

Ejercicios resueltos de programación 3

Tema 9. Exploración de grafos.

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

| | |
|--|----|
| 1. Introducción teórica | 4 |
| 2. Cuestiones de exámenes | 8 |
| 3. Problemas de exámenes solucionados | 16 |
| 4. Problemas de exámenes sin solución o planteados | 63 |

Ramificación y poda

| | |
|--|----|
| 1. Introducción teórica | 71 |
| 2. Cuestiones de exámenes | 73 |
| 3. Problemas de exámenes solucionados | 74 |
| 4. Problemas de exámenes sin solución o planteados | 98 |

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  $v \in N$  hacer  $\text{marca}[v] \leftarrow$  no visitado
  para cada  $v \in N$  hacer
    si  $\text{marca}[v] \neq$  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 }
   $\text{marca}[v] \leftarrow$  visitado
  para cada nodo  $w$  adyacente a  $v$  hacer
    si  $\text{marca}[w] \neq$  visitado entonces rp ( $w$ )
```

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  $v \in N$  hacer marca [v] ← no visitado
  para cada  $v \in N$  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:

$$\left[\left[[-7,5][-3][-10,-20,0] \right] \left[[-5,-10][-15,20] \right] \left[[1][6,-8,14][-30,0][-8,-9] \right] \right]$$

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.

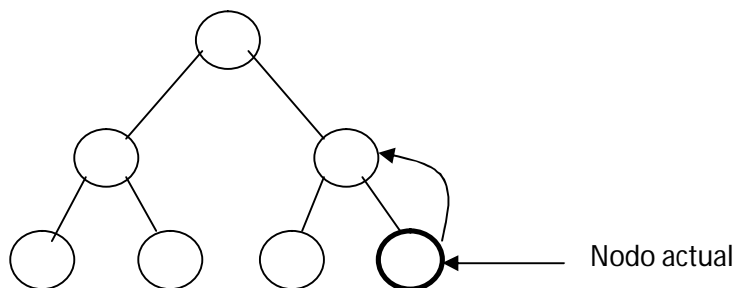
Sin embargo, no existe una función $eval(u)$ perfecta en el caso del ajedrez, deberemos entonces buscar una función $eval(u)$ aproximada con una relación coste/precisión lo mejor posible. En este caso, $eval(u)$ no nos confirma que el mejor movimiento considerado por ella sea en realidad el más adecuado. Y es aquí donde entra 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)

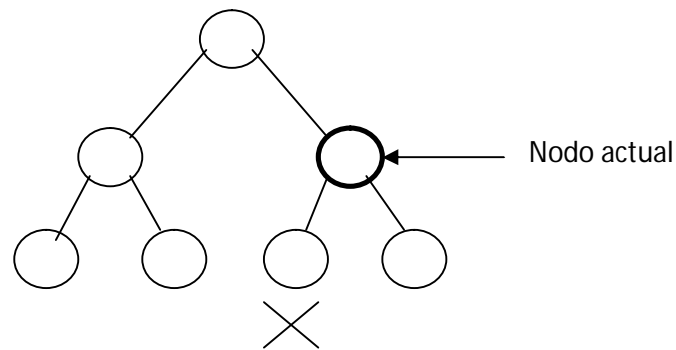
Enunciado: En los algoritmos de vuelta atrás explica la diferencia entre una condición de poda y una condición de retroceso.

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 gráfico 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 p_1 , p_2 , p_3 , p_4 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

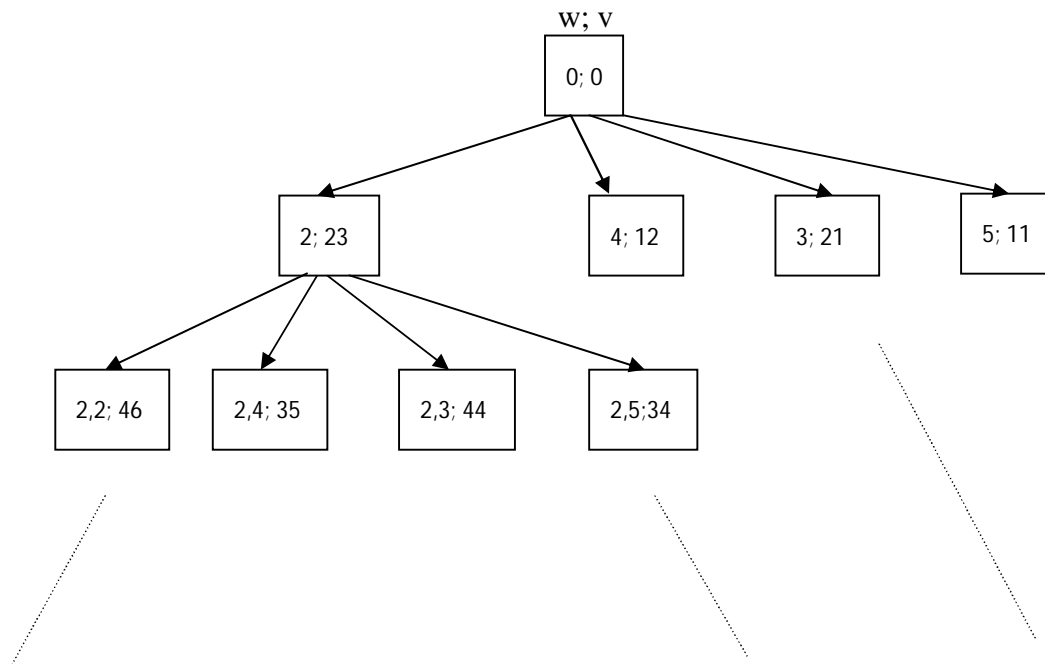
árbol, por lo que sólo lo haremos de un par de niveles. Se puede ver en el resumen de este tema un ejercicio similar a este (problema de la mochila), dando resultados posibles y podando ramas. Por tanto, para completar este ejercicio, se puede ver dicho árbol.

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 conexas 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)

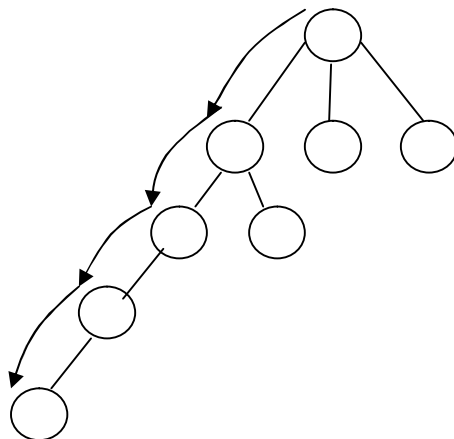
Enunciado: Explique las diferencias entre un recorrido en anchura y un recorrido en profundidad. Exponga un algoritmo iterativo para cada uno de los recorridos explicando las estructuras de datos asociadas, así como su coste.

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 .

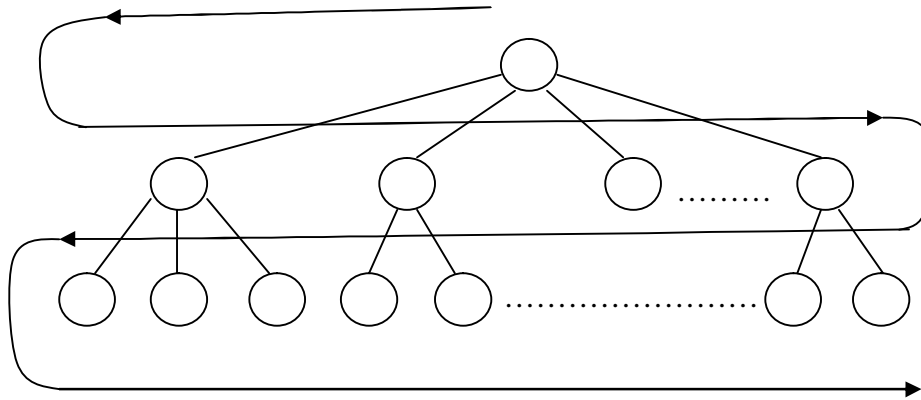
Gráficamente, será algo así:



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)

$P \leftarrow$ pila-vacía

apilar w en P

mientras P no esté vacía hacer

 mientras exista un nodo w adyacente a cima (P)

 tal que marca $[w] \neq$ visitado hacer

 marca $[w] \leftarrow$ 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 $\theta(\max(a, n))$, 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 $N \times N$ 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 $N \times N$ sin que se amenacen entre sí. Si un nodo no cumple esta 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 $N \times N$ sin que se amenacen entre sí” requiere ir solucionando la siguiente secuencia de soluciones parciales:

Poner 1 reina en un tablero de $N \times N$ sin que esté amenazada (trivial).

Poner 2 reinas en un tablero de $N \times N$ sin que se amenacen entre sí.

...

Poner k reinas en un tablero de $N \times N$ sin que se amenacen entre sí.

...

Poner N reinas en un tablero de $N \times N$ 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 fichas 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 $caja[i, j] = \text{cierto}$.
- 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 última 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 ← última-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 recorreremos.

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 $d(x) = x \text{ div } 2$ (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

$$2 = d(d(m(d(7))))$$

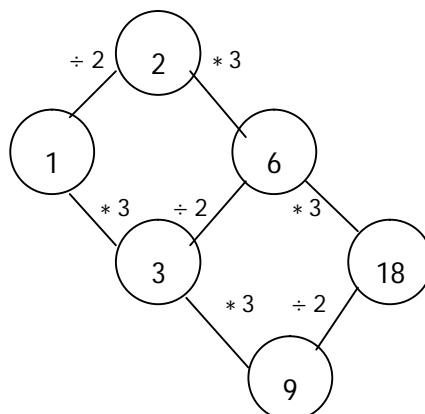
Respuesta:

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-vacia
  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
  índice: 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 tenía 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 esta 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 serían 0 puestos que deben ser válidas) 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
     $(i, j) \leftarrow e.ultima-posición$ 
     $N \leftarrow e.numero-movs$ 
    último-tablero ← e.tablero
    /* Generamos todas las posibles ramas */
    para  $i \leftarrow -2$  hasta 2 hacer
        para  $j \leftarrow -2$  hasta 2 hacer
            si  $(abs(i) + abs(j) = 3) \wedge (e.tablero[i, j] = 0)$  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 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 9 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 cuadrícula 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 cuadrícula 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, cuadrícula.
- Un conjunto de números disponibles. Por ejemplo, candidatos o disponibles.
- Un número natural n que indique cuantos valores hay colocados en la cuadrícula. 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 cuadrícula 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.cuadrícula[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 cuadrícula 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 aquellas 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 3×3 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 cuadrícula.

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 un 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

 registro

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 **$O(ANCHO * LARGO)$** , 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.

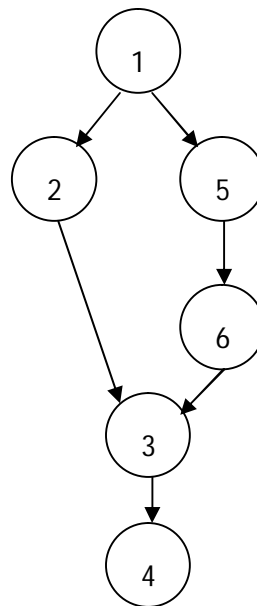
2. Descripción del esquema e identificación con el problema

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[hijo] = 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 (\leftarrow) 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 realizará con el nodo inicial y los elementos del vector de visitados inicializado a falso.

NOTA DEL AUTOR: Recalamos 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

$c * n^k$: Todas aquellas operaciones que hacen falta además de la recursividad. Tendremos, por tanto, $c * n^k = n \Rightarrow n = 1$.

Aplicando la fórmula siguiente para dicha recursión de **reducción por sustracción** tendremos:

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

Como hemos visto previamente el caso será el de $a > 1$, por el que resolviéndolo tendremos que $T(n) \in \theta(a^{n \text{ div } b}) = \theta(n^n)$.

Se puede llegar a esta conclusión razonando sobre la forma del árbol de exploración.

Febrero 2005-1ª (problema)

Enunciado: Partiendo de un conjunto $N = \{n_1, n_2, \dots, n_m\}$ 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:

$$\left(\left(\left((6 * 5) * 4 \right) * 2 \right) * (3 + 1) \right) = 960.$$

Si $P = 970$, el algoritmo encontraría la solución exacta con el conjunto de números inicial y la secuencia más próxima por debajo de P sería 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 aquella 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: $\langle 1,2,3,4 \rangle$

operaciones: $< \begin{array}{|c|} \hline 1 \\ \hline \end{array} \begin{array}{|c|} \hline 2 \\ \hline \end{array} \begin{array}{|c|} \hline 3 \\ \hline \end{array} \begin{array}{|c|} \hline 4 \\ \hline \end{array} >$

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: $\langle 1,6,3 \rangle$

operaciones: $< \begin{array}{|c|} \hline 1 \\ \hline \end{array} \begin{array}{|c|} \hline 2 \\ \hline 4 \\ \hline + \\ \hline \end{array} \begin{array}{|c|} \hline 3 \\ \hline \end{array} >$

solucion: false

vacio: false

Nótese que:

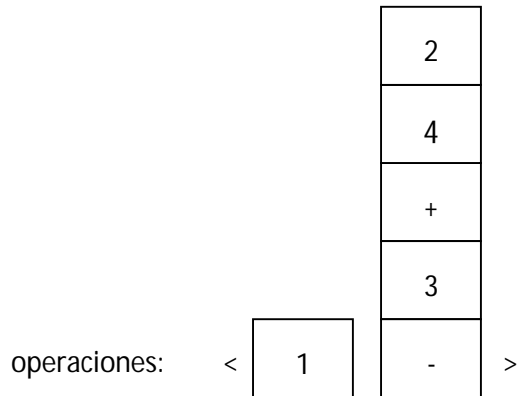
1. Desaparece el elemento que ocupa la posición 3 del vector y su pila asociada.
2. El vector de candidatos ahora almacena el valor 6 allí donde se ha realizado la operación.

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⟩



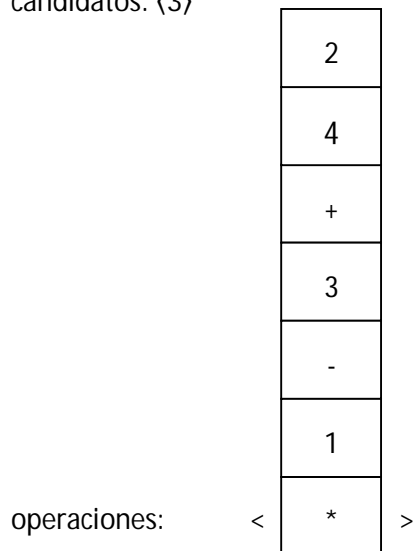
solucion: false

vacio: false

Finalmente, si operamos el candidato 1 con el 3 utilizamos el operador '*'. El ensayo obtenido sería:

ENSAYO

candidatos: ⟨3⟩



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 vacío, 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 (c2Index)
  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
  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 = valorAux
  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.

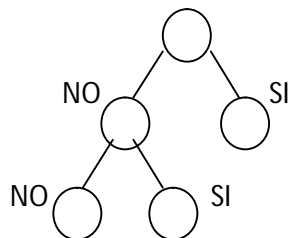
Descripción algorítmica del esquema:

```
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 ( $\text{suma}[1] = \text{suma}[2]$ ) 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 ( $\text{suma}[1] = \text{suma}[2]$ )
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
       $x[k] \leftarrow 1$ 
       $\text{suma}[i] \leftarrow \text{suma}[i] + v[k]$  /* v es el vector de valores */
      si  $\text{suma}[i] \leq (\text{sumaTotal} \text{ DIV } 2)$  entonces
        si  $k < n$  entonces
           $x \leftarrow \text{separacionSociosMitad}(v, k + 1, \text{suma}, \text{sumaTotal})$ 
        fsi
      si no
         $\text{suma}[i] \leftarrow \text{suma}[i] - v[k]$  /* 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: $T(n) \in O(2^n)$.

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.

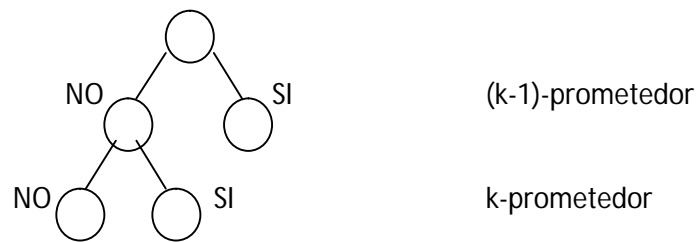
Tenemos que recorrer un árbol de soluciones candidatas, pero siguiendo aquellas ramas que tengan opción de convertirse en una solución (k -prometedoras), por lo que el esquema de **vuelta atrás** es adecuado. También es válido un recorrido del árbol de soluciones en profundidad, siempre que se detenga 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 serían 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:



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  $n^{\text{os}}$  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

$$\text{si } m \rightarrow n \Rightarrow T(n) \in O(n!)$$

$$\text{si } m \rightarrow 1 \Rightarrow T(n) \in O(n)$$

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 (aquel 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)  $\wedge$  (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)  $\wedge$  (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  $m \rightarrow n \Rightarrow T(n) \in O(2^n)$ 
si  $m \rightarrow 1 \Rightarrow T(n) \in O(n)$ 
```

Este algoritmo es más eficiente que el anterior en el caso $m \rightarrow n$.

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 acaba 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 vacío consecutivo, se puede saltar con la primera bola al hueco vacío, 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:

| | | | |
|---|---|--|--|
| O | X | | |
| | | | |
| | | | |
| | | | |

En teoría y según el enunciado tendríamos que cubrir todo con bolas salvo el punto central, que es un vacío, pero para nuestro ejemplo podría valer para ver este juego. Por tanto, el segundo dibujo del que nos hablan sería cuando el O salta y se come al X, siendo ésto lo siguiente:

| | | | |
|--|--|---|--|
| | | O | |
| | | | |
| | | | |
| | | | |

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 río Guadalupe, se encuentran 3 alumnos y 3 profesores del tribunal de la UNED que deben cruzar el río 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 río considerando las siguientes restricciones:

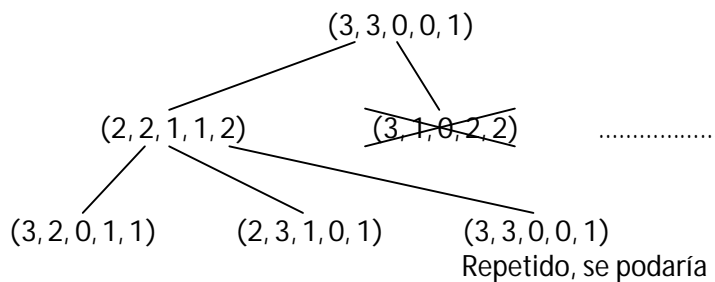
- La canoa no puede viajar sola.
- En ningún momento puede haber grupos en las orillas donde el número de alumnos supere al de los profesores.

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 $n \times m$ 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 | ++O+ | ++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 $n \times n$ cuyas casillas contienen valores del conjunto $C = \{\text{rojo}, \text{verde}, \text{amarillo}\}$ 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 cuadrícula de 81 (9×9) casillas distribuidas a su vez en 9 cajas de 3×3 . 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 cuadrícula. 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 $N \times N$ 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 $M[i, \cdot]$ es una ordenación de mayor a menor de las mujeres cliente según las preferencias del i -ésimo hombre, y la fila $H[i, \cdot]$ 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
    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 $\text{cota-inferior (nodo)} \geq \text{cota-superior}$ 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 vacío 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
    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 ←  $\begin{pmatrix} 30 & 40 & 70 \\ 60 & 20 & 10 \\ 40 & 90 & 20 \end{pmatrix}$ 
```

```
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 +  $\sum_{i=e.ultima-asignacion + 1}^3$  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 pade 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
    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 $O(n!)$, 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). Esta 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 ($\text{coste} = 2 * \text{long_consulta}$) y realizar la **inserción** de los caracteres que faltan ($\text{coste} = 3 * (\text{long_texto} - \text{long_consulta})$). 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 ($\text{coste} = 0$) y realizar tantas inserciones como caracteres nos falten ($\text{coste} = 2 * (\text{long_texto} - \text{nodo.long})$).

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 "sí" 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 $3^{\text{long_texto}}$.

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)**: Posibilidades 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 Ø                                  /* 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 $n!$

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 vacío y, por tanto, se necesitan un máximo de n envases. Las soluciones se representan en tuplas de la forma (x_1, \dots, 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 vacío. La búsqueda podrá acabarse si encontramos una solución con el valor:

$$\text{óptimo} = \left\lceil \frac{\sum_{i=1}^n v_i}{E} \right\rceil$$

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 minimización, 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 sería 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 =  $\lceil total/E \rceil$ ; 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  $\neg$ encontrada  $\wedge$   $\neg$ es-cp-vacia?(C)  $\wedge$ 
    mínimo(C).envases  $\leq$  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  $\leq$  Y.envases  $\wedge$   $\neg$ encontrada hacer
      si X.capacidad[i]  $\geq$  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  $\neg$ 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  $\leq$  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
    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 $O(n!)$, 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 sería 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, \dots, 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.

$$d_1 \leq d_2 \leq \dots \leq d_n$$

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.

$$[h_1, h_2, \dots, h_k, \{h_{k+1}\}, \dots, h_n]$$

\swarrow \nwarrow
 $T = 50 \text{ días}$ $T = 10 \text{ días}$

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
  fmientras
ffun

```

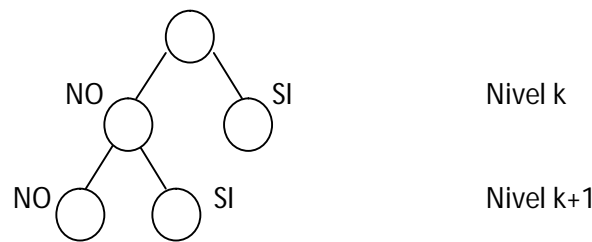
En nuestro caso:

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:



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 esta 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 **$O(n!)$** . 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 $n \times n$ 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 $n \times n$ 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, \dots, 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.