Julio Gonzalo Arroyo
Miguel Rodríguez Artacho

ESQUEMAS ALGORÍTMICOS ENFOQUE METODOLÓGICO Y PROBLEMAS RESUELTOS

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

CUADERNOS DE LA UNED

Julio Gonzalo Arroyo
Miguel Rodríguez Artacho

ESQUEMAS ALGORÍTMICOS ENFOQUE METODOLÓGICO Y PROBLEMAS RESUELTOS

UNIVERSIDAD NACIONAL DE EDUCACIÓN A DISTANCIA

CUADERNOS DE LA UNED (35167CU01A01)
ESQUEMAS ALGORÍTMICOS: ENFOQUE METODOLÓGICO
Y PROBLEMAS RESUELTOS

Librería UNED: C/ Bravo Murillo, 38. 28015 Madrid
Teléfs.: 91 398 75 60/73 73 E-mail: libreria@adm.uned.es

ISBN: 84-362-3622-X
Depósito legal: M. 17.296-2002

Primera edición: noviembre de 1997
Tercera reimpresión: abril de 2002

Impreso en Fernández Ciudad, S. L.
Catalina Suárez, 19. 28007 Madrid
Impreso en España - Printed in Spain

Agradecimientos

No queremos resultar pretenciosos escribiendo una extensa página de agradecimientos en un libro tan modesto, pero nos gustaría que constase nuestra gratitud por la ayuda recibida de algunas de las personas que nos rodean.

Estamos en deuda con María Felisa Verdejo por su orientación constante para el desarrollo de la asignatura. Casi todo lo que hemos aprendido en nuestra labor docente en la UNED, se lo debemos a ella. En particular, sus consejos sobre la estructura de este texto y sobre la idea de presentarlo en este formato han sido decisivos.

También queremos mencionar a María Teresa Abad por la ayuda documental que nos prestó cuando comenzábamos a preparar la asignatura, y a José Ignacio Mayorga por su apoyo y su ayuda prestada durante todo este tiempo y, en especial, durante las primeras andanzas de la asignatura de *Programación III*. Su trabajo nos ayudó a superar aquel difícil obstáculo.

Prefacio

El presente texto consiste en una serie de problemas a los que se da solución mediante su identificación con esquemas algorítmicos conocidos. Está pensado como texto de apoyo[1] para el estudio de la asignatura *Programación III* de segundo curso de la Ingeniería Técnica Informática de la UNED, en las especialidades de Gestión y Sistemas.

Nuestra experiencia en los primeros años de andadura de la asignatura nos ha mostrado que ese texto (como tantos otros) no ofrece suficiente material práctico como para que el alumno pueda contrastar el grado de asimilación de las materias estudiadas. Creemos, sin embargo, que es importante *aprender haciendo*. Por ello hemos decidido recopilar las soluciones a la mayoría de los problemas de examen propuestos en cursos anteriores, junto con otros problemas dirigidos a cubrir de forma homogénea el temario de la asignatura. Junto con ellos hemos realizado un breve resumen de la metodología, herramientas y materias cubiertas en la asignatura, de forma que el alumno pueda tener una referencia sencilla y directa con la que ponerse a resolver problemas de algoritmia una vez estudiados, en un libro de texto usual como es el que recomendamos, los aspectos teóricos esenciales de la asignatura.

Hemos optado por presentar los problemas resueltos en un formato similar (en cuanto a planteamiento y profundidad) al que esperamos en la resolución de un examen. En ocasiones hemos añadido, además, comentarios adicionales que profundizan en aspectos laterales del problema, lo relacionan con otros problemas o sirven como aclaración adicional. Estos comentarios que no forman parte esencial de la resolución, pero facilitan el estudio de la asignatura, están tipografiados con otro tipo de letra para distinguirlos de las soluciones propiamente dichas.

[1] El texto base de la asignatura es el Brassard, G. y Bratley, P. 1997. *Fundamentos de Algoritmia*. Prentice Hall International.

