijo (e, '-', c1, c2)
            si (no hijo.esVacio ()) entonces
                vhijos.addElement (hijo)
            fsi
            hijo = obtieneHijo (e, '*', c1, c2)
            si (no hijo.esVacio ()) entonces
                vhijos.addElement (hijo)
            fsi
            hijo = obtieneHijo (e, '/', c1, c2)
            si (no hijo.esVacio ()) entonces
                vhijos.addElement (hijo)
            fsi
        fpara
    fpara
ffun
```

Ahora especificaremos la función auxiliar *obtieneHijo* como sigue:

```
fun obtieneHijo (e: Tensayo, char operator, int c1Index, int c2Index): Tensayo
{ Función que copia la estructura del padre al hijo (apreciación del autor) }
    c1 ← getCandidato (c1Index)
    c2 ← getCandidato (c1Index)
    nuevoEnsayo ← e
    si (operator = '+') entonces
        res ← c1 + c2
    si no
        si (operator = '-') entonces
            res ← c1 − c2
        si no
            si (operator = '∗') entonces
                res ← c1 ∗ c2
            si no
                si (operator = '/') entonces
                    si (c2 ! = 0) ∧ (c1 ÷ c2 = 0) entonces
                        res ← c1 / c2
                    si no
                        res ← -1
                    fsi
                fsi
            fsi
        fsi
    fsi
    si (res ≥ 0) entonces
        nuevoEnsayo ← e
        pila1 = e.getOperacion (c1)
        pila2 = e.getOperacion (c2)
        pila = generaNuevaPila (pila1, pila2, operator)
        nuevoEnsayo.removeCandidato (c2)
        nuevoEnsayo.setCandidato (res, c1)
        nuevoEnsayo.removeOperacion (c2)
        nuevoEnsayo.setOperacion (pila, c1)
        dev nuevoEnsayo
    si no
        dev ensayoVacio
    fsi
ffun
```

Nótese cómo la función *generaNuevaPila* recibe como parámetros las dos pilas ya existentes (pertenecientes a cada uno de los candidatos), así como el operador que va a ser utilizado para combinar ambos candidatos y genera como resultado la nueva pila correspondiente al candidato generado.

La función **podar** es la siguiente:

```
fun podar (e: Tensayo): boolean
{ Función que devuelve un booleano dependiendo de si es posible continuar
explorando por el ensayo e que recibe como parámetro o no. La única condición
de poda que impondremos será que alguno de los candidatos calculados hasta el
momento sobrepase el valor de P }
    para c desde 0 hasta numCandidatos hacer
        candidato ← e.getCandidatos (c)
        si candidato > P entonces
            dev true
        fsi
    fpara
    dev false
ffun
```

La función **mejor** es:

```
fun mejor (e: Tensayo): Tensayo
{ Función que compara el ensayo e, que recibe como parámetro, con la solución
calculada hasta el momento. Devuelve a la salida aquél ensayo que contenga el
candidato más próximo a la solución solicitada }
    v1 ← valorMax (e)
    v2 ← valorMax (solución)
    si v1 < v2 entonces
        dev solución
    si no
        dev e
    fsi
ffun
```

Por último, la función auxiliar *valorMax* será:

```
fun valorMax (e: Tensayo): int
    value ← e.getCandidato (0)
    para cada c desde 1 hasta numCandidatos hacer
        valorAux ← e.getCandidato (c)
        si (valorAux > value) ∧ (valorAux ≤ P) entonces
            value = valAux
        fsi
    fpara
ffun
```

Febrero 2006-2ª (problema)

Enunciado: Dos socios que conforman una sociedad comercial deciden disolverla. Cada uno de los n activos que hay que repartir tiene un valor entero positivo. Los socios quieren repartir dichos activos a medias y, para ello, primero quieren comprobar si el conjunto de activos se puede dividir en 2 subconjuntos disjuntos, de forma que cada uno de ellos tenga el mismo valor. La resolución de este problema debe incluir por este orden:

1. Elección del esquema más apropiado y explicación de su aplicación al problema.
2. Descripción de las estructuras de datos necesarias.
3. Algoritmo completo a partir del refinamiento del esquema.
4. Estudio del coste del algoritmo desarrollado.

Respuesta:

Previamente a resolver el problema decir que en este caso lo que solicitan no corresponde exactamente con los 5 pasos que solemos poner en los ejercicios anteriores, por ello, el punto 2 (el de escribir el esquema general) lo incluiremos en el primero de ellos.

1. Elección del esquema más apropiado y explicación de su aplicación al problema

   No se puede encontrar una función de selección que garantice, sin tener que reconsiderar decisiones, una elección de los activos que cumpla con la restricción del enunciado, por ello NO se puede aplicar el esquema voraz. Tampoco se puede dividir el problema en subproblemas que al combinarlos nos lleven a una solución, por lo que se descarta divide y vencerás.

   Al no ser un problema de optimización, el esquema de exploración de grafos más adecuado es el esquema de **vuelta atrás.** Vuelta atrás es un recorrido en profundidad de un grafo dirigido implícito. En él, una solución puede expresarse como un n-tupla $[x_1, x_2, x_3, \dots, x_n]$, donde cada x_i representa una decisión tomada en la etapa i-ésima, de entre un conjunto finito de alternativas.

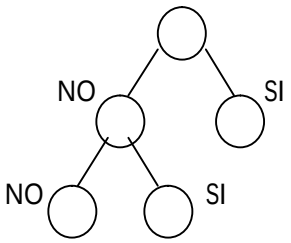
   Descripción algorítmica del esquema:

```
fun vuelta-atrás (e: ensayo) dev resultado
    si valido (e) entonces
        devolver e
    si no
        listaensayos ← compleciones (e)
        mientras no vacía (listaensayos) ∧ no resultado hacer
            hijo ← primero (listaensayos)
            listaensayos ← resto (listaensayos)
            si condiciones-de-poda (hijo) entonces
                resultado ← vuelta-atrás (hijo)
            fsi
        fmientras
        dev resultado
    fsi
ffun
```

Otra posible descripción:

```
fun vuelta-atrás (v[1..k]: nTupla)
    si valido (e) entonces
        devolver v
    si no
        para cada vector w (k-1)-prometedor hacer
            si condicionesDePoda (w) hacer
                vuelta-atrás (w[1..k + 1])
            fsi
        fpara
    fsi
ffun
```

En este caso, el espacio de búsqueda es un árbol de grado 2 y altura n + 1. Cada nodo del nivel i-ésimo nivel tiene dos hijos correspondientes a si el i-ésimo activo va de un socio o al otro. El árbol sería algo así:



Para poder dividir el conjunto de activos en dos subconjuntos disjuntos, de forma que cada uno de ellos tenga el mismo valor, su valor debe ser par. Así, si el conjunto inicial de activos no tiene un valor par, el problema no tiene solución.

2. Descripción de las estructuras de datos necesarias

- El conjunto de activos y sus valores se representan en un **array de enteros** $v = [v_1, v_2, v_3, \dots, v_n]$, donde cada v_i representa el valor del activo i-ésimo.
- La solución se representa mediante un **array de valores** $x = [x_1, x_2, x_3, \dots, x_n]$, donde cada x_i podrá tomar el valor 1 ó 2 en función de que el activo i-ésimo se asigne al subconjunto de un socio o del otro.
- Un **array de dos elementos suma**, que acumule la suma de los activos de cada subconjunto.

3. Algoritmo completo a partir del refinamiento del esquema general

En esta solución se sigue el segundo de los esquemas planteados en el apartado 1. Una posible condición de poda consistirá en que se dejarán de explorar aquellos nodos que verifiquen que alguno de los dos subconjuntos se van construyendo tiene un valor mayor que la mitad del valor total.

La función **solución** será la siguiente:

```
fun solución (k: entero; suma: array [1..2]) dev boolean
    si (k = n) AND (suma[1] = suma[2]) entonces
        dev verdadero
    si no
        dev falso
    fsi
ffun
```

Esta función equivale a esta otra, que pienso que es mucho más simple de entender y escribir:

```
fun solución (k: entero, suma: array [1..2]) dev boolean
    dev (k = n) AND (suma[1] = suma[2])
ffun
```

La función de vuelta atrás, que la denominaremos separacionSociosMitad sería la siguiente (personalmente el nombre es demasiado largo, pondría otro más corto acorde a la función en sí):

```
fun separacionSociosMitad (v: array [1..n], k: entero, suma: array [1..2],
                           sumaTotal: entero)
                           dev (x: array [1..n], separación: boolean)
    si solución (k, suma) entonces
        devolver x, verdadero
    si no
        para i desde 1 hasta 2 hacer
            x[k] ← 1
            suma[i] ← suma[i] + v[k]        /* v es el vector de valores */
            si suma[i] ≤ (sumaTotal DIV 2) entonces
                si k < n entonces
                    x ← separacionSociosMitad (v, k + 1, suma, sumaTotal)
                fsi
            si no
                suma[i] ← suma[i] − v[k]  /* Eliminamos el valor del socio */
            fsi
        fpara
    fsi
ffun
```

Por último, la **función principal** de nuestro algoritmo sería:

```
fun problemaSeparacionSociosMitad (v: nTupla)
                                  dev (x: nTupla, separación: boolean)
    para i desde 1 hasta n hacer
        sumaTotal ← sumaTotal + v[i]
    fpara
    si par (sumaTotal) entonces
        dev separacionSociosMitad (v, 1, suma[0,0], sumaTotal)
    si no
        dev 0
    fsi
ffun
```

La **llamada inicial** es problemaSeparacionSociosMitad (v).

NOTA DEL AUTOR: Además del problema visto anteriormente, que la función tiene el nombre demasiado largo, bajo mi punto de vista hay otro inconveniente que creo más importante y es que inicialmente se supone (o eso creo) que sumaTotal es 0, pero no está inicializado, por lo que pondría la línea de la inicialización justo antes del bucle "para".

4. Estudio del coste del algoritmo desarrollado

En este caso, el coste viene dado por el número máximo de nodos del espacio de búsqueda, esto es: $\boldsymbol{T(n) \in O(2^n)}$.

Febrero 2007-2ª (problema)

Enunciado: Disponemos de un conjunto A de números enteros (tanto positivos como negativos) sin repeticiones, almacenados en una lista. Dados dos valores enteros *m* y *C*, siendo $m < n$. Se desea resolver el problema de encontrar un subconjunto de A compuesto por exactamente *m* elementos y tal que la suma de los valores de esos *m* elementos sea C. la resolución de este problema debe incluir, por este orden:

1. Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema.
2. Descripción de las estructuras de datos necesarias.
3. Algoritmo completo a partir del refinamiento del esquema general.
4. Estudio del coste del algoritmo desarrollado.

Respuesta:

En este problema, como igualmente el anterior se ha resuelto en 4 puntos en lugar de en 5, como hemos visto previamente. Por ello, pondremos el esquema general en el punto primero. Pasamos, por tanto, a resolver el problema:

1. Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema.

   Selección del esquema algorítmico:

   Se trata de una búsqueda, no de una optimización, por lo que no son aplicables el esquema voraz ni el de ramificación y poda. Tampoco se puede dividir el problema en subproblemas que se resuelvan exactamente con el mismo procedimiento.

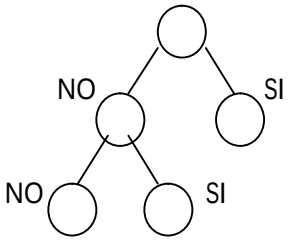
   Tenemos que recorrer un árbol de soluciones candidatas, pero siguiendo aquellas ramas que tengan opción de convertirse en una solución (k-prometedoras), por lo que el esquema de **vuelta atrás** es adecuado. También es válido un recorrido del árbol de soluciones en profundidad, siempre que se detenga en el nivel correspondiente a *m*.

   Dependiendo de la representación del ensayo y de la estrategia de ramificación (generación de hijos) tenemos al menos dos aproximaciones para el esquema de vuelta atrás.

   En una **primera aproximación** (que llamaremos A) podría representarse el ensayo como un vector de m enteros y los descendientes de un nodo de nivel k serian todos los valores no tomados anteriormente. Así del nodo raíz tendríamos n opciones para el nivel 1 del árbol, en el nivel 2 tendríamos $n * (n - 1)$ hijos, etc. hasta el nivel *m*. Como se verá más adelante, en el caso peor que es cuando *m* tiende a *n*, el número de nodos del árbol es $n!$

   La **segunda aproximación** (que llamaremos B) es más correcta y consistiría en representar el ensayo como un vector de n booleanos. De este modo los descendientes de un nodo de nivel *k* serían las condiciones de tomar o no el valor de k + 1 del vector de entrada. Es decir, siempre se tendrían dos opciones. Cuando *m* tiende a *n*, el número de nodos sería 2^n que es mejor que $n!$ Lo que está ocurriendo básicamente es que la segunda alternativa evita considerar permutaciones, es decir, tomar los mismos números pero en orden diferente. Además, esta segunda opción es más fácil de implementar.

El árbol de esta aproximación será:



(k-1)-prometedor

k-prometedor

El árbol se parece al del ejercicio de la división de activos (el ejercicio anterior), siendo la idea exactamente la misma.

A continuación, mostraremos la implementación de las dos alternativas. Consideremos los siguientes apartados para cada una de las alternativas:

Alternativa A (usando el esquema general de vuelta atrás):

1. Esquema general

El **esquema general** de vuelta atrás para nuestro problema es el siguiente:

```
fun vuelta-atrás (e: ensayo) dev (boolean, ensayo)
    si valido (e) entonces
        dev (cierto, e)
    si no
        listaensayos ← compleciones (e)
        mientras no vacía (listaensayos) y no es_solución hacer
            hijo ← primero (listaensayos)
            listaensayos ← resto (listaensayos)
            si esprometedora (hijo) entonces
                (es_solucion, solución) ← vuelta-atrás (hijo)
            fsi
        fmientras
        dev (es_solucion, solución)
    fsi
ffun
```

Las funciones que se tienen que especificar son las siguientes:

compleciones: Es una función que considera todos los números que aun no han sido probados como siguiente elemento.
esprometedora: Comprueba si se cumplen las condiciones de poda, es decir, será una función que compruebe que el número de sumandos ya seleccionados es menor que *m*.
valido: Comprueba que se haya alcanzado una solución.

2. Estructuras de datos

```
tipo ensayo = tupla
    candidatos: vector de enteros     // Números que aun no se han probado
    solucion: vector de enteros       // Números de la posible solución
    suma: entero                      // Valor alcanzado hasta ese momento
    num_sumandos: entero              // Cantidad de nºs que ya se han sumado
```

3. Algoritmo completo

Especificaremos en este apartado las funciones anteriores, siendo la primera de ellas la de **compleciones**:

```
fun compleciones (e: ensayo) dev lista_compleciones
    lista_compleciones ← lista_vacia
    para cada i en e.candidatos hacer
        nuevo_ensayo ← e
        eliminar (nuevo_ensayo.candidatos, i)
        añadir (nuevo_ensayo.solucion, i)
        nuevo_ensayo.suma ← e.suma + i
        nuevo_ensayo.num_sumados ← e.num_sumados + 1
        lista_compleciones ← añadir (lista_compleciones, nuevo_ensayo)
    fpara
    dev lista_compleciones
ffun
```

La segunda función es la denominada **esprometedora**:

```
fun esprometedora (e: ensayo) dev boolean
    si (e.num_sumados < m) entonces
        dev cierto
    si no
        dev falso
    fsi
ffun
```

Al igual que en ocasiones anteriores, se podrá reescribir esta función como sigue:

```
fun esprometedora (e: ensayo) dev boolean
    dev (e.num_sumados < m)
ffun
```

Por último, la función **válido** será:

```
fun valido (e: ensayo) dev boolean
    si (e.suma = C) ∧ (e.num_sumados = m) entonces
        dev cierto
    si no
        dev falso
    fsi
ffun
```

Podremos reescribirla de esta misma manera:

```
fun valido (e: ensayo) dev boolean
    dev (e.suma = C) ∧ (e.num_sumados = m)
ffun
```

4. Complejidad del algoritmo

Una **cota superior** al tiempo de ejecución sería el número de nodos del árbol, es decir, las combinaciones de n elementos tomadas de *m* en *m*, multiplicado por el número de posibles permutaciones, ya que éstas no se excluyen:

$$T(n,m) = \binom{n}{m} * m! = \frac{n!}{(n-m)! * m!} * m! = \frac{n!}{(n-m)!} = n * (n-1) * \dots * (n-m+1)$$

Llegamos al mismo resultado calculando el número de hijos de cada nivel del árbol: el número de hijos de cada nivel es $n * (n-1)$, hasta llegar al nivel $(n-m+1)$, cuyo número de hijos es $n * (n-1) * \dots * (n-m+1)$.

Podemos observar que

si $m \to n \Rightarrow T(n) \in O(n!)$
si $m \to 1 \Rightarrow T(n) \in O(n)$

Recordemos de los temas anteriores que el coste del algoritmo es muy malo, es de los peores, por lo que no es eficiente el algoritmo, aún resolviendo el problema.

Alternativa B (usando vectores k-prometedores)

1. Esquema general

Tendremos el esquema general mediante vectores k-prometedores:

```
fun vuelta-atrás (e: ensayo, k: entero) dev (boolean, ensayo)
    // e es k-prometedor
    si k = limite entonces
        dev (cierto, e)
    si no
        listaensayos ← compleciones (e, k + 1)
        mientras no vacía (listaensayos) y no es_solución hacer
            hijo ← primero (listaensayos)
            listaensayos ← resto (listaensayos)
            si esprometedora (hijo, k + 1) entonces
                (es_solucion, solución) ← vuelta-atrás (hijo, k + 1)
            fsi
        fmientras
        dev (es_solucion, solución)
    fsi
ffun
```

En la llamada inicial a vuelta-atrás, todas las posiciones de e.candidatos se inicializan a true y k toma el valor 0. Sería algo así (añadido del autor):

```
vuelta-atrás (ensayo_inicial, 0)
```

2. Estructuras de datos

Tendremos la siguiente estructura:

```
datos: vector de enteros                    // Lista de números de entrada
tipo ensayo = tupla
    candidatos: vector de booleanos         // Indica si el número de cada
                                               posición se incluye en la solución
    suma: entero                            // Valor alcanzado hasta ese
                                               momento
    num_sumados: entero                     // Cantidad de valores true de los
                                               candidatos
```

3. Algoritmo completo

Tal y como hemos visto antes veremos las funciones, que serían las mismas que las de la otra alternativa (recordemos que era compleciones, esprometedora y válido).

Empezaremos por **compleciones**:

```
fun compleciones (e: ensayo, k: entero) dev lista_compleciones
    lista_compleciones ← lista_vacia

    // El valor asignado a la posición k es true (hijo primero)
    nuevo_ensayo ← e
    nuevo_ensayo.suma ← e.suma + datos[k]
    nuevo_ensayo.num_sumados ← e.num_sumados + 1
    lista_compleciones ← añadir (lista_compleciones, nuevo_ensayo)

    // El valor asignado a la posición k es false (hijo segundo)
    nuevo_ensayo ← e
    nuevo_ensayo.candidatos[k] ← false
    lista_compleciones ← añadir (lista_compleciones, nuevo_ensayo)

    dev lista_compleciones
ffun
```

Como explicación del valor que toma el segundo hijo (aquél en el que el valor es *falso*), ya que hemos visto que en un principio se inicializan los valores de los valores del vector k-prometedor siempre a *verdadero*, de ahí que no se asigne ningún valor al primer hijo.

Las funciones esprometedora y válido son iguales a la alternativa A, aunque los pondremos, para completar el ejercicio:

Tendremos que la función **esprometedora** es:

```
fun esprometedora (e: ensayo) dev boolean
    si (e.num_sumados < m) entonces
        dev cierto
    si no
        dev falso
    fsi
ffun
```

la cual se podrá reescribir como sigue:

```
fun esprometedora (e: ensayo) dev boolean
    dev (e.num_sumados < m)
ffun
```

Por último, la función **válido** será:

```
fun valido (e: ensayo) dev boolean
    si (e.suma = C) ∧ (e.num_sumados = m) entonces
        dev cierto
    si no
        dev falso
    fsi
ffun
```

Podremos reescribirla de esta misma manera:

```
fun valido (e: ensayo) dev boolean
    dev (e.suma = C) ∧ (e.num_sumados = m)
ffun
```

4. Complejidad del algoritmo

En cada nivel al árbol se divide como máximo de 2 ramas (si no se ha llegado a tener *m* posiciones a true, en cuyo caso no haría falta crear ningún descendiente, sería el caso mejor). Luego, una **cota superior** es 2^n.

Podemos observar que

si $m \to n \Rightarrow T(n) \in O(2^n)$
si $m \to 1 \Rightarrow T(n) \in O(n)$

Este algoritmo es más eficiente que el anterior en el caso $m \to n$.

3ª parte. Problemas de exámenes sin solución o planteados:

Febrero 1997-1ª (problema 2)

Enunciado: El juego del 31 utiliza las cartas de la baraja española: 1, 2, 3, 4, 5, 6, 7, 10 (sota), 11 (caballo) y 12 (rey) con los 4 palos: oros, copas, espadas y bastos. Diseñar un algoritmo que calcule las posibles formas de obtener 31 utilizando a lo sumo 4 cartas y 2 palos distintos en cada combinación.

Respuesta: Este problema se resuelve usando el algoritmo de **vuelta atrás**, ya que no se nos pide un problema de optimización, si no resolver un problema dado. Descartaremos los otros esquemas, por no ser aplicable el esquema voraz, ya que no es posible encontrar una función de selección que nos permita llegar a solución óptima y el de divide y vencerás no se puede dividir el problema en subproblemas menores.

La dificultad que yo creo que tiene este problema (y todos los de este tema) es el ensayo, que en este caso sería algo así:

```
tipo ensayo = tupla
    baraja: arreglo[0..11] de boolean
    suma: entero
    palo: arreglo[0..3] de boolean
```

A continuación, creo que se resolvería de manera más o menos similar a los **juegos** antes vistos, teniendo en cuenta en el árbol asociado las distintas combinaciones desplegando el árbol con las posibles combinaciones. No acabaremos este ejercicio, pero lo dejamos planteado para su posterior resolución.

Septiembre 1997 (problema 2)

Enunciado: Sea un juego de tablero para dos personas, en la que se turnan para mover las piezas del tablero según unas reglas determinadas. Daremos por conocidos:

- Una estructura de datos *jugada* que nos da dos tipos de información: en un registro *tablero*, por un lado, la situación de las fichas en el tablero. Y en un registro *turno*, por otro lado, quién debe mover a continuación (o, en caso de ser una posición final, quién es el ganador), con la siguiente convención:

  1 significa que le toca mover al jugador que comenzó.
  2 al segundo jugador.
  3 que la partida acabo con triunfo del primer jugador, y
  4 que la partida acabó con triunfo del segundo jugador.

- Una función

  funcion movimientos (j:jugada) devolver l:lista de jugada

  que da todas las jugadas posibles a partir de una dada.

Supondremos que en el juego no se pueden dar ciclos (volver a una situación previa en el juego), que no se puede prolongar indefinidamente y que la partida no puede acabar en tablas. Diseñar un algoritmo que averigüe si existe una estrategia óptima para el jugador que mueve primero, de forma que se asegure la victoria siguiendo esta estrategia. ¿Se puede aplicar el algoritmo, en teoría, al juego del ajedrez? ¿Y en la práctica?

Respuesta: Este ejercicio es parecido al del juego de Nim (**vuelta atrás**), en el que se tenían que escoger una serie de cerillas, hasta que el jugador que gane se quede con la última. Por

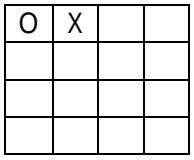
tanto, en este caso, tendremos que dos jugadores se turnan para mover las piezas (sería algo parecido al juego de Nim con las cerillas).

No vamos a detallar este algoritmo, ya que viene en el libro de Brassard en las páginas 319-326 (recalcamos de nuevo que es el del juego de Nim). Sólo contestar a la última pregunta, en las que se nos preguntaba si se puede aplicar este algoritmo para el ajedrez, siendo en teoría **si** aplicable para dicho juego, pero en la práctica **no**, ya que el árbol que se genera es demasiado grande para poder explorarlo todo.

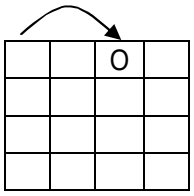
Febrero 1998-1ª (problema 1)

Enunciado: Se considera un juego solitario en el que el tablero se compone de varios huecos dispuestos en forma de cruz y un conjunto de bolas que, inicialmente, cubren todos ellos excepto el hueco central (ver figura izquierda). Siempre que haya dos bolas y un hueco vacio consecutivos, se puede saltar con la primera bola al hueco vacio, retirando la bola intermedia (ver un posible primer movimiento en la fig. derecha). Sólo se consideran movimientos horizontales o verticales, nunca en diagonal. El objetivo del juego es eliminar progresivamente todas las bolas del tablero hasta que sólo quede una. Diseñar un algoritmo que encuentre una solución al juego.

Respuesta: No se nos da ningún dibujo en el enunciado del examen, pero leyéndolo podremos deducir que el primero de ellos será:



En teoría y según el enunciado tendríamos que cubrir todo con bolas salvo el punto central, que es un vacio, pero para nuestro ejemplo podría valer para ver este juego. Por tanto, el segundo dibujo del que nos hablan seria cuando el O salta y se come al X, siendo ésto lo siguiente:



Como en juegos como el de Nim usaríamos el esquema de **vuelta atrás**, teniendo claro el número de hijos que generará el árbol, siendo en este caso las posibles jugadas que haga la ficha. Algo así, como si come la ficha o no. De todas maneras, no seguiremos haciendo este ejercicio, sólo lo plantearemos. Se deja como ejercicio su resolución.

Febrero 1999-1ª (problema 2)

Enunciado: Dado un mapa político de países, diseñar un algoritmo que coloree con 3 colores los países de manera que dos países fronterizos no tengan el mismo color.
¿Tiene este problema solución en todos los casos? Pon ejemplo si los hay.

Respuesta: Este problema es una especie de juego, en el que nos dan un mapa y hay que colorearlo. Descartaremos el esquema voraz, ya que en ocasiones, al poner el color, podremos arrepentirnos y volver atrás. En cambio, estaremos ante un empate entre el esquema de vuelta atrás y ramificación y poda, al no comentar nada de optimizar el problema (y además, generará pocos hijos, a lo sumo 3) creo que el mejor esquema es el de **vuelta atrás**. Al igual que en el ejercicio anterior no resolveremos el problema, ya que hemos visto una gama amplia bajo mi punto de vista y en este caso no está resuelto. Se dejaría, por tanto, para resolver.

Febrero 2000-1ª (problema) (parecido al problema 2 de Septiembre 1997)

Enunciado: Se considera un juego de tablero genérico con las siguientes características:

- Juegan dos oponentes entre sí. Cada jugador mueve alternativamente. El juego acaba con la victoria de uno de los dos jugadores, o en tablas.
- Se supone conocido:
  1. Una estructura de datos *situación* que nos da la situación actual del juego: qué jugador tiene el turno, disposición de las fichas en el tablero, etc.
  2. Una función *función movimientos (j:situación) devolver l:lista* que nos da todas las jugadas posibles a partir de una dada. Aceptaremos que, si nos devuelve una lista vacía, es porque el juego ha terminado.

Se pide diseñar un algoritmo que encuentre cuál es el número de movimientos de la (o las) partidas más cortas.

Respuesta: Este problema incluido el enunciado es parecido a dicho problema, por tanto, no lo resolveremos, sólo lo dejaremos señalado en referencia a dicho ejercicio. Es, por tanto, otro de **vuelta atrás**.

Febrero 2001-1ª (problema)

Enunciado: A orillas del rio Guadalhorce, se encuentran 3 alumnos y 3 profesores del tribunal de la UNED que deben cruzar el rio para ir juntos al Centro Asociado. Se dispone de una pequeña canoa de 2 plazas. Diseñar un algoritmo que encuentre una solución al problema de cruzar el rio considerando las siguientes restricciones:

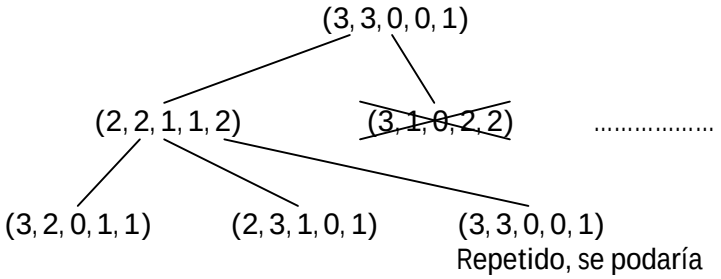
- La canoa no puede viajar sola.
- En ningún momento puede haber grupos en las orillas donde el número de alumnos supere al de los profesores.

Diseñar ahora el algoritmo para que calcule la solución con menor número de movimientos. Explica los cambios que hay que hacer al anterior algoritmo.

Respuesta:

En este problema tenemos dos partes, las cuales veremos:

En la primera parte se nos dice que hay una canoa en la que tiene que viajar, al menos, un alumno, ya que tiene que llevar a los profesores de una orilla a otra (se admiten bromas ;)) y nunca puede viajar sola. Este ejercicio nunca antes se ha visto y por lo que parece se resuelve usando una especie de grafo (no árbol, cuidado), como sigue:



Desarrollaremos este grafo como sigue significando cada campo lo siguiente:

- **Primer campo**: Número de alumnos de origen en la orilla dada.
- **Segundo campo**: Número de profesores de origen en la orilla dada.
- **Tercer campo**: Número de alumnos que llegan a la orilla dada.
- **Cuarto campo**: Número de profesores que llegan a la orilla dada.
- **Quinto campo**: Orilla actual, siendo 1 ó 2 el valor que tenga, según la orilla de la que partan. Suponemos que parten de la orilla 1 y se van de la 1 hasta la 2, así hasta que se complete el algoritmo.

Lo que hemos tachado es porque como hemos visto las condiciones del enunciado es que nunca viaje sola la canoa y encima la lleve un alumno, por lo que se descartará. Al igual que también la repetida, ya que llegaría a ser un grafo cíclico y nunca llegaríamos a solución (nodo destino), el cual sería llegar mediante transformaciones a $(3, 3, 0, 0, 2)$.

Por tanto, podríamos considerar este problema como el problema 2 de Septiembre 1996 (igual al 4.5 libro de problemas resueltos), en la que tendríamos una **variable global** llamada nodos-visitados, para así rechazar repeticiones como vimos en ese ejercicio.

Además, tendremos en cuenta que el nodo será el registro de esos campos distintos, los vistos anteriormente, que iremos modificando al avanzar el grafo (como apunte sólo hemos hecho dos niveles del grafo).

El segundo algoritmo sería una modificación de éste mismo, pero al ser uno de optimización deberíamos resolverlo usando **ramificación y poda**. No añadiremos esta parte en la sección adecuada (de ramificación y poda) por continuar con dicho ejercicio. No entraremos en la resolución del problema, por lo que lo dejaremos más o menos planteado. Como en ocasiones anteriores habría que ver las cotas superiores, cotas inferiores, costes de los nodos,…

Septiembre 2001 (problema)

Enunciado: Se plantea el siguiente juego: Tenemos un rectángulo (tablero) de *n* filas y *m* columnas con *n x m* casillas y suponemos que el tablero descansa sobre uno de sus lados de tamaño *m* al que denominaremos *base*, y donde cada casilla puede contener símbolos de k tipos diferentes. Al comienzo del juego el usuario comienza con un tablero lleno de símbolos. En cada movimiento el jugador elige un símbolo del tablero y éste es eliminado del mismo junto con (si las hay) las agrupaciones adyacentes (en horizontal o vertical) de su mismo símbolo siempre que mantengan la continuidad. Una vez eliminada la agrupación, los huecos dejados por ésta se rellenan deslizando hacia la *base* los símbolos de su columna que queden por encima del hueco dejado por los símbolos eliminados. El objetivo del juego es el de dejar el tablero vacío.

Para ilustrar el enunciado supongamos un tablero de 4 filas y 5 columnas con 4 símbolos o, +, ∗, x y donde – representa una casilla vacía. Una secuencia del juego sería la siguiente:

oox+x	--x+x	---+x	-----	-----	-----	-----	-----
o++x+	--+x+	---x+	---+-	-----	-----	-----	-----
o**ox	-+*ox	--xox	--xxx	---+-	---+-	-----	-----
oo*xx	-**xx	-++xx	-++0+	-++o+	---o+	--++-	-----

Se pide programar un algoritmo que resuelva el juego. Explicar además (si las hay) las diferencias que habría de introducir a este algoritmo si se exigiera resolver el juego en el menor número de movimientos.

Respuesta: Este problema al igual que pasa en ocasiones corresponde con dos tipos distintos, donde la primera parte es de vuelta atrás, ya que solicitan resolver el juego y como hemos visto en innumerables ocasiones, se resolverían de esta manera.

Por otro lado, nos piden en la segunda parte que se resuelva con el menor número de movimientos, por lo que podremos hacerlo o bien usando **recorrido en anchura** o bien **ramificación y poda**. Siendo coherente escogería el segundo tipo, ya que creo que es el que más se adecua al problema en cuestión.

Febrero 2002-2ª (problema)

Enunciado: Se dispone de un tablero de 8 casillas similar al que muestra la figura. En la situación inicial, las casillas están ocupadas por cuatro O y cuatro X distribuidas de forma aleatoria. En cada paso, la única operación posible es intercambiar dos fichas adyacentes. Programar de la forma más eficiente posible el algoritmo que devuelva la secuencia de pasos más corta para ordenar el tablero de forma que todas las fichas tengan al menos otra igual adyacente.

X	O	X	X	O	O	O	X

Respuesta: Al pedirnos la secuencia de pasos más cortas podremos estar hablando de una **búsqueda en anchura** o bien de un esquema de **vuelta atrás**, por extensión podría ser incluso uno de ramificación y poda, pero en este caso el árbol que se expande es pequeño. Como hemos visto antes podremos hacerlo de cualquiera de las dos maneras, dependerá del gusto del alumno en cuestión.

Septiembre 2002 (problema)

Enunciado: Escribir un algoritmo que descubra cómo encontrar el camino más corto desde la entrada hasta la salida de un laberinto (suponer que hay sólo una entrada y sólo una salida).

Respuesta: Se nos pide que encontremos el camino más corto desde la entrada hasta la salida de un laberinto, por lo que tendremos dos modos de resolverlo, o bien usando un *recorrido en anchura* o bien un algoritmo de *ramificación y poda*. Del primero decir que encontraba el camino más cercano a la raíz y del segundo, como es habitual, será un problema de minimización del camino, es decir, de las celdas recorridas. Dejamos señalado en este caso también en el problema lo que usaremos sin llegar a resolverlo.

Febrero 2004-1ª (problema)

Enunciado: Se tiene una matriz ***n x n*** cuyas casillas contienen valores del conjunto $C = \{rojo, verde, amarillo\}$ excepto una que es 'blanco'. El único movimiento permitido consiste en que una casilla de color 'blanco' puede intercambiarse por cualesquiera de sus adyacentes no diagonales. Una solución es una tabla con una fila llena de valores iguales. Desarrollar un algoritmo que dada una matriz inicial averigüe cuántas soluciones se pueden encontrar aplicando movimientos permitidos a partir de aquella.

Respuesta: Tendremos que resolver este problema usando un algoritmo de **vuelta atrás**, en la que cada nodo tiene 3 hijos, correspondiente con los colores. Este problema es similar los ejercicios de colorear un mapa político, que hemos visto previamente. Igualmente, no entraremos a resolverlo, sólo lo plantearemos levemente.

Septiembre 2005-reserva (problema)

Enunciado: El Sudoku es un pasatiempo consistente en rellenar con cifras del 1 al 9 una cuadricula de 81 (9*9) casillas distribuidas a su vez en 9 cajas de 3 x 3. El juego consiste en rellenar cada caja de 9 casillas con cifras del 1 al 9 sin repetirlas. No se puede repetir tampoco cifras en líneas o columnas de la cuadricula. Se pide diseñar un algoritmo que complete por nosotros este pasatiempo. La tabla adjunta muestra un ejemplo resuelto

9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	6	1	7	3

Respuesta: Este ejercicio corresponde con la práctica del año 2005-06 (la primera que hice ;)), que se resolvía usando el esquema de vuelta atrás. Tendremos que ver que las condiciones de poda son aquellas en las que al poner un número no se puede repetir tanto en fila, columna como en el cuadrante, por lo que habría que volver atrás y seleccionar el siguiente número. Para hacer este ejercicio mejor es ver dicha práctica.

Septiembre 2006 (problema)

Enunciado: Desarrollar un programa que compruebe si es posible que un caballo de ajedrez mediante una secuencia de sus movimientos permitidos recorra todas las casillas de un tablero N x N a partir de una determinada casilla dada como entrada y sin repetir ninguna casilla. Se pide:

1. Determinar qué esquema algorítmico es el más apropiado para resolver el problema. Razonar la respuesta y escribir el esquema general.
2. Indicar qué estructuras de datos son necesarias.
3. Desarrollar el algoritmo completo en el lenguaje de programación que utilizaste para la práctica.
4. Hallar el orden de complejidad del algoritmo desarrollado.

Respuesta: Este ejercicio es similar al **problema 4.3** del libro de problemas, sólo que lo trataremos de manera separada a dicho problema, por tener un enunciado algo distinto. Por tanto, se resuelve de la misma manera que dicho ejercicio. Dejaríamos, por ello, el ejercicio para que se resuelva, aunque ya lo hemos visto previamente.

Septiembre 2007-reserva (problema)

Enunciado: La agencia matrimonial Celestina & Co. quiere informatizar parte de la asignación de parejas entre sus clientes. Cuando un cliente llega a la agencia se describe a sí mismo y como le gustaría que fuera su pareja. Con la información de los clientes la agencia construye dos matrices M y H que contienen las preferencias de los unos por los otros, tales que la fila M[i, .] es una ordenación de mayor a menor de las mujeres cliente según las preferencias del i-ésimo hombre, y la fila H[i, .] es una ordenación de mayor a menor de los hombres cliente según las preferencias de la i-ésima mujer. Por ejemplo, M[i, 1] almacenaría a la mujer preferida por i y M[i, 2] a su segunda preferida. Dado el alto índice de divorcios, la empresa se ha planteado como objetivo que los emparejamientos sean lo más estables posibles evitando las siguientes situaciones:

1. Que dada una pareja (h', m') se dé el caso que m' prefiera a un h sobre h' y además h' prefiera a un m sobre m'.
2. Que dada una pareja (h'', m'') se dé el caso que h'' prefiera a un m sobre m'' y además m prefiera a h sobre h''.

La agencia quiere que dadas las matrices de preferencia un programa establezca parejas evitando las dos situaciones descritas con anterioridad. La resolución de este problema debe incluir, por este orden:

1. Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema.
2. Descripción de las estructuras de datos necesarias.
3. Algoritmo completo a partir del refinamiento del esquema general.
4. Estudio del coste del algoritmo desarrollado.

Respuesta: Este ejercicio, como los de esta parte no lo vamos a hacer, lo único que este enunciado en especial está sacado del libro de *técnicas de diseños de algoritmos* realizado por la Universidad de Málaga (no sé el autor), sería el correspondiente al ejercicio 6.5. En dicho ejercicio al igual que los que hemos visto previamente tendremos que las condiciones de poda nos las suministra el enunciado con las dos posibles situaciones que no deberían ocurrir. Dicho esto, se debería expandir el grafo para así ver mejor lo que debería hacer el algoritmo.

Ramificación y poda

Introducción teórica:

Tal y como hemos dicho en los esquemas de vuelta atrás pondremos de modo separado el de ramificación y poda para tratarlo más adecuadamente de este modo, ya que es **importantísimo** el controlarlo adecuadamente. Se observará a continuación que cuestiones de exámenes no hay muchos, ya que dan más importancia a los problemas, puestos en los últimos exámenes (de 2008).

- Ramificación y poda:

  El **segundo esquema** será aquel en el que el cálculo de las cotas se utilizan para seleccionar el camino que, entre los abiertos, parezca más prometedor para explorarlo primero y además como el esquema anterior para podar ramas. Será el que más empleemos en los distintos ejercicios que se nos den. Tenemos el algoritmo:

```
fun ramificación-y-poda (ensayo)
    m ← montículo-vacío
    cota-superior ← inicializar-cota-superior
    solución ← solución-vacía
    añadir-nodo (ensayo, m)
    mientras no vacío (m) hacer
        nodo ← extraer-raíz (m)
        si valido (nodo) entonces
            si coste (nodo) < cota-superior entonces
                solución ← nodo
                cota-superior ← coste (nodo)
            fsi
        si no                   { Nodo no es válido (solución) }
            si cota-inferior (nodo) ≥ cota-superior entonces
                devolver solución
            si no               { cota-inferior (nodo) < cota-superior }
                para cada hijo en compleciones (nodo) hacer
                    si condiciones-de-poda (hijo) y
                    cota-inferior (hijo) < cota-superior entonces
                        añadir-nodo (hijo, m)
                    fsi
                fpara
            fsi
        fsi
    fmientras
ffun
```

Tenemos estas nuevas funciones y variables:

- **Cota-superior**: Será, en cada momento, el coste de la mejor solución encontrada hasta el momento. En el esquema anterior es una pequeña modificación respecto al mismo, en la que llaman c a esta variable, pero conceptualmente es similar.
- **Cota-superior-inicial**: Será aquella cota que en un primer momento estimemos. Podrá ser tanto un valor muy alto, que luego haga podar menos ramas tanto un valor cercano a la solución, en todo caso, dependerá del tipo del problema en cuestión.
- **Función cota-inferior (nodo)**: Será aquel valor de cota en el nodo que se estime para alcanzar la solución.
- **Función coste (nodo)**: Será aquel valor del nodo una vez alcanzada la solución. Podrá mejorar al valor de la cota-superior, ante lo cual se actualiza esta última.

En este caso usaremos una estructura de datos *montículo* que nos hará escoger siempre el nodo más prometedor (lo usaremos como una lista con prioridad). Es el mismo esquema que previamente, sólo que añadimos la selección del camino que nos lleve antes a solución.

Se añaden un par de líneas en este esquema que son:

si cota-inferior (nodo) ≥ cota-superior entonces
    devolver solución

Esto significará que cuando en un nodo tengamos una cota-inferior (recordemos que es la estimación hasta encontrar la solución) igual o mayor a la cota-superior (que es el coste de la mejor solución encontrada hasta el momento) entonces no podremos llegar por ningún otro nodo del montículo a una solución mejor, por lo que dejamos de explorar el resto del grafo implícito (devolvemos la solución mejor y salimos del bucle).

Los dos esquemas anteriores para **problemas de minimización**, ya que iremos rebajando la cota superior una vez encontrada la solución si la mejora (podaremos más ramas).

NOTA DEL AUTOR: Al igual que con el vuelta atrás tenemos más esquemas de ramificación y poda, que estarán en el resumen del tema.

1ª parte. Cuestiones de exámenes:

Aunque en las cuestiones de vuelta atrás hemos puesto varios ejercicios relacionados de modo indirecto con ramificación y poda, hemos encontrado sólo este que es únicamente de este esquema. Pasamos a verlo, por tanto.

Febrero 2000-2ª (ejercicio 2)

Enunciado: ¿Puedes explicar brevemente qué papel desempeña un montículo en el algoritmo de ramificación y poda? ¿Cuál es el criterio general para realizar la poda en este algoritmo?

Respuesta: El **montículo** es la forma ideal de mantener la lista de nodos que han sido generados pero no explorados en su totalidad en los diferentes niveles del árbol, tomando como valor el de la *función de cota*. De este modo, tendremos ordenados los nodos según la cota, de modo que el elemento en la cima será el más factible (prometedor) de ser explorado a continuación (sería algo así como una lista de prioridad).

El criterio consiste en **calcular una cota** del posible valor de aquellas soluciones que el grafo pudiera tomar más adelante, si la cota muestra que cualquiera de estas soluciones será necesariamente peor que la mejor solución hallada hasta el momento entonces no seguimos explorando esa parte del grafo.

2ª parte. Problemas de exámenes solucionados:

Tendremos en cuenta los pasos a seguir para resolver problemas:

- Elección del esquema (voraz, vuelta atrás, divide y vencerás).
- Identificación del problema con el esquema.
- Estructura de datos.
- Algoritmo completo (con pseudocódigo).
- Estudio del coste.

Vamos a ver los problemas resueltos de exploración en profundidad, anchura y vuelta atrás, siendo estos últimos los más importantes y más comúnmente puestos en ejercicios de exámenes. Separaremos, por tanto, los esquemas de ramificación y poda.

Representaremos las funciones en pseudocódigo empleando cualquiera de las dos técnicas en los parámetros: escribiendo como una variable de la estructura de datos (por ejemplo, e:ensayo) o bien como la propia estructura de datos (por ejemplo, ensayo). Se verán ejemplos más adelante donde usaremos ambas técnicas (o casos) de modo indiferente, ya que la solución es siempre la misma, siendo, por tanto, su uso indiferente.

Otro asunto que es importante destacarlo es que no podremos punto y coma (;) en todas las sentencias, aunque lo ideal es que se ponga. Es importante saberlo, ya que nuestro propósito es didáctico y siendo estrictos habría que ponerlo siempre.

Septiembre 1997 (problema 1) (igual a 4.6 libro de problemas resueltos)

Enunciado: Una empresa de mensajería dispone de 3 motoristas en distintos puntos de la ciudad y tiene que atender a 3 clientes en otros 3 puntos. Se puede estimar el tiempo que tardaría cada motorista en atender a cada uno de los clientes (en la tabla, en minutos):

	Moto 1	Moto 2	Moto 3
Cliente 1	30	40	70
Cliente 2	60	20	10
Cliente 3	40	90	30

Diseñar un algoritmo que distribuya un cliente para cada motorista de forma que se minimice el coste total (en tiempo) de atender a los 3 clientes.

Respuesta:

NOTA: Hay que prestar especial cuidado con estos ejercicios, debido a que son básicos de ramificación y poda. De hecho, se ven como tipo de problema en la teoría de este tema.

1. Elección razonada del esquema algorítmico

   Se trata de un **problema de optimización**; sin embargo, no parece existir ningún criterio que nos permita ir asignando tareas a mensajeros sin tener que deshacer opiniones; por tanto, hemos de desestimar el esquema voraz a favor de una exploración ciega de un árbol de búsqueda. Cada nodo del árbol consistirá en una asignación parcial de tareas a mensajeros; las hojas de árbol serán aquellos nodos en los que las tres tareas estén repartidas entre los tres mensajeros. El *nodo raíz* será un nodo vacio sin asignaciones.

   En este caso, el esquema más útil es el de **ramificación y poda**. Para cada asignación parcial de tareas, podemos estimar una cota inferior al coste total en tiempo, sumando el coste de las tareas asignadas de cada una de las tareas no asignadas (30, 10 y 20, respectivamente).

Por ejemplo, si sólo tenemos asignado el cliente 1 al motorista 2 con un coste de 40 minutos, cualquier solución obtenida a partir de esta asignación parcial tendrá un coste igual o peor a 40 + 20 + 20 = 80 min.

En lugar de explorar el árbol en anchura o profundidad, examinaremos en cada momento, el **nodo más prometedor**, es decir, aquella cuya cota inferior sea mínima (recordemos que es como una *lista con prioridad*). Para ello, los almacenaremos según un montículo de mínimos que nos de siempre, en la raíz, el nodo más prometedor. Cuando la cota inferior más baja sea superior a una solución obtenida, el algoritmo habrá terminado, puesto que no es posible encontrar soluciones mejores.

2. Descripción del esquema usado e identificación con el problema

El esquema de ramificación y poda de búsqueda ciega en un árbol, en la que se busca una solución óptima de acuerdo a un criterio dado (minimización de una variable). Se dispone de una **cota superior global**, que coincide en cada momento con la mejor solución encontrada hasta el momento, y de una forma de asignar cotas inferiores a cada nodo (ninguna solución obtenida a partir de ese nodo mejor que su cota inferior).

El algoritmo se distingue porque aquellas ramas cuya **cota inferior** sea mayor que la cota superior global pueden abandonarse sin ser exploradas. Además, se puede recorrer el árbol expandiendo en cada momento la rama más prometedora (la de menor cota inferior), estrategia que reduce el número de nodos que es necesario visitar.

El **esquema general** de ramificación y poda correspondiente a aquél en el que se podan ramas y se selecciona el camino (el segundo de los vistos en el resumen del tema):

```
fun ramificación-y-poda (ensayo)
    m ← montículo-vacío
    cota-superior ← inicializar-cota-superior
    solución ← solución-vacía
    añadir-nodo (ensayo, m)
    mientras ¬vacío (m) hacer
        nodo ← extraer-raíz (m)
        si válido (nodo) entonces
            si coste (nodo) < cota-superior entonces
                solución ← nodo
                cota-superior ← coste (nodo)
            fsi
        si no                   { Nodo no es válido (solución) }
            si cota-inferior (nodo) ≥ cota-superior entonces
                devolver solución
            si no               { cota-inferior (nodo) < cota-superior }
                para cada hijo en compleciones (nodo) hacer
                    si condiciones-de-poda (hijo) y
                    cota-inferior (hijo) < cota-superior entonces
                        añadir-nodo (hijo, m)
                    fsi
                fpara
            fsi
        fsi
    fmientras
ffun
```

En nuestro caso, tendremos estas variables y funciones:

- **Cota-inferior (nodo)**: Devolverá la suma de los costes de las tareas asignadas y los costes mínimos de las no asignadas.
- **Cota-superior**: Será, en cada momento, el coste de la mejor solución encontrada.
- **Cota-superior-inicial**: Puede ser una solución cualquiera. Por ejemplo, la diagonal principal: 30 + 20 + 10 = 60.
- **Compleciones**: Pueden funcionar tomando la primera tarea sin asignar y generando tantos nodos como mensajeros disponibles haya para esa tarea.
- **Condiciones-de-poda (nodo)**: Si hemos generado sistemáticamente los nodos mediante la función compleciones mencionada, ya no es necesario considerar ninguna condición en particular (como puede ser que no haya dos mensajeros efectuando la misma tarea).

3. Estructura de datos

El tipo ensayo podría ser simplemente un vector de naturales de tamaño 3, identificando la posición en el vector con un mensajero y el valor del elemento que ocupa con la tarea que se ha asignado. Sin embargo, optamos por una estructura más rica que nos evite repetir cálculos:

```
ensayo = tupla
    asignaciones: vector [1..3] de natural
    tareas-no-asignadas: lista de natural
    último-mensajero-asignado: natural
    coste: natural
```

4. Algoritmo completo a partir del refinamiento del esquema general

Las funciones que hemos de especificar, tal y como hemos visto en el ejercicio anterior (es decir, válido, compleciones y condiciones-de-poda). Veremos, en esta ocasión la primera, que es la más importante (**compleciones**):

```
fun compleciones (e: ensayo) dev lista de ensayos

    matriz-de-costes ← ⎛30  40  70⎞
                       ⎜60  20  10⎟
                       ⎝40  90  20⎠

    lista-compleciones ← lista vacía
    para cada tarea en e.tareas-no-asignadas hacer
        hijo ← e
        hijo.ultimo-mensajero-asignado ← e.ultimo-mensajero-asignado + 1
        hijo.asignaciones ← e.asignaciones
        hijo.asignaciones[hijo.ultimo-mensajero-asignado] ← tarea
        hijo.coste ← e.coste + matriz-de-costes[tarea, mensajero]
        hijo.tareas-no-asignadas ← e. tareas-no-asignadas
        eliminar (tarea, hijo.tareas-no-asignadas)
        añadir (hijo, lista-compleciones)
    fpara
    dev lista-compleciones
ffun
```

NOTA DEL AUTOR: Esta función es una variación respecto a la dada en el ejercicio en especial. Se ha tomado la dada en el problema de Febrero de 2006-1ª semana y modificada, ya que estimaba que la solución era incorrecta. Este ejercicio en cuestión lo trataremos más adelante, debido a que es interesante verlo con más detenimiento.

La función que calcula la **cota inferior** de los nodos sería:

```
fun cota-inferior (e: ensayo) dev natural
    vector-de-mínimos ← [30,10,20]
    dev (e.coste + ∑_{i=e.ultima−asignacion + 1}^{3} vector-de-mínimos[i])
ffun
```

Recordemos que esta función era propia del esquema de ramificación y poda, donde se calculaba la estimación para llegar a la solución. Al ser un esquema de minimización deberemos escoger la mejor cota inferior que nos lleve a la solución y actualizar la cota superior al encontrar solución.

La función **válido** será la siguiente:

```
fun válido (e: ensayo) dev boolean
    dev (e.ultimo-mensajero-asignado = 3)
ffun
```

No se va a poner la función **condiciones-de-poda**, en este caso, ya que no se considerará ninguna condición en especial que pode ninguna rama. Lo hemos explicado anteriormente, pero de nuevo lo ponemos de nuevo insistiendo en este asunto.

Por último, tras estas funciones dadas, el algoritmo general modificado quedará así:

```
fun tareas-y-mensajeros (ensayo)
    m ← montículo-vacío
    cota-superior ← 70
    solución ← solución-vacía
    añadir-nodo (ensayo, m)
    mientras ¬vacío (m) hacer
        nodo ← extraer-raíz (m)
        si válido (nodo) entonces
            si coste (nodo) < cota-superior entonces
                solución ← nodo
                cota-superior ← coste (nodo)
            fsi
        si no                   { Nodo no es válido (solución) }
            si cota-inferior (nodo) ≥ cota-superior entonces
                devolver solución
            si no               { cota-inferior (nodo) < cota-superior }
                para cada hijo en compleciones (nodo) hacer
                    si cota-inferior (hijo) < cota-superior entonces
                        añadir-nodo (hijo, m)
                    fsi
                fpara
            fsi
        fsi
    fmientras
ffun
```

Se ha hecho pocas modificaciones con respecto al esquema original, por lo que la importancia este tipo de problemas se ve claramente.

5. Análisis del coste

En principio, el algoritmo tiene un coste constante, puesto que el tamaño del problema es constante. Sin embargo, introduciendo cambios mínimos el algoritmo sirve para resolver, en general, asignaciones de n tareas a n mensajeros. En este caso, una estimación del coste es el tamaño del árbol de búsqueda, que crece como $\boldsymbol{O(n!)}$, ya que cada nodo del nivel *k* puede expandirse con los $n - k$ mensajeros restantes sin asignar.

Septiembre 2004 (problema)

Enunciado: Sean dos vectores de caracteres. El primero se denomina *texto* y el segundo *consulta*, siendo éste de igual o menor longitud que el primero. Al vector consulta se le pueden aplicar, cuantas veces sea necesario, los siguientes tres tipos de operaciones, que añaden cada una de coste diferente a la edición de la consulta:

1. *Intercambio* de dos caracteres consecutivos: coste de edición igual a 1.
2. *Sustitución* de un carácter por otro cualquiera: coste de edición igual a 2.
3. *Inserción* de un carácter cualquiera en una posición cualquiera: coste de edición igual a 3.

Implementar una función que escriba la secuencia de operaciones con menor coste total que hay que realizar sobre el vector consulta para que coincida exactamente con el vector texto. La función devolverá el coste total de estas operaciones, es decir, la suma de los costes asociados a cada operación realizada.

Respuesta:

1. Elección del esquema algorítmico

   La solución es el resultado de una secuencia de pasos o decisiones que **no** se puede establecer un criterio óptimo de selección para cada decisión por lo que no puede ser un esquema voraz. Por tanto, hay que realizar una exploración de las posibles soluciones y buscar la óptima. Debido a que hay un criterio de optimalidad, es decir, existe una función que decide si un nodo puede llevar a una solución mejor que la encontrada hasta el momento, el esquema adecuado es de **ramificación y poda**.

2. Descripción del esquema

El esquema que se da en este ejercicio está algo modificado respecto al que hemos visto anteriormente:

```
fun ramificaciónPoda (nodo_raiz) dev nodo
    inicializarSolucion (Solución, valor_sol_actual)
    Montículo := monticuloVacio()
    cota := acotar (nodo_raiz)
    poner ((cota, nodo_raiz), Montículo)
    mientras no vacio (Montículo) hacer
        (cota, nodo) := quitarPrimero (montículo)
        si es_mejor (cota, valor_sol_actual) entonces
            si válido (nodo) entonces
            /* cota es el valor real de la mejor solución hasta ahora */
                valor_sol_actual := cota
                Solución := nodo
            si no
                para cada hijo en compleciones (nodo) hacer
                    cota := acotar (hijo)
                    poner ((cota, hijo), Montículo)
                fpara
            fsi
        si no
            /* Termina el bucle, la solución actual no tiene mejor cota */
            devolver Solución
        fsi
    fmientras
    devolver Solución
ffun
```

NOTA DEL AUTOR: De nuevo se insiste en finalizar las sentencias con punto y coma en pseudocódigo, cosa que por el momento no hacemos (en este tema, al menos). Está función es otro ejemplo del uso de := en vez de ←, que como hemos comentado daría exactamente el mismo resultado.

3. Estructuras de datos

Usaremos lo siguiente:

- Vector de texto.
- Vector de consulta.
- Montículo de mínimos en el que cada componente almacena una solución parcial (nodo) con su cota correspondiente.
- nodo = tupla

    acciones: lista de Strings;
    ultimo_coincidente: cardinal;
    long: cardinal;
    coste: cardinal;

4. Algoritmo completo

Veremos, paso por paso las especificaciones de las funciones, que nos fijaremos en las explicaciones de cada una de ellas.

inicializarSolucion(): La solución inicial debe tener un coste asociado mayor o igual que la solución final. De acuerdo con el problema, basta con realizar **sustituciones** de los caracteres de la consulta (coste = 2 * long_consulta) y realizar la **inserción** de los caracteres que faltan (coste = 3 * (long_texto – long_consulta)). Además, tenemos que construir la solución inicial: la secuencia de sustituciones e inserciones.

```
fun inicializarSolucion (long_consulta, texto, long_texto, Solución, valor_sol_actual)
    valor_sol_actual := 2 * long_consulta + 3 * (long_texto – long_consulta)
    Solución := listaVacia()
    para i desde 1 hasta long_consulta hacer
        añadir ("sustituir i por texto[i]", Solución)
    fpara
    para i desde long_consulta + 1 hasta long_texto hacer
        añadir ("insertar en i por texto[i]", Solución)
    fpara
ffun
```

válido (nodo): Un nodo será válido como solución si se han hecho coincidir los long_texto caracteres de texto. Para ello, vamos a ir haciendo coincidir la solución desde el principio del vector hasta el final. Por tanto, un nodo será válido (aunque no sea la mejor solución todavía) si:

```
nodo.ultimo_coincidente == long_texto
```

es_mejor (cota, valor_sol_actual): Como se trata de encontrar la solución de menor coste, una cota será mejor que el valor de la solución actual si:

```
cota < valor_sol_actual
```

acotar (nodo): Se trata de realizar una estimación que sea mejor o igual que la mejor solución que se puede alcanzar desde ese nodo. De esta manera sabremos que si, aún así, la estimación es peor que la solución actual, por esa rama no encontraremos nada mejor y se puede podar. La mejor estimación será sumar el coste ya acumulado, el menor coste que podría completar la solución: que el resto de caracteres de consulta coincidieran (coste = 0) y realizar tantas inserciones como caracteres nos falten (coste = 2 * (long_texto – nodo.long)).

Es decir, acotar (nodo) es:

```
nodo.coste + 2 * (long_texto – nodo.long)
```

compleciones (nodo): Sabiendo que hasta nodo.ultimo_coincidente todos los caracteres coinciden, se trata de considerar las **posibilidades** para que el siguiente carácter de la consulta coincida con el texto:

- No hacer nada si ya coinciden de antemano.
- Intercambiarlo por el siguiente si éste coincide con el texto.
- Sustituirlo por el correspondiente del texto.
- Insertar el correspondiente del texto y correr el resto del vector, siempre que la consulta no haya alcanzado el tamaño del texto.

Si se da el *primer caso*, no es necesario generar el resto de compleciones porque no se puede encontrar una solución mejor con ninguna de las otras alternativas. Aún así, no pasaría nada si se incluyen. Sin embargo, para el *resto de alternativas* no hay garantías de que una permita encontrar mejor solución que otra, aunque el coste de la acción puntual sea menor. Por tanto, hay que incluir todas las alternativas.

La función será:

```
fun compleciones (nodo, texto, long_texto, consulta) dev lista-nodos
    lista := crearLista()
    si consulta [hijo.ultimo_coincidente] == texto [hijo.ultimo_coincidente]
    entonces
        hijo := crearNodo()
        hijo.ultimo_coincidente := nodo.ultimo_coincidente + 1
        hijo.acciones := nodo.acciones
        hijo.coste := nodo.coste
        hijo.long := nodo.long
        añadir (hijo, lista)
    si no
        /* Intercambio */
        si consulta [hijo.ultimo_coincidente+1]==texto [hijo.ultimo_coincidente]
        entonces
            intercambiar (consulta [hijo.ultimo_coincidente],
                          consulta [hijo.ultimo_coincidente+1])
            hijo := crearNodo()
            hijo.ultimo_coincidente := nodo.ultimo_coincidente + 1
            hijo.acciones := nodo.acciones
            añadir ("intercambiar consulta [hijo.ultimo_coincidente]
                     por consulta [hijo.ultimo_coincidente+1]", hijo.acciones)
            hijo.coste := nodo.coste + 1
            hijo.long := nodo.long
            añadir (hijo, lista)
        fsi
        /* Sustitución */
        hijo := crearNodo()
        hijo.ultimo_coincidente := nodo.ultimo_coincidente + 1
        hijo.acciones := nodo.acciones
        insertar ("sustituir consulta [hijo.ultimo_coincidente]
                   por consulta [hijo.ultimo_coincidente+1]", hijo.acciones)
        hijo.coste := nodo.coste + 2
        hijo.long := nodo.long
        añadir (hijo, lista)
        /* Inserción */
        si (nodo.long < long_texto) entonces
            hijo := crearNodo()
            hijo.ultimo_coincidente := nodo.ultimo_coincidente + 1
            hijo.acciones := nodo.acciones
            insertar ("insertar en hijo.ultimo_coincidente
                       texto [hijo.ultimo_coincidente]", hijo.acciones)
            hijo.coste := nodo.coste + 3
            hijo.long := nodo.long + 1
            añadir (hijo, lista)
        fsi
    fsi
    devolver lista
ffun
```

NOTA DEL AUTOR: Como en ocasiones anteriores hemos intercambiado los parámetros de las funciones insertar y añadir, de tal manera que lo que se inserte sea el primero y donde se inserte el segundo.

Además, se ha respetado la solución aportada por ser correcta salvo una línea, que no tengo clara (y no se ha modificado), que es la correspondiente con el primer "si" de la función. Se observa en dicha línea que el valor que se compara es el de hijo.ultimo_coincidente, aunque lo que me extraña en este caso es que la estructura hijo (o eso creo) aun no esté creada ya que se hace más adelante, por lo que pienso que sería nodo.ultimo_coincidente.

La **función principal**, que no es más que el refinamiento de la función de ramificación y poda empleando las funciones anteriores (acotar, válido, ...), quedaría como sigue:

```
fun ajustar (consulta, long_consulta, long_texto) dev coste
    inicializarSolucion (long_consulta, texto, long_texto, Solución,
    valor_sol_actual)
    Montículo := monticuloVacio()
    nodo.acciones = listaVacia()
    nodo.ultimo_coincidente = 0
    nodo.coste = 0
    nodo.long = long_consulta
    cota := nodo.coste + 2 * (long_texto – nodo.long)
    poner ((cota, nodo_raiz), Montículo)
    mientras no vacio (Montículo) hacer
        (cota, nodo) := quitarPrimero (montículo)
        si cota < valor_sol_actual entonces
        si nodo.ultimo_coincidente == long_texto entonces
            /* cota es el valor real de la mejor solución hasta ahora */
                valor_sol_actual := cota
                Solución := nodo.acciones
            si no
                para cada hijo en compleciones (nodo, texto, long_texto, consulta)
                hacer
                    cota := nodo.coste + 2 * (long_texto – nodo.long)
                    poner ((cota, hijo), Montículo)
                fpara
            fsi
        si no
            /* Ya no puede haber una solución mejor */
            escribir (Solución)
            devolver valor_sol_actual
        fsi
    fmientras
    escribir (Solución)
    devolver valor_sol_actual
ffun
```

5. Estudio del coste

   En este caso, únicamente podemos hallar una **cota superior** del coste del algoritmo por descripción del espacio de búsqueda. En el caso peor, se generan 3 hijos por nodo (intercambio, sustitución e inserción) hasta llegar a una profundidad long_texto. Por tanto, el espacio a recorrer siempre será *menor* que $\mathbf{3}^{long\_texto}$.

Febrero 2006-1ª (problema) (igual a problema Febrero 2008-2ª semana, a problema Septiembre 2008 y parecido al ejercicio 4.6 del libro de problemas)

Enunciado: Una empresa de montajes tiene n montadores con distintos rendimientos según el tipo de trabajo. Se trata de asignar los próximos n encargos, uno a cada montador, minimizando el coste total de todos los montajes. Para ello, se conoce de antemano la tabla de costes $C[1..n, 1..n]$ en la que el valor C_{ij} corresponde al coste de que el montador i realice el montaje j. Se pide:

1. Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema.
2. Descripción de las estructuras de datos necesarias.
3. Algoritmo completo a partir del refinamiento del esquema general.
4. Estudio del coste el algoritmo desarrollado.

Respuesta:

Tal y como hemos comentado en los ejercicios anteriores este ejercicio lo veremos aparte, ya que últimamente están poniendo problemas de exámenes parecidos a éstos y merece la pena tratarlos de nuevo e **insistir** en ellos. La única diferencia con respecto a este enunciado es que la matriz de costes la denominan T, por lo demás es idéntico.

1. Determinar qué esquema algorítmico es el más apropiado para resolver el problema

   Se trata de un **problema de optimización con restricciones**. Por tanto, podría ser un esquema voraz o uno de ramificación y poda. Sin embargo, descartamos el esquema voraz, porque no es posible encontrar una función de selección y de factibilidad tales que una vez aceptado un candidato se garantice que se va a alcanzar la solución óptima.

2. Escribir el esquema general

Tendremos el siguiente esquema general:

```
fun ramificaciónPoda (nodo_raiz) dev nodo
    Montículo := monticuloVacio()
    cota := acotar (nodo_raiz)
    poner ((cota, nodo_raiz), Montículo)
    mientras no vacio (Montículo) hacer
        (cota, nodo) := quitarPrimero (montículo)
        si es_mejor (cota, valor_sol_actual) entonces
            si solución (nodo) entonces
                devolver nodo;
            si no
                para cada hijo en compleciones (nodo) hacer
                    cota := acotar (hijo)
                    poner ((cota, hijo), Montículo)
                fpara
            fsi
    fmientras
    devolver Ø              /* No encuentra solución */
ffun
```

De nuevo, se insiste en este esquema que es bastante importante, por lo que razonaremos en la medida de lo posible el esquema (apreciación del tutor), teniendo en cuenta la teoría del tema, donde se especifica las funciones a implementar (recordemos que hay que ver las cotas, costes, el uso del montículo, etc.).

3. Indicar qué estructura de datos son necesarias

```
nodo = tupla
    asignaciones: vector [1..N];
    ultimo_asignado: cardinal;
    filas_no_asignadas: lista de cardinal;
    coste: cardinal
ftupla
```

Se usará un **montículo de mínimos**, donde la cota mejor es la de menor coste.

4. Desarrollar el algoritmo completo

Las **funciones generales** del esquema general hay que **instanciar** son:

a) **solución (nodo)**: Si se han realizado N asignaciones (último_asignado == N).
b) **acotar (nodo, costes)**: nodo.coste + "mínimo coste de las columnas no asignadas".
c) **compleciones (nodo, costes)**: Posibillidades para la siguiente asignación (valores posibles para asignaciones[último_asignado + 1]).

Observamos que la primera función ya está hecha, por tanto, especificaremos el resto, empezando por la función principal (como la lanzadera), que la denominaremos *asignación* (apreciación del autor):

```
fun asignación (costes[1..N, 1..N]) dev solucion[1..N]
    Montículo := monticuloVacio()
    nodo.ultimo_asignado = 0
    nodo.coste = 0
    cota := acotar (nodo,costes)
    poner ((cota, nodo), Montículo)
    mientras ¬vacio (Montículo) hacer
        (cota, nodo) := quitarPrimero (montículo)
        si nodo.ultimo_asignado == N entonces        /* Es solución */
            si solución (nodo) entonces
                devolver nodo.asignaciones;
            si no
                para cada hijo en compleciones (nodo) hacer
                    cota := acotar (hijo)
                    poner ((cota, hijo), Montículo)
                fpara
            fsi
    fmientras
    devolver ∅                                  /* No encuentra solución */
ffun
```

NOTA DEL AUTOR: Como hemos visto en todos los ejercicios anteriores recordamos de nuevo que al final de las sentencias hay que poner un punto y coma (;), dicho esto es una modificación con respecto al esquema general puesto anteriormente. De nuevo, es importante comprender bien de que constan estos esquemas, así como tener claras las funciones que posteriormente veremos.

La siguiente función es la de acotar el nodo, que corresponde con el punto 2 de los dados anteriormente:

```
fun acotar (nodo, costes[1..N, 1..N]) dev cota
    cota := nodo.coste
    para columna desde nodo.ultimo_asignado + 1 hasta N hacer
        mínimo = ∞
        para cada fila en nodo.filas_no_asignadas hacer
            si costes[columna, fila] < mínimo entonces
                mínimo := costes[columna, fila]
            fsi
        fpara
        cota := cota + mínimo
    fpara
    devolver cota
ffun
```

Por último y no menos importante, es la de compleciones, que sería:

```
fun compleciones (nodo, costes[1..N, 1..N]) dev lista_nodos
    lista := crearLista ()
    para cada fila en nodo.filas_no_asignadas hacer
        hijo := crearNodo ()
        hijo.ultimo_asignado := nodo.ultimo_asignado + 1
        hijo.asignaciones := nodo.asignaciones
        hijo.asignaciones[hijo.ultimo_asignado] := fila
        hijo.coste := nodo.coste + costes[hijo.ultimo_asignado, fila]
        hijo.filas_no_asignadas := nodo.filas_no_asignadas
        eliminar (fila, hijo.filas_no_asignadas)
        añadir (hijo, lista)
    fpara
    devolver lista
ffun
```

5. Estudio del coste

   En este caso, únicamente podemos hallar una **cota superior** del coste del algoritmo por descripción del espacio de búsqueda. En el caso peor, se generan (k – 1) hijos por cada nodo del nivel k, habiendo n niveles. Por tanto, el espacio a recorrer siempre será menor que $\boldsymbol{n}!$

Septiembre 2007 (problema)

Enunciado: Tenemos n objetos de volúmenes $v_1 \dots v_n$ y un número ilimitado de recipientes iguales con capacidad R (con $v_i \leq R, \forall i$). Los objetos se deben meter en los recipientes sin partirlos y sin recuperar su capacidad máxima. Se busca el mínimo número de recipientes necesarios para colocar todos los objetos.

La resolución de este problema debe incluir, por este orden:

1. Elección razonada del esquema algorítmico más apropiado para resolver el problema. Escriba dicho esquema general.
2. Descripción de las estructuras de datos necesarias.
3. Algoritmo completo a partir del refinamiento del esquema general.
4. Estudio del coste del algoritmo desarrollado.

Respuesta:

Se nos da una respuesta dada por el equipo docente al problema, aunque el algoritmo básico lo escribiremos, basándonos en ejercicio 15.9 del libro *Estructuras de datos y métodos algorítmicos. Ejercicios resueltos* de Narciso Martí, Y. ortega, J.A. Verdejo. Por ello, se da el algoritmo completo empleando la nomenclatura del dicho libro, dejándose como ejercicio aparte adaptarlo a la nomenclatura de nuestros exámenes. Queda por decir que ambas soluciones realmente es la misma, pero merece la pena verlas, ya que son puntos de vista algo distintos. Pasamos a resolver el problema.

1. Elección razonada del esquema algorítmico más apropiado para resolver el problema. Escriba dicho esquema general.

   La solución de este apartado dada por el equipo docente es: Se trata de un problema de optimización, para el que no existe una función de selección que permita ir seleccionando a cada paso el objeto que dé lugar a la construcción parcial de la solución óptima. Por tanto, **no** es posible aplicar un esquema voraz. Tampoco existe una forma de dividir el problema en subproblemas que se puedan resolver independientemente, por lo que **tampoco** es posible un esquema de divide y vencerás.

   La solución de este apartado por el libro de Martí: Recordemos que en la solución de vuelta atrás se supone que todo objeto cabe en un envase vacio y, por tanto, se necesitan un máximo de n envases. Las soluciones se representan en tuplas de la forma $(x_1, .., x_n)$, donde x_i es el envase donde hemos colocado el objeto i. Como los envases son indistinguibles, el primer objeto siempre se coloca en el primer envase (x_1 = 1) y para cada objeto de los restantes se puede usar uno de los envases ya ocupados si cabe en alguno, o coger uno vacio. La búsqueda podrá acabarse si encontramos una solución con el valor:

   óptimo = $\left\lceil \frac{\sum_{i=1}^{n} v_i}{E} \right\rceil$

   siendo E la capacidad de los envases.

   Tras ver lo anterior, el problema que tendremos será, por tanto, de **ramificación y poda**, donde nuestro esquema será de <u>minimización</u>, siendo éste el siguiente:

```
fun ramificación-y-poda (ensayo)
    m ← montículo-vacío
    cota-superior ← inicializar-cota-superior
    solución ← solución-vacía
    añadir-nodo (ensayo, m)
    mientras no vacío (m) hacer
        nodo ← extraer-raíz (m)
        si valido (nodo) entonces
            si coste (nodo) < cota-superior entonces
                solución ← nodo
                cota-superior ← coste (nodo)
            fsi
        si no                   { Nodo no es válido (solución) }
            si cota-inferior (nodo) ≥ cota-superior entonces
                devolver solución
            si no               { cota-inferior (nodo) < cota-superior }
                para cada hijo en compleciones (nodo) hacer
                    si condiciones-de-poda (hijo) y
                    cota-inferior (hijo) < cota-superior entonces
                        añadir-nodo (hijo, m)
                    fsi
                fsi
            fpara
        fsi
    fmientras
ffun
```

Posteriormente veremos las funciones que hay que especificar, así como el significado en el esquema, siendo esto básico para comprenderlo.

2. Descripción de las estructuras de datos necesarias

La solución dada por el equipo docente es:

Vector de objetos: Podemos representar el reparto de objetos entre recipiente mediante un vector en el que cada posición indique a qué recipiente se ha asignado al objeto correspondiente.

objetos = vector[1..n] de enteros

La **solución** es la cantidad entera S de recipientes empleados.

Montículo de mínimos, en el que cada componente almacena una solución parcial (nodo) con su cota correspondiente.

Por lo dicho anteriormente, la estructura de datos será:

```
nodo = tupla
    asignaciones: vector[1..n] de enteros
    etapa: cardinal                 // Objeto por el que se va procesando
    num_recip: cardinal             // Números de recipientes (solución)
    capacidad[1..n] de real         // Capacidad disponible de cada envase utilizado
```

En este apartado, la solución dada por el libro de Martí: En cada nodo, además de la información usual (solución parcial, etapa y prioridad) guardamos el número de envases utilizados y las capacidades disponibles de cada envase utilizado.

```
tipo nodo = reg
    sol[1..n] de 1..n
    k: 1..n
    envases: 1..n           { Prioridad }
    capacidad[1..n] de real
ftipo
```

3. Algoritmo completo a partir del refinamiento del esquema general

En cuanto a las estimaciones que definirán nuestro problema tendremos lo siguiente, siendo este apartado más o menos coincidente en ambas soluciones (del problema y del libro de Martí):

Cota inferior: Es el número de envases ya utilizados en la solución parcial.
Cota superior: Una cota superior seria considerar un envase extra por cada objeto que nos queda por empaquetar, pero resulta demasiado grosera. Podemos, en cambio, ir considerando cada objeto restante, en el orden que se haya dado e intentar meterlo en el primer envase utilizado y, en el caso en que no quepa, intentarlo con el segundo envase, y así hasta agotar todas las posibilidades, en cuyo caso, se añadirá un nuevo envase a la solución parcial.

Ahora veremos la solución que nos suministra el libro de Martí, en la cual se cambia algo la nomenclatura. Lo escribo de modo personal, ya que estimo interesante el poder compararlo con el resto de ejercicios, y como vimos previamente se deja como ejercicio aparte el hacerlo con nuestra nomenclatura (la de los exámenes de la UNED).

```
fun empaquetar-rp (E: real+, V[1..n] de real+) dev (sol-mejor[1..n] de 1..n,
                                                    envases-mejor: 1..n)
    var X, Y: nodo, C: colapr[nodo]
    total := 0
    para i = 1 hasta n hacer total := total + V[i] fpara
    óptimo = ⌈total/E⌉; encontrada := falso
    { Generamos la raíz: el primer objeto en el primer envase }
    Y.k := 1;  Y.sol[1] := 1; Y.envases := 1;
    Y.capacidad[1] = E – V[1]; Y.capacidad[2..n] := [E]
    C := cp_vacia (); añadir (C, Y)
    envases-mejor := calculo-pesimista (E, V, Y)
    mientras ¬encontrada ∧ ¬es-cp-vacía?(C) ∧
                mínimo(C).envases ≤ envases-mejor hacer
        Y := mínimo(C); eliminar-min(C);
        { generamos los hijos de Y }
        X.k := Y.k + 1; X.sol := Y.sol;
        X.envases := Y.envases; X.capacidad := Y.capacidad;
        { probamos con cada envase ya utilizado }
        i := 1;
        mientras i ≤ Y.envases ∧ ¬encontrada hacer
            si X.capacidad[i] ≥ V[X.k] entonces
                X.sol[X.k] := i; X.capacidad[i] := X.capacidad[i] – V[X.k]
                si X.k = n entonces
                    sol_mejor := X.sol; envases-mejor := X.envases;
                    encontrada := (envases-mejor = óptimo)       { terminar }
                si no
                    añadir (C, X); pes := calculo-pesimista (E, V, X)
                    envases-mejor := min (envases-mejor, pes)
                fsi
                X.capacidad := Y.capacidad
            fsi
        fmientras
        si ¬encontrada entonces
            { probamos con un nuevo envase }
            nuevo := Y.envases + 1; X.sol[X.k] := nuevo; X.envases = nuevo;
            X.capacidad[nuevo] := E – V[k]
            si X.envases ≤ envases-mejor entonces
                si X.k = n entonces
                    sol-mejor := X.sol; envases-mejor := nuevo
                    encontrada := (envases-mejor = optimo)       { terminar }
                si no
                    añadir (C, X); pes := calculo-pesimista (E, V, X)
                    envases-mejor := min (envases-mejor, pes)
                fsi
            fsi
        fsi
    fmientras
ffun
```

Como curiosidades a esta función, tendremos que la función añadir tiene los parámetros invertidos (he intentado rectificarlo en los problemas anteriores), siendo el primero la estructura (cola, en este caso) y en el segundo el dato que se añade (X, Y).
Otra curiosidad que encontramos es que igual que antes no finalizan todas las sentencias con punto y coma, estimando de nuevo que deberían hacerlo.

Para calcular las **estimaciones** (de cota superior) usamos la siguiente función:

```
fun calculo-pesimista (E: real+, V[1..n] de real+, X: nodo) dev (pes: 1..n)
    var capacidad-aux[1..n] de real
    pes := X.envases; capacidad-aux := X.capacidad
    para i = X.k + 1 hasta n hacer
        j := 1
        mientras V[i] > capacidad-aux[j] hacer j := j + 1 fmientras
        capacidad-aux[j] := capacidad-aux[j] – V[i]
        pes := max (pes, j)
    fpara
ffun
```

4. Estudio del coste del algoritmo desarrollado

   El número de recipientes está limitado a n, es decir, al número de objetos. Una estimación del coste es el tamaño del árbol, que en el peor caso crece como $\boldsymbol{O(n!)}$, ya que cada nodo del nivel k puede expandirse con los n – k objetos que quedan por asignar a recipientes.

Febrero 2008-1ª (problema)

Enunciado: El tío Facundo posee n huertas, cada una con un tipo diferente de árboles frutales. Las frutas ya han madurado y es hora de recolectarlas. El tío Facundo conoce, para cada una de las huertas, el beneficio b_i que obtendría por la venta de lo recolectado. El tiempo que se tarda en recolectar los frutos de cada finca es así mismo variable (no unitario) y viene dado por t_i. También sabe los días d_i que tardan en pudrirse los frutos de cada huerta. Se pide ayudar a decidir al tío Facundo qué debe recolectar para maximizar el beneficio total obtenido.

La resolución de este problema debe incluir, por este orden:

1. Elección justificada del esquema más apropiado, el esquema general y explicación de su aplicación al problema
2. Descripción de las estructuras de datos necesarias
3. Algoritmo completo a partir del refinamiento del esquema general
4. Coste del algoritmo

Respuesta:

Este ejercicio está basado en el del libro de Martí ejercicio 15.4, combinándola con otra solución distinta, la cual son exactamente iguales. La única diferencia estriba en que es una nomenclatura mucho más acorde a los ejercicios de exámenes, por lo demás, es casi similar.

1. Elección justificada del esquema más apropiado, el esquema general y explicación de su aplicación al problema

   El problema es de **maximización** (**optimización**) del beneficio total del tío Facundo, por tanto, podremos escoger entre algoritmos voraces o bien ramificación y poda, descartando el divide y vencerás, ya que no se puede dividir el problema en subproblemas del mismo tamaño y luego combinarlos. Igualmente, descartaríamos el esquema de vuelta atrás, ya

que se nos pide que sea el más eficiente y en este caso al ser de optimización seria ineficiente, seleccionando todas las posibles soluciones y quedándonos con la mejor.

Como hemos visto antes, dudamos entre el esquema voraz y el de ramificación y poda. Para ello podemos un ejemplo, como sigue:

Contraejemplo (que no es esquema voraz): Si cogemos la función de selección como la de mayor beneficio tendremos la siguiente tabla con dos posibles huertas:

	1	2
b_i	100	50
t_i	4	1
d_i	100	90
b_i/t_i	25	50

siendo:

b_i: Beneficio.
t_i: Tiempo en recolectar
d_i: Plazo de la recolección.

Siguiendo nuestro planteamiento voraz cogeremos la de mayor relación b_i/t_i. Haremos esta huerta aunque sea la de menor beneficio. Por ello, descartamos el algoritmo voraz, por no llegar a una solución óptima.

Además de rechazar mediante el contraejemplo antes escrito tendremos que comprobar que un subconjunto de huertas es **factible**, es decir, que todas pueden recolectarse sin superar su plazo, manteniendo las huertas ordenadas por tiempo de caducidad creciente. Para ello, Representamos las **soluciones** mediante tuplas $(x_1, x_2, .., x_n)$ donde $x_i = 1$ indica que los frutos de la huerta i se recolectan mientras que $x_i = 0$ indica que los frutos de la huerta i no se recolectan.

Recordaremos del tema de los algoritmos voraces de planificación con plazo fijo el siguiente lema:

Lema 6.6.4 Un conjunto J de n tareas es factible si y sólo si se puede construir una secuencia factible que incluya a todas las tareas de J en la forma siguiente. Se empieza por una planificación vacía, de longitud n. entonces para cada tarea $i \in J$ sucesivamente, se planifica i en el instante t_i en donde t es el mayor entero tal que $1 \leq t \leq min(n, d_i)$ y la tarea que se ejecuta en el instante t no está decidida todavía.

Nos va a servir para probar que las huertas puestas por orden creciente de caducidad es una secuencia factible.

$d_1 \leq d_2 \leq \cdots \leq d_n$

En cada paso se comprueba si el plazo de la última tarea puede entrar dentro de la planificación general de las huertas, pudiéndose hacerse, es decir, si la huerta es posible recolectarse dentro del plazo establecido.

Un ejemplo de ello será el siguiente, suponiendo que se nos dan ya k huertas recolectadas y queremos ver si la k + 1 es posible recolectarla.

$[h_1, h_2, \dots, h_k, |\{h_{k+1}\}, \dots, h_n]$

T = 50 días T = 10 días

Nuestra huerta h_{k+1} tiene caducidad de 70 días, por lo que el tiempo total de recolección será: 50 + 10 = 60 días, que es menor que 70. Viendo así con todas las demás huertas que se pueden recolectar.

Conviene hacer esta misma demostración en el examen, ya que es un problema parecido al de los algoritmos voraces, sólo que al tener un plazo de recolección variable (como pone en el enunciado) no cumple la optimalidad el algoritmo voraz y se descarta. Además, al ser un problema de maximización tendremos el siguiente esquema, que recordemos que era el segundo de los que dimos en la teoría:

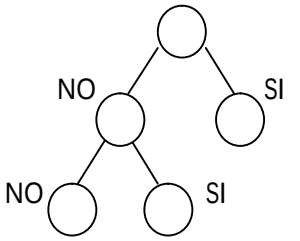
```
fun ramificación-y-poda (ensayo)
    m ← montículo-vacío
    cota-superior ← inicializar-cota-superior
    solución ← solución-vacía
    añadir-nodo (ensayo, m)
    mientras no vacío (m) hacer
        nodo ← extraer-raíz (m)
        si valido (nodo) entonces
            si coste (nodo) > cota-inferior entonces
                solución ← nodo
                cota-inferior ← coste (nodo)
            fsi
        si no                   { Nodo no es válido (solución) }
            si cota-superior (nodo) ≤ cota-superior entonces
                devolver solución
            si no               { cota-superior (nodo) > cota-superior }
                para cada hijo en compleciones (nodo) hacer
                    si condiciones-de-poda (hijo) y
                    cota-superior (hijo) > cota-superior entonces
                        añadir-nodo (hijo, m)
                    fsi
                fpara
            fsi
        fsi
    fmientras
ffun
```

En nuestro caso:

Válido: Tendremos un vector de false y si no se sigue podríamos darlo por bueno, es decir, es solución cuando se llega al final y la última huerta es recolectable.
Condiciones de poda: Será el que realmente nos seleccionará el nodo que queremos añadir. Se nos da la solución siguiente, en la que añadiremos la huerta h_{k+1} (es parecido al ejemplo anterior):

$[h_1, h_2, \dots, h_k, \{h_{k+1}\}, \dots, h_n]$

Al añadirlo nos quedaría el siguiente árbol:



Nivel k

Nivel k+1

Cota: Tiene que ser lo más cercano posible a la solución óptima. Debería requerir un esfuerzo razonable para hallar la cota, aunque no siempre es así. La cota nos servirá para dirigir la búsqueda y podar más ramas, para así irnos por la rama más prometedora.
Cota inferior: Podemos aproximar superiormente el beneficio obtenido sumando el beneficio de todas las huertas que pueden recolectarse sin llegar a su fecha de caducidad después de las ya elegidas (es decir, empezando a recolectar cada uno justo después de terminar con la última ya elegida).
La **cota superior** será aquella en la que calculamos una posible solución extensión de la que tenemos: las huertas no consideradas se van recolectando en orden (acumulando el tiempo), siempre que no se supere su fecha de caducidad.

NOTA DEL AUTOR: Este ejercicio, como comentábamos antes, está quedando de un modo "extraño" y poco claro, por ello decir que según entiendo en el libro de Martí la cota inferior la denominan estimación optimista y la superior estimación pesimista, por lo que las dos cotas anteriores (inferior y superior) son copia textual del libro. Dicho esto, no sé si estará bien hecho el ejercicio en sí, sólo que esa es mi interpretación personal.

En este ejercicio nos iremos por la rama más prometedora siguiendo el árbol anterior, hasta llegar a recolectar todas las huertas, por lo que la solución será $[h_1, h_2, \dots, h_n]$.

2. Descripción de las estructuras de datos necesarias

Tendremos estas estructuras de datos, siendo la información de partida la siguiente:

```
H[1..n] de huerta          // Array de huertas dadas

tipo huerta = registro
    id                     // Indica el nº de huerta del vector ordenado de caducidad
    beneficio
    tiempo
    caducidad
```

Podremos quitar el campo *id* si estuvieran ordenados las huertas por orden creciente de tiempos, con lo que podríamos ponerlo como otra estructura distinta. Lo dejaremos así. Nuestro ensayo será el siguiente:

```
tipo ensayo = registro
        V[1..n] de boolean    // Valor de la solución
        K                     // Nivel del árbol, nivel de decisión de las huertas
        valor                 // Beneficio acumulado
        tiempo                // Tiempo para recolectarlo
```

3. Algoritmo completo a partir del refinamiento del esquema general

Tendremos que ordenar por orden creciente las huertas, con lo que el esquema refinado nos quedará empleando la segunda estructura de datos como sigue:

```
fun ramificación-y-poda (H[1..n] de huerta)
    m ← montículo-vacío
    cota-inferior ← 0              // No se puede recolectar ninguna huerta
    solución ← nuevo-ensayo
    solución.v ← [0,0,0,...,0]
    solución.k ← 0
    solución.tiempo ← 0
    añadir-nodo (solucion, m)
    mientras no vacío (m) hacer
        nodo ← extraer-raíz (m)
        si (nodo.k == n) entonces                    // Ya es solución
            si nodo.valor > cota-inferior entonces
                solución ← nodo
                cota-inferior ← nodo.valor
            fsi
        si no                    // No necesitamos seguir explorando más nodos
            si cota-superior (nodo) ≤ cota-superior entonces
                devolver solución
            fsi
            // Realiza compleciones del nodo, primero el hijo del "SI" y luego el
            del "NO"
            k ← nodo.k
            hijo1 ← nodo
            hijo1.V[k + 1] ← cierto
            hijo1.valor ← hijo1.valor + H[k + 1].beneficio
            hijo1.tiempo ← hijo1.tiempo + H[k + 1].tiempo
            hijo1.k ← k + 1
            // Creamos hijo del "NO"
            hijo2 ← nodo
            hijo2.V[k + 1] ← falso
            hijo2.k ← k + 1

            si cota-superior (hijo1) > cota-inferior entonces
                si hijo1.tiempo ≤ H[k + 1].caducidad entonces
                    añadir-nodo (hijo1, m)
                fsi
            fsi
            si cota-superior (hijo2) > cota-inferior entonces
                añadir-nodo (hijo2, m)
            fsi
        fsi
    fmientras
ffun
```

Vemos que para el hijo1 hay una condición más que cumplir y es que tendrá que tener un tiempo menor o igual que la suma de su caducidad y del tiempo de recolección. Recordemos el ejemplo anterior, donde se daba está situación al añadir el nodo del "SI".

Para el hijo2 (el del "NO") vemos que dicha condición no se cumple, debido a que tiene el mismo tiempo que su padre, al no recolectarse esta huerta.

Por último, la función que halla la **cota superior** será la siguiente:

```
fun cota-superior (ensayo, H) dev entero
    aux ← ensayo.valor
    para i desde k + 1 hasta n hacer
        si H[i].caducidad ≥ ensayo.tiempo + H[i].tiempo entonces
            aux ← aux + H[i].valor
        fsi
    fpara
    dev aux
ffun
```

siendo H, una *variable global* o bien un *parámetro de la función*, en nuestro caso es un parámetro, aunque el resultado es igual. Si fuera variable global se tendría que quitar de la función simplemente.

Podremos hacer una **función principal** que ordene las huertas y lance el procedimiento de ramificación y poda.

4. Coste del algoritmo

Es un árbol binario, aunque implica explorar todo el árbol. Podremos dar una estimación del coste total, posiblemente (no estoy segura), podrá ser coste $\boldsymbol{O(n!)}$. Además de todo esto, pese a ser un cota superior muy mala hay costes ocultos del montículo que no se cuentan.

3ª parte. Problemas de exámenes sin solución o planteados:

Febrero 1997-2ª (problema 2)

Enunciado: Queremos grabar *n* canciones de duraciones $t_1, t_2, \dots, t_n$ en una cinta de audio de duración $T < \sum_{i=1}^{n} t_i$.
Diseñar un algoritmo que seleccione las canciones de forma que se minimice el espacio vacío que queda en la cinta.

Respuesta: este problema no lo sabría resolver muy bien, pero al ser un problema de minimización debería ser o bien un algoritmo voraz o bien uno de **ramificación y poda**. Como hemos visto en numerosos ejercicios anteriores no hay función de selección adecuada para dicho problema, por lo que descartamos este primero. Nos quedamos, por tanto, con el de ramificación y poda.

Febrero 1998-2ª (problema 2)

Enunciado: Se tiene un mecano de 8 piezas. Las piezas se acoplan entre sí mediante tornillos, formando distintos juguetes dependiendo de cómo se combinen. Un juguete completo es aquel formado por las 8 piezas. El gasto de tornillos de cada acoplamiento es el indicado en la siguiente tabla (un – indica que las piezas no encajan):

	P_1	P_2	P_3	P_4	P_5	P_6	P_7	P_8
P_1	-	7	12	-	4	-	-	-
P_2	7	-	2	6	1	-	1	-
P_3	12	2	-	4	-	10	-	3
P_4	-	6	4	-	-	3	2	-
P_5	4	1	-	-	-	20	10	-
P_6	-	-	10	3	20	-	3	-
P_7	-	1	-	2	10	5	-	-
P_8	-	-	3	-	-	-	-	-

Se pide construir un algoritmo que calcule la combinación de piezas que forman un juguete completo minimizando el gasto de tornillos.

Respuesta: Este problema es parecido (similar) al de los mensajeros y tareas, salvo que en este caso hay que minimizar el gasto de tornillos que montan juguetes y hay que minimizar el gasto de dichos tornillos. Se nos da una tabla equivalente a la de costes de ese ejercicio. Al estar resuelto el problema de los mensajeros y tareas, dejaremos como ejercicio aparte el actual problema.

Febrero 1999-1ª (problema 1)

Enunciado: La nave Mir tiene que reensamblar sus paneles modulares debido a una sobrecarga. Hay 6 paneles que se ensamblan unos con otros formando una estructura única y de la combinación del ensamblaje depende el gasto de energía. La tabla adjunta muestra el coste de amperios de cada unión:

	P_1	P_2	P_3	P_4	P_5	P_6
P_1	5	9	4	2	1	1
P_2	6	0	1	1	3	5
P_3	1	9	5	5	2	5
P_4	1	4	2	3	5	6
P_5	3	6	7	7	1	3
P_6	1	3	5	6	2	8

Se pide diseñar un algoritmo que forme una estructura con todos los módulos minimizando el coste total tanto de uniones como del coste de las mismas.

Respuesta: Este ejercicio es similar en su resolución al de los **mensajeros y tareas**, incluido el anterior problema. Por tanto, igualmente lo dejaremos sin resolver por haberse hecho ya numerosas veces.

Febrero 1999-2ª (problema 1)

Enunciado: Diseñar un algoritmo que tome un mapa político y coloree el mapa con el mínimo número de colores posible de manera que dos países fronterizos no tengan el mismo color.

Respuesta: Este ejercicio es similar al de los colores de los mapas políticos (problema 2 de Febrero 99-1ª), sólo que nos piden que minimicemos el número de colores posible. Por tanto, se correspondería con un problema de **ramificación y poda** o eso creo yo. Se deja la resolución para mis ávidos lectores ;)

Septiembre 2002-reserva (problema)

Enunciado: El juego del buscaminas consiste en un tablero T de tamaño *n x n* cuyas casillas $T(i,j)$ están ocupados por una mina, o por una cifra que indica el número de minas adyacentes en horizontal, diagonal o vertical (por lo tanto, el número estará entre 0 y 8). Al comienzo del juego, el contenido de todas las casillas está oculto. En cada jugada, se debe escoger una casilla para descubrir su contenido. Si se trata de una mina, el juego termina en fracaso. Si se trata de una cifra, ésta puede usarse para razonar sobre la posición de las minas; por ejemplo, si se trata de un cero, puede descubrirse todas las casillas contiguas sin peligro. El juego termina con éxito si se consiguen descubrir todas las casillas sin minas, y sólo esas. El número total de minas, *m*, es conocido de antemano por el jugador.

Supóngase programada de antemano una función $P(i,j)$ que devuelve la probabilidad de que una casilla $T(i,j)$ esté ocupada por una mina, con la información de la que se dispone en cada momento (para un tablero dado T y en el instante de juego *t*). Por ejemplo, antes de comenzar la partida (tablero T, e instante $t = 0$) $P(i,j) = \frac{m}{n^2}$. Para todas las casillas $T(i,j)$.

Escribir un algoritmo que, dado un tablero *n x n* con *m* minas, juegue al buscaminas maximizando la probabilidad de finalizar la partida con éxito. Una vez realizado el algoritmo, se valorarán también los razonamientos sobre cómo puede calcularse y actualizarse el valor $P(i,j)$.

Respuesta: En este caso se ve que es un problema de optimización donde por cada casilla descubierta y sin encontrar ninguna mina hay que minimizar el valor de $P(i,j) = \frac{m}{n^2}$. Por tanto, lo único que nos están contando en el enunciado es que habrá más posibilidades (que por eso tiene el nombre P o eso creo) de encontrar la mina en una casilla en la que tenga un número más alto y menos posibilidad en caso contrario. Por eso, trataremos de minimizar la posibilidad de encontrar las minas. Se deja también como ejercicio al lector la resolución del mismo.

Septiembre 2003-reserva (problema)

Enunciado: Daniel se va de veraneo y tiene que decidir qué tesoros se lleva (el coche de hojalata, el madelman, el pañuelo de María, etc.). Su madre le ha dado una bolsa de V_b centímetros cúbicos con la orden expresa de que no se puede llevar nada que no le quepa en la bolsa. Daniel tiene N objetos candidatos. Cada objeto i ocupa un volumen V_i y tiene un valor sentimental S_i para él. Se trata de llenar la bolsa, maximizando el valor sentimental de los objetos que contiene. Evidentemente, Daniel no está dispuesto a romper en pedazos sus tesoros.

Respuesta: Al igual que hemos visto en la teoría este problema corresponde con el de **ramificación y poda**, por ser un problema de optimización, en el que hay que maximizar el valor sentimental del mismo. Solamente planteamos el problema sin resolverlo.

Diciembre 2006 (problema) (igual al problema 2 de Febrero 1997-2ª)

Enunciado: Se desea grabar un CD de audio, de capacidad T, con canciones de una colección de *n* elementos, cuyas duraciones $t_1, \ldots, t_n$ cumplen: $\sum_{i=1}^{n} t_i > T$. Diseña un algoritmo que permita almacenar el máximo número de canciones en el espacio disponible. Puede suponerse que la colección de canciones está ordenada por longitud de menor a mayor.

La resolución de este problema debe incluir, por este orden:

1. Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema (1 punto).
2. Descripción de las estructuras de datos necesarias (0.5 puntos).
3. Algoritmo completo a partir del refinamiento del esquema general (2 puntos).
4. Estudio del coste del algoritmo desarrollado (0.5 puntos).

Respuesta: Este problema puede ser o bien un voraz o bien uno de ramificación y poda. No podremos resolverlo por el primer tipo de esquema, ya que se encuentra un **contraejemplo** por la que no se puede llegar a solución óptima (es parecido en este caso al problema de las mondas, solo que se maximiza el número de canciones y en el de las monedas se devuelve el mínimo de monedas). No resolveremos el problema por tenerlo en la sección de la teoría de este tema.

Febrero 2008-2ª (problema) (parecido al problema 4.6 del libro de problemas)

Enunciado: Una flota de 4 camiones ($T1..T4$) debe transportar un cargamento variado a otras tantas ciudades ($C1..C4$). el coste de adjudicar el transporte varía en función de la distancia y de la peligrosidad del trayecto y se resume en la siguiente tabla. Exponer un algoritmo que calcule de manera óptima a quién encargarle qué destino de manera que en total el coste sea mínimo.

	T1	T2	T3	T4
C1	24	45	12	34
C2	56	56	12	76
C3	90	67	32	54
C4	32	23	12	23

La resolución de este problema debe incluir, por este orden:

1. Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema.
2. Descripción de las estructuras de datos necesarias.
3. Algoritmo completo a partir del refinamiento del esquema general.
4. Estudio del coste del algoritmo desarrollado.

Respuesta: De nuevo tenemos un ejercicio similar al de los mensajeros y tareas (recordemos **problema 4.6** del libro de problemas). No nos pararemos en más detalles.

Septiembre 2008-reserva (problema)

Enunciado: Un cajero automático dispone de n tipos distintos teniendo cada uno de los n tipos un valor distinto. Se trata de calcular si es posible suministrar al cliente el valor exacto solicitado, y si éste fuera el caso el sistema deberá suministrar el conjunto de billetes que forman una solución, además se desea que el sistema utilice el menor número de billetes posible. Se puede suponer que el número de billetes de cada tipo disponibles es infinito.

1. Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema.
2. Descripción de las estructuras de datos necesarias.
3. Algoritmo completo a partir del refinamiento del esquema general.
4. Estudio del coste del algoritmo desarrollado.

Respuesta: Este problema es muy parecido al del **cambio de monedas** realizado usando el algoritmo de ramificación y poda (dado en la teoría del tema), por lo que lo dejaremos como ejercicio aparte para hacerlo. De igual manera que antes, no nos pararemos en detalles.

Ejercicios resueltos de programación 3

Prácticas de la asignatura.

Alumna: Alicia Sánchez
Centro: UNED-Las Rozas (Madrid)

Hemos creado un documento aparte donde iremos guardando los ejercicios de las distintas prácticas hasta el momento propuestas por el equipo docente. Al ser un documento aparentemente sencillo, ya que en ocasiones dependerá de la solución lo que el alumno haya implementado en su práctica (es bastante subjetivo), no en todas las preguntas se pondrá la solución. La idea de hacer este documento es ver la tónica general a la hora de preguntar y además completar los ejercicios que faltan.

No tendremos un índice como tal, ya que estas cuestiones forman un conjunto, por tanto, empezaremos a ver estas preguntas.

Febrero 2000-1ª (ejercicio 2)

Enunciado: Comenta las condiciones de poda que has utilizado en la realización de la práctica del Nim

Respuesta: Este ejercicio es totalmente subjetivo como hemos comentado antes, pero podremos decir basándonos en el libro de Brassard, que las condiciones de poda podrán ser aquellas que no lleguen a ganar el juego o bien en tablas. Por ello, se tendría que hacer el árbol de expansión hasta llegar a solución. De todas maneras, para ello en el citado libro vendría más expandido, incluso más que en nuestro resumen de la asignatura.

Febrero 2006-1ª (ejercicio 2)

Enunciado: Declara en Java o en Modula-2 las clases y/o estructuras de datos que utilizarías en el problema del Su-doku (práctica de este año) para comprobar en un tiempo de ejecución constante respecto a la dimensión n del tablero (n x n) que una casilla contiene un valor factible.

Respuesta: Este tipo de problemas creo que es preferible implementarlos en Java, antes que en Modula-2, ya que será el lenguaje que posteriormente se use. Este año fue especial, ya que al ser de transición se dejaron ambos lenguajes.

Existen varias alternativas, pero de nada sirve que verificar una casilla se realice en tiempo constante si luego utilizar su valor se realiza en tiempo lineal. En la siguiente solución se declaran tres tablas de booleanos que indican si ya hay o no un determinado valor en una determinada fila, columna o región (cuadrante), respectivamente. Si no es así, entonces es factible poner el valor en la casilla. Tanto la función de verificación como las de actualización tienen un coste computacional constante.

```
public class Tablero {
    int N = 9;
    int subN = (int)Math.sqrt (N);
    boolean val_en_fil [][] = new boolean [N] [N];
    boolean val_en_col [][] = new boolean [N] [N];
    boolean val_en_reg [][] = new boolean [N] [N];

    boolean valorFactible (int fil, int col, int val) {
        int reg = region (fil, col);
        return (!val_en_fil [fil][val] &&
                !val_en_col [col][val] &&
                !val_en_reg [reg][val]);
    }

    void poner (int fil, int col, int val) {
        int reg = region (fil, col);
        val_en_fil [fil][val] = true;
        val_en_col [col][val] = true;
        val_en_reg [reg][val]= true;
    }
```

```
void quitar (int fil, int col, int val) {
    int reg = region (fil, col);
    val_en_fil [fil][val] = false;
    val_en_col [col][val] = false;
    val_en_reg [reg][val]= false;
}

int región (int fil, int col) {
    return (col/subN) * subN + fil/subN;
}
```

Febrero 2006-2ª (ejercicio 1)

Enunciado: En la práctica obligatoria del presente curso 2005/2006 se ha tenido que diseñar y desarrollar un algoritmo para resolver el juego del Su-doku. Codifique en Java o en Modula-2 un algoritmo iterativo que recorra el cuadrado de 3 x 3 casillas que corresponda a una casilla que ocupa las posiciones *i*, *j* del tablero.

Respuesta: Se trataba de encontrar la relación entre las posiciones *i*, *j* del tablero y las posiciones de comienzo del cuadrado de 3 x 3 que les corresponde en el tablero del Su-doku. Supongamos que el tablero tiene como índices 0..8, 0..8 y que *i* y *j* son de tipo entero, la relación se puede establecer de la siguiente manera:

```
int coordFilaInicioCuadrado = (i/3) * 3;              // División entera
int coordColumnaInicioCuadrado = (j/3) * 3;           // División entera
```

Ahora sólo queda hacer el recorrido:

```
for (int k = coordFilaInicioCuadrado; k < coordFilaInicioCuadrado + 3; k++) {
    for (int l = coordColumnaInicioCuadrado; l < coordColumnaInicioCuadrado + 3;
    l++) {
        procesar (tab [k][l]);
    }
}
```

De nuevo y como curiosidad vemos que las variables tienen un nombre enorme, yo personalmente lo recortaría y pondría algo más adecuado para el examen, ya que como se ha insistido casi no hay tiempo material y perder el tiempo supone casi suspender. Queda, por tanto, como ejercicio el cambiar el nombre.

Febrero 2007-1ª (ejercicio 1) (igual al ejercicio 3 de Septiembre 2007-reserva)

Enunciado: Implementa en Java o Modula-2 la función de estimación de la cota en el problema del n-puzzle (práctica de este curso).

Respuesta: En este caso el ejercicio no tiene solución oficial, por lo que dependerá en cada caso del alumno. En estos tipos de ejercicios aunque sea más complicado preferiría implementarlo en Java que en Modula-2, por la facilidad del lenguaje. En cuanto a la estimación de la cota podremos decir que es la suma de la cota realizada hasta el momento más la distancia de Manhattan, recordemos que sería esta la suma del número de posiciones que hay desde su disposición inicial al final. Para implementarlo en Java (preferiblemente) se debería realizar dos funciones, la de la distancia de Manhattan y la de la cota superior propiamente dicha.

Febrero 2007-2ª (ejercicio 2)

Enunciado: Escribe en Java el algoritmo principal del esquema de ramificación y poda utilizado en la práctica del presente curso (no es necesario implementar las funciones auxiliares).

Respuesta: La implementación más sencilla es dejar los nodos solución dentro de la cola de prioridad. Cuando el primero de la cola sea solución hemos encontrado la solución óptima, porque el resto de nodos proporcionan una estimación que aún en el mejor de los casos sería peor:

```
public class MejorPrimero implements SearchAlgorithm {
    public solution SearchSolution (Node nodoInicial) {
        Solution solución;
        Queue<Node> cola = new PriorityQueue<Node> ();
        cola.add (nodoInicial);
        while (!cola.isEmpty()) {
            Node nodo = cola.poll();
            If (nodo.isSolution () ) {
                return nodo.getSolution () ;
            } else {
                for (Node nodoHijo: nodo.generatePossibleChildren()) {
                    cola.offer (nodoHijo);
                }
            }
        }
        return null;
    }
}
```

Sin embargo, la solución anterior tiene el problema de que todos los nodos (incluidas algunas soluciones) se guardan en memoria (en la **cola de prioridad**) aún sabiendo que muchos no pueden proporcionar la solución óptima. Por tanto, es posible refinar el algoritmo anterior para ir eliminando de la colad de prioridad todos los nodos que podamos ir descartando. Para descartar nodos, primero tenemos que encontrar una solución que se convierta en la solución de referencia. Todos los nodos ya existentes con una estimación peor ya no se meten en la cola. Está versión necesita menos memoria, pero puede tener mayor coste computacional, puesto que hay que recorrer la cola cada vez que encontramos una nueva solución. Por otro lado, mantener la cola tiene menos coste porque en general contendrá menor número de nodos.

```
public class RamificacionYPoda implements SearchAlgorithm {
    public solution SearchSolution (Node nodoInicial) {
        Queue<Node> cola = new PriorityQueue<Node> ();
        cola.add (nodoInicial);
        int costeMejor = Integer.MAX_VALUE;       // Solución mejor
        while (!cola.isEmpty()) {
            Node nodo = cola.poll();
            if (nodo.isSolution () ) {
                int costeReal = nodo.getCost ();
                if (costeReal < costeMejor) {
                    costeMejor = costeReal;
                    solucion = nodo.getSolution ();
                    depurar (cola, costeMejor);
                }
            } else {
                for (Node nodoHijo: nodo.generatePossibleChildren()) {
                    int cota = nodoHijo.calculateEstimatedCost();
                    if (cota < costeMejor) {
                        cola.offer (nodoHijo);
                    }
                }
            }
        }
        return solucion;
    }

    private void depurar (Queue<Node> cola, int costeMejor) {
        Iterator it = cola.iterator ();
        List<Node> depurationList = new ArrayListe<Node> ();
        while (it.hasNext ()) {
            Node nodo = (Node) it.next ();
            if (node. calculateEstimatedCost() > costeMejor) {
                depurationList.add (node);
            }
        }
        cola.removeAll (depurationList);
    }
}
```

NOTA DEL AUTOR: De nuevo, vemos que los nombres de los métodos de java son grandísimos, por lo que los reduciría a 8 caracteres o 10 a lo sumo. Por otro lado, se observa que es un refinamiento. Como he comentado en ejercicios anterior, en todo caso el código lo haría de una manera más fácil y más general, aun así es interesante ver el uso de las clases, métodos e incluso interfases (SearchAlgorithm) o eso creo en Java.

Septiembre 2007 (ejercicio 1)

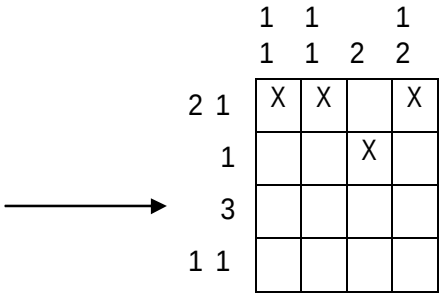
Enunciado: Describe el grafo asociado al espacio de solución del problema del *n-puzzle* (práctica de este curso).

Respuesta: En estos ejercicios al pedir el grafo asociado personalmente lo dibujaría y lo explicaría. Al ser personal este grafo no lo haríamos, en este caso, pero pondremos la solución aportada del ejercicio:

- Cada nodo se bifurca como máximo en los cuatro movimientos posibles del hueco (arriba, abajo, izquierda y derecha), siempre que sea posible.
- La profundidad del árbol está limitada por el número de distintos tableros posibles ($(n*n)!$, siendo n el tamaño del lado del tablero), ya que no se deben repetir.

Febrero 2008-1ª (ejercicio 1)

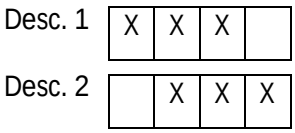
Enunciado: Escribe el grafo asociado al espacio de soluciones del problema del nonograma (práctica de este año), a partir del nodo que se presenta a continuación. Se supone que a partir de dicho nodo se van a comenzar a explorar la fila 3 (no puntúa la exploración por fuerza bruta de coste 2^{n^2}).



Respuesta: En TODOS los exámenes de este mismo curso va a ser la tónica general. Por ello, con hacer un solo ejercicio nos es suficiente para los demás.

Recordemos y así lo hice yo es que tenemos dos vuelta atrás, una que es la que vamos a hacer para hallar los descendientes de las filas según las restricciones dadas al nonograma y otra para verificar si en las columnas es correcto y cumple las restricciones de las columnas. Suponemos que inicialmente están vacías las casillas y que conformen se completen tienen valor 0 ó 1. Pondremos un pequeño ejemplo y se verá como se hace.

Para la fila 3, que es donde empezaremos tendremos estos descendientes:



Empezaremos colocando el primer descendiente al grafo anterior:

	1 1	1 1	2	1 2
2 1	X	X		X
1			X	
3	X	X	X	
1 1				

Verificamos que las columnas cumplan las condiciones, es decir, que al intentar probar con la restricción 1 1 (recordemos que tiene, al menos, un espacio en blanco) realmente pueda llegar a solución. En este caso, sería positiva la verificación por lo que probaríamos con la siguiente fila de igual manera:

Desc. 1	X		X	
Desc. 2	X			X
Desc. 3		X		X

Probaríamos ahora con el primer descendiente, aunque no acabaremos de resolver el nonograma lo dejaremos marcado para que se vea cuando hay que hacer vuelta atrás, ya que el procedimiento es similar.

	1 1	1 1	2	1 2
2 1	X	X		X
1			X	
3	X	X	X	
1 1	X			X

Al verificar las restricciones de la primera columna ya vemos que no las cumple, por lo que se tendría que probar con el siguiente descendiente, hasta verlos todos. Si no se subiría de nivel y se cambiaria de descendiente consecutivamente. Gráficamente, es más un árbol donde se podaría al llegar a este nivel con este descendiente. Se deja, por tanto, la finalización pendiente de hacer.