Este texto sólo será útil a los alumnos una vez superados los primeros temas introductorios sobre algoritmos en general, complejidad y estructuras de datos. Nuestra recomendación, sin embargo, es que los estudiantes no se detengan en exceso en el estudio de estas materias en una primera aproximación: es al aplicarlas al estudio práctico de los esquemas algorítmicos cuando tendrán ocasión de volver a estudiar y asimilar definitivamente esos conceptos introductorios. En ocasiones, la preocupación de los estudiantes por entender hasta el último detalle de estos temas introductorios antes de pasar a la práctica hace que se descuide el estudio de los esquemas fundamentales (voraz, vuelta atrás y algoritmos de exploración en grafos). Hay que tener en cuenta que es en esta segunda parte de la asignatura cuando los conceptos discutidos en la primera se clarifican, al aplicarlos en el análisis práctico de algoritmos.

Para aquellos que dispongan del texto base y de esta colección de problemas, proponemos el siguiente esquema de estudio:

- Temas 1 y 2: Preliminares y algoritmia elemental. Una lectura detallada debe ser suficiente. En general, se trata de refrescar conocimientos que ya se habían adquirido previamente en otras asignaturas.
- Temas 3 y 4: Notación asintótica y análisis de algoritmos. Deben refrescarse las ideas esenciales que se habían adquirido en Programación II. El tema cuatro incluye consideraciones y razonamientos formales acerca del coste que, aún siendo interesantes, no deben distraer al estudiante del objetivo esencial de estos temas, que es disponer de las herramientas necesarias para analizar el coste de un algoritmo. Si se encuentran dificultades en comprender parte de estos temas, puede posponer su estudio al momento en que se enfrente con los esquemas algorítmicos que se estudian en la parte central de la asignatura (y, en particular, al momento en el que se enfrente con esta colección de problemas). En este mismo texto ofrecemos un resumen de las herramientas que deben retener para analizar el coste de los algoritmos que se estudiarán en esta asignatura.
- Tema 5: Estructuras de datos. Es importante que se tenga una idea muy clara de las estructuras de datos que se habrán de utilizar en el resto de la asignatura, y de las distintas formas en que pueden implementarse. Para el núcleo central de la asignatura, el alumno debe retener las funciones principales de acceso y modificación de las estructuras de datos más relevantes, pero puede obviar los detalles de implementación a casi todos los efectos (a veces hay que tenerlos en cuenta cuando la implementación de las estructuras de datos es determinante para hallar el coste de un algoritmo).
- Temas 6, 8 y 9: Algoritmos voraces, divide y vencerás y algoritmos de exploración de grafos. Estos temas deben estudiarse a fondo, y para cada uno de ellos deben intentar resolverse todos los problemas que aquí planteamos. Las soluciones sólo deben consultarse después de haber dedicado un tiempo razonable (más de media hora en la mayoría de los casos) a su resolución. Así se conseguirá la agilidad necesaria en todos los aspectos del planteamiento de un algoritmo que esta asignatura contempla.

Por tanto, cuando el alumno se enfrente a un problema ha de encajarlo dentro de uno de estos esquemas: voraz, divide y vencerás o exploración ciega de un grafo (mediante búsqueda en profundidad, en anchura o ramificación y poda). Aunque el estudio de los esquemas algorítmicos podría abarcar muchos otros esquemas, hemos creido suficientes estos tres para adquirir una metodología precisa y segura que debe ser extendible sin problemas a otros esquemas.

Este libro puede ser también de utilidad para estudiantes de otras universidades interesados en esquemas algorítmicos. En particular, hemos puesto un cuidado especial en la modularización entre esquemas, algoritmos particulares y estructuras de datos. Además de simplificar el estudio, de esta manera se simplifica también su implementación. Por ejemplo, todos los problemas de vuelta atrás pueden implementarse con el mismo programa genérico, que llama en cada caso particular a funciones *compleciones*, *condiciones de poda* y *solución* diferentes. Si se quieren utilizar implementaciones distintas para las estructuras de datos, sólo es necesario invocar módulos de implementación diferentes para esas estructuras.

Índice General

Capítulo 1

Metodología

Al acercarse al estudio de los esquemas algorítmicos, se debe aprender a resolver problemas mediante su clasificación en familias predefinidas para las que existen soluciones generales. Efectivamente, la algoritmia no consiste en dar soluciones puntuales y aisladas a problemas concretos, sino que identifica técnicas generales (*esquemas algorítmicos*) capaces de dar solución a un gran número de problemas. El conocimiento y manejo de estos esquemas es, pues, una herramienta fundamental a la hora de dar solución computacional a muchos problemas y, por tanto, a la hora de programar.

En este texto hemos puesto un énfasis especial en proporcionar un método uniforme y fiable con el que enfrentarse a la resolución de problemas de algoritmia. Una aproximación metódica y modular no sólo simplifica enormemente la especificación de algoritmos, sino también su implementación posterior en cualquier lenguaje de programación. En este sentido, hemos hecho un esfuerzo por no mezclar esquemas generales, por un lado; particularizaciones de esos esquemas a problemas concretos, por otro; e implementación de estructuras de datos, por otro.

En este capítulo veremos brevemente el método general para resolver problemas mediante esquemas algorítmicos. También resumiremos los esquemas particulares que forman parte de la materia de *Programación III*, particularizando esa metodología para cada esquema y también ofreciendo algunos consejos de utilidad al lidiar con cada uno de ellos.

1.1 Método general de resolución

El esquema general para resolver problemas de esquemas algorítmicos que adoptamos en este texto es el siguiente:

Elección del esquema Todos los problemas que proponemos pueden verse como instancias de algún esquema algorítmico estudiado. El primer paso para resolverlos es, por tanto, decidir cual es el esquema más apropiado para el problema en cuestión.

Ha de ponerse un cuidado especial en dar las razones positivas por las que se escoge un esquema en particular. Como la decisión ha de tomarse entre tres esquemas básicos, se puede caer en la tentación de seleccionar un esquema por eliminación del resto; obviamente, no debe ser así. En general, es necesario identificar los elementos esenciales del esquema en el problema que nos ocupa, dejando sólo los detalles para el paso siguiente.

En el caso de que haya más de un esquema aplicable, debe escogerse aquel que se aplique de forma natural al problema, y, en cualquier caso, el que previsiblemente tenga un coste menor. Al comentar los esquemas uno a uno comentaremos los conflictos más usuales entre ellos.

Identificación del problema con el esquema elegido Se identifica en detalle cada uno de los aspectos del problema con el esquema algorítmico elegido. Para ello hay que especificar quienes son los ejemplares del problema y cómo se particularizan para el problema cada una de las funciones que forman parte del esquema.

Estructuras de datos Se menciona la estructura de los ejemplares del problema y la de cualquier otro dato relevante para su resolución. Las estructuras básicas: listas, conjuntos, grafos, vectores, etc. se dan por conocidas; no es necesario entrar en detalles sobre su implementación. En caso de que sea relevante para el problema, las distintas implementaciones posibles se discutirán al evaluar el coste del algoritmo.

Algoritmo completo En la medida de lo posible, no se reescribirá el esquema para ajustarlo al problema. El esquema general puede ser el cuerpo del algoritmo si se especifican las funciones que utiliza el esquema y que deben ser particularizadas según se ha discutido previamente. Aún siendo este apartado en el que se da verdaderamente el algoritmo, debería ser el más sencillo de todos si el estudio preliminar se ha hecho completo y con corrección.

Estudio del coste El coste del algoritmo se discute siempre en el caso peor, salvo que se indique lo contrario. En ocasiones, distintas implementaciones de las estructuras de datos pueden modificar el coste de un algoritmo; en ese caso, debe discutirse cual es la implementación más adecuada.

1.2 Algoritmos voraces

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

En notación algorítmica puede escribirse así:

```
fun voraz(C:conjunto) dev (S: conjunto)
S ← ∅
mientras ¬ solución(S) ∧ C≠ ∅ hacer
  x ← seleccionar(C)
  C ← C \ {x}
  si completable( S ∪ {x})
  entonces S ← S ∪ {x}
  fsi
fmientras
dev S
ffun
```

Las funciones que hay que particularizar para cada problema son:

1. *solución(S)*. Decide si el conjunto S, donde se almacenan los candidatos seleccionados, es o no una solución.
2. *seleccionar(C)*. De entre los candidatos restantes en cada iteración del bucle, selecciona el más prometedor según el criterio que recoge la función *objetivo*. Esta función es la que determina el comportamiento del algoritmo.

3. *completable.* Determina si un conjunto de elementos seleccionados puede llegar a ser una solución o no.

Para cada uno de los aspectos de la resolución de un problema mediante el esquema voraz pueden tenerse en cuenta los siguientes aspectos:

Elección del esquema Para aplicar el esquema voraz, el problema debe ser de optimización. Además, se debe poder distinguir un conjunto de candidatos de forma que la solución pase por ir escogiéndolos o desechándolos en un cierto orden.

Para identificar un problema como voraz es necesario, además, tener una idea clara de cómo se va a resolver el problema. La función de selección que tendremos que especificar debe asegurar que la solución que se alcanza es la óptima. Si no se tiene esa seguridad, es muy posible que el algoritmo que propongamos no resuelva realmente el problema, y que lo hayamos confundido con un problema de búsqueda en grafos. Veamos un ejemplo.

Problema: Las máquinas de bebidas, tabaco, etc., han de devolver una cierta cantidad de dinero cuando no se introduce el precio exacto. Diseñar un algoritmo que, dada una cantidad, la devuelva utilizando el número mínimo de monedas. Se dispone de una cantidad ilimitada de monedas de 1, 5, 25 y 100 pesetas.

La solución es un algoritmo voraz que toma en cada momento la moneda más grande dentro de las que no superan la cantidad total una vez acumuladas a las ya escogidas. Sin embargo, supongamos que las monedas disponibles son las inglesas, antes de su normalización decimal: 1, 3, 6, 12, 24 y 30 peniques. Es tentador pensar que el mismo algoritmo resolverá en este caso el problema... Pero falso. Por ejemplo, si han de devolverse 48 peniques, nuestro algoritmo escogería la secuencia 30-12-6. Pero existe una secuencia más corta: 24-24.

Identificación con el problema Es necesario determinar quienes son los candidatos, la función de selección que nos dice en cada momento qué candidato escoger, la función solución que comprueba si la hemos alcanzado, y la función factible, que comprueba si el conjunto de candidatos seleccionados puede llegar a convertirse en solución o no.

Estructuras de datos Hay que especificar, al menos, la estructura de datos correspondiente a los elementos (candidatos) que componen el problema.

El conjunto de candidatos puede representarse normalmente mediante un conjunto, salvo que tenga algún otro tipo de estructura intrínseca. A veces, sin embargo, la solución no consiste simplemente en un conjunto, sino que el orden en que se introducen los candidatos en el conjunto solución es relevante. En ese caso es mejor representarlo mediante una lista. Recordad que no es necesario discutir a priori la implementación que se da a listas y conjuntos.

Por último, se debe discutir cuál es la estructura de datos que corresponde a los candidatos.

Algoritmo completo En principio, casi siempre es posible dar el algoritmo completo especificando simplemente las funciones `solución`, `seleccionar` y `factible`. En ese caso, no es necesario reescribir el algoritmo.

Estudio del coste Los algoritmos voraces son iterativos y, por tanto, evaluar su coste es sencillo. Normalmente el coste del bucle voraz es el que determina el coste global del algoritmo.

En el bucle voraz, una de las operaciones principales consiste en escoger, del conjunto de candidatos, aquel que minimiza o maximiza una función objetivo. Este es el momento de discutir la implementación de las estructuras de datos, y en particular del conjunto de candidatos: si lo representamos como un montículo de mínimos (o máximos) según esa función objetivo, la selección tendrá coste constante. Al retirar el elemento seleccionado del montículo, hay que restaurarlo, lo que tiene un coste $\mathcal{O}(\log n)$. Es muy posible que la combinación de esas dos operaciones sea más eficiente que el resto de las implementaciones posibles para el conjunto.

Demostración de optimalidad Una argumentación clara debe ser suficiente. Es interesante tener presentes las técnicas de demostración (reducción al absurdo e inducción matemática) que se revisan en texto base, capítulo primero.

1.3 Divide y vencerás

El esquema divide y vencerás es una técnica recursiva que consiste en dividir un problema en varios subproblemas del mismo tipo. Las soluciones a estos subproblemas se combinan a continuación para dar la solución al problema. Cuando los subproblemas son más pequeños que un umbral prefijado, se resuelven mediante

un algoritmo específico. Si su tamaño es mayor, se vuelven a descomponer. El esquema es el siguiente:

```
fun divide_y_venceras (problema)
  si suficientemente-simple (problema)
  entonces dev solucion-simple (problema)
  si no hacer
   { p_1 ... p_k } ← descomposicion(problema)
   para cada p_i hacer
   s_i ← divide_y_venceras(p_i)
   fpara
   dev combinacion(s_1 ... s_k)
  fsi
ffun
```

Las funciones que han de particularizarse para un problema concreto son, por tanto:

- *suficientemente_simple.* Decide si un problema está por debajo del tamaño umbral o no.
- *solucion_simple.* Algoritmo para resolver los casos más sencillos, por debajo del tamaño umbral.
- *descomposición.* Descompone el problema en subproblemas de tamaño menor.
- *combinación.* Algoritmo que combina las soluciones a los subproblemas en la solución al problema del que provienen.

A continuación comentamos las distintas partes de la resolución de un problema en el caso de algoritmos divide y vencerás.

Elección del esquema Para resolver un problema mediante divide y vencerás, es fundamental ver claramente cómo un problema puede dividirse en *subproblemas idénticos* a los que pueda aplicarse exactamente el mismo procedimiento. Además, debe tenerse una idea más o menos clara de cómo se van a combinar las soluciones a los subproblemas.

Identificación con el problema Es necesario especificar cómo se divide el problema en subproblemas y cómo se combinan éstos. Además hay que

decidir cual será el tamaño umbral y qué función solucionará los problemas por debajo de ese tamaño umbral. El tamaño umbral no influye, en principio, en el coste del algoritmo (sólo en un factor constante), de modo que lo más útil en una primera aproximación es tomar como tamaño umbral aquel que hace más sencilla la solución trivial. Casi siempre se trata de $n = 1$ (siendo n el tamaño del problema).

También ha de comentarse cual es el preorden bien fundado para los problemas , es decir, hay que garantizar que la forma de dividir un problema en subproblemas conduce, efectivamente, a problemas más sencillos, y que esa división siempre termina en el tamaño trivial.

Estructuras de datos Hay que especificar en qué consiste un problema y, en su caso, los elementos del problema.

Algoritmo completo Casi siempre se reduce a especificar las funciones de simplificación y combinación, junto con otra función *suficientemente-simple* que determina si se ha alcanzado el tamaño umbral y otra *solución-simple* que resuelve ese caso.

Estudio del coste Al tratarse de algoritmos recursivos, es necesario plantear la ecuación de recurrencia. Al estar dividiendo el problema, normalmente se podrá aplicar la fórmula para reducciones por división:

$$T(n) = \begin{cases} cn^k & \text{, si } 1 \le n < b \\ aT(n/b) + cn^k & \text{, si } n \ge b \end{cases}$$

$$\Rightarrow$$

$$T(n) \in \begin{cases} \Theta(n^k) & \text{, si } a < b^k \\ \Theta(n^k \log n) & \text{, si } a = b^k \\ \Theta(n^{\log_b a}) & \text{, si } a > b^k \end{cases} \qquad (1.1)$$

Por tanto, hay que plantear la ecuación de recurrencia para $T(n)$ e identificar en ella las constantes a, b y k.

1.4 Algoritmos de exploración en grafos

Hay una cantidad enorme de problemas que se pueden formular en términos de una búsqueda ciega en grafos. Es decir, la solución es uno de los nodos, pero no hay ningún orden o procedimiento establecido para llegar directamente a ese nodo: es necesario explorar exhaustivamente el grafo hasta encontrarlo. Este tipo de algoritmos realizan distintos tipos de búsqueda sistemática en grafos

(muy a menudo árboles), complementada con criterios que permiten *podar*, o desechar antes de ser exploradas, las ramas que los satisfacen. A estos criterios se les llama *condiciones de poda*.

1.4.1 Búsqueda en profundidad

Consiste en comenzar por la raíz y, en cada momento, explorar el hijo más a la izquierda (o a la derecha, es sólo una convención) que no haya sido aún explorado. De este modo, si un nodo tiene varios hijos, hasta que no se ha explorado totalmente la rama a la que da lugar el primer hijo no se entra al segundo, y así sucesivamente. El último nodo examinado será la hoja de la rama más a la derecha del árbol.

Vuelta atrás

El esquema de backtracking o vuelta atrás realiza una exploración en profundidad mediante un algoritmo recursivo. Para cada nodo, vuelta atrás genera sus compleciones (es decir, sus hijos directos) y aplica de nuevo vuelta atrás sobre cada uno de ellos. Si en algún punto del recorrido se dan condiciones que hagan inútil seguir el camino tomado (condiciones de poda) se abandona esa rama, retrocediendo a la última decisión. El esquema general es el siguiente:

```
fun vuelta-atrás (ensayo)
   si válido (ensayo)
   entonces   dev ensayo
   si no   para hijo ∈ compleciones(ensayo)
           si condiciones-de-poda(hijo)  hacer  vuelta-atrás(hijo)  fsi
   fsi
ffun
```

Para un problema dado, es necesario especificar

- Qué es un *ensayo*, es decir, un nodo del árbol.
- La función *válido*, que determina si un nodo es solución al problema o no.
- La función *compleciones*, que genera los hijos de un nodo dado.
- la función *condiciones-de-poda* que verifica si puede descartarse 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.

Tal y como lo recogemos, el algoritmo de vuelta atrás no se detiene al encontrar una solución, sino al terminar de explorar el árbol, es decir, después de haber encontrado todas las soluciones. Si sólo se necesita encontrar una solución cualquiera, o si el árbol de búsqueda es infinito, hay que modificar el esquema algorítmico para que finalice al encontrar la primera solución:

```
fun vuelta-atrás* (ensayo)
   si válido (ensayo)
   entonces   dev ensayo
   si no
   lista-ensayos ← compleciones(ensayo)
    mientras ¬ vacia(lista-ensayos) ∧ ¬ resultado  hacer
    hijo ← primero(lista-ensayos)
    lista-ensayos ← resto(lista-ensayos)
 si condiciones-de-poda(hijo)  hacer resultado ← vuelta-atrás*(hijo)  fsi
    fmientras
    dev resultado
   fsi
ffun
```

Búsqueda en profundidad iterativa

La exploración en profundidad que realiza implícitamente el esquema de vuelta atrás puede también realizarse mediante un algoritmo iterativo, utilizando una pila para mantener los nodos generados y aún no visitados:

```
fun búsqueda-en-profundidad (ensayo)
 p ← pila-vacía
 apilar(ensayo,p)
  mientras ¬vacía(p)  hacer
  nodo ← desapilar(p)
   si válido(nodo)  entonces   dev nodo
   si no
    para cada hijo en compleciones(nodo)  hacer
     si condiciones-de-poda(hijo)  entonces  apilar(hijo,p)  fsi
```

```
      fpara
    fsi
  fmientras
ffun
```

Nótese que *compleciones*, *condiciones-de-poda* y *válido* son funciones idénticas a las del esquema anterior. Para pasar de una implementación a otra sólo habría que cambiar el cuerpo del programa.

1.4.2 Búsqueda en anchura

En lugar de profundizar en cada rama que generamos, puede explorarse también el árbol mediante un recorrido *en anchura*, en el que se visitan todos los nodos de un nivel dado antes de pasar al siguiente nivel. Dicho de otra manera: se explora primero la raíz, a continuación sus hijos, luego los hijos de los hijos, etc.

Este tipo de búsqueda puede realizarse mediante un algoritmo idéntico a la búsqueda en profundidad iterativa, sustituyendo la pila por una cola (ahora el primero que entra es el primero que sale, al contrario que en la pila). Al encolar todas las compleciones de un nodo a la vez, estas se visitarán consecutivamente:

```
fun búsqueda-en-anchura (ensayo)
 p ← cola-vacía
 encolar(ensayo,p)
  mientras ¬vacía(p) hacer
  nodo ← desencolar(p)
   si válido(nodo) entonces dev nodo
   si no
    para cada hijo en compleciones(nodo) hacer
     si condiciones-de-poda(hijo) entonces encolar(hijo,p) fsi
    fpara
   fsi
  fmientras
ffun
```

Mediante la búsqueda en anchura nos aseguramos de que la primera solución encontrada tendrá una profundidad mínima en el árbol; en determinados problemas este puede ser un requisito del enunciado.

1.4.3 Ramificación y poda

En ocasiones se requiere que la solución alcanzada sea óptima. En ese caso, se calcula para cada nodo una cota inferior del posible valor de aquellas soluciones que pudieran encontrarse a partir de ese nodo en el grafo. Si la cota muestra que cualquier solución que se halle a partir de ese nodo será necesariamente peor que la mejor solución de la que disponemos, entonces no es necesario seguir explorando esa parte del grafo.

El cálculo de cotas se puede usar como una condición más de poda, o utilizarlo también para guiar la búsqueda: en lugar de buscar en profundidad o en anchura, seleccionamos en cada momento el nodo más prometedor de entre los que quedan por examinar.

En caso de que se utilice como una condición adicional de poda, pongamos por ejemplo en la búsqueda en anchura:

```
fun ramificación-y-poda-en-anchura (ensayo)
 p ← cola-vacía
 c ← coste-mínimo
 solución ← solución-vacía
 encolar(ensayo,p)
  mientras ¬vacía(p) hacer
  nodo ← desencolar(p)
   si válido(nodo) entonces hacer
    si coste(nodo) < c entonces hacer
     solución ← nodo
     c ← coste(nodo)
    fsi
   si no
    para cada hijo en compleciones(nodo) hacer
     si condiciones-de-poda(hijo) y cota(hijo) < c
      entonces encolar(hijo,p)
     fsi
    fpara
   fsi
  fmientras
ffun
```

Para cada problema en particular hay que especificar la función *cota* y la

función *coste* (que no hace sino devolver la cota para un nodo solución).

En caso de que la cota se utilice también para determinar el orden en que se visitan los nodos, ya no podemos usar pilas ni colas, ya que debemos tener la posibilidad de extraer cualquier elemento. Como siempre vamos a buscar el que minimice la cota, podemos utilizar un montículo de mínimos; de esta forma, la raíz será siempre el más adecuado en cada momento:

```
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-raiz(m)
    si válido(nodo) entonces hacer
      si coste(nodo) < cota-superior entonces hacer
        solución ← nodo
        cota-superior ← coste(nodo)
      fsi
      si no
        si cota-inferior(nodo) ≥ cota-superior entonces
          dev solución
        si no
          para cada hijo en compleciones(nodo) hacer
            si condiciones-de-poda(hijo) y cota-inferior(hijo) < cota-superior
              entonces añadir-nodo(hijo,m)
            fsi
          fpara
        fsi
      fsi
    fmientras
  ffun
```

A continuación comentamos más en detalle la metodología general para el caso de los algoritmos de búsqueda ciega en árboles.

Elección del esquema Hay una enorme cantidad de problemas que pueden interpretarse en términos de una búsqueda ciega en un árbol, debido a que estos algoritmos no son más que versiones sofisticadas del "prueba y error". Por eso mismo suelen ser algoritmos de coste muy elevado, y debe restringirse su uso a los casos en que ninguno de los demás esquemas estudiados es aplicable.

Una vez descartados el esquema voraz y el divide y vencerás, para encajar un problema como búsqueda ciega en un árbol hay que tener claro en qué consiste ese grafo: cuáles son sus elementos y cómo pasar de un nodo dado a sus hijos. A menudo, el árbol no existe como estructura de datos, sino que se trata de un árbol "virtual"que se va generando según se explora, y cuya estructura está implícita en la acumulación de llamadas recursivas o en la pila de nodos explorados y por explorar.

Una vez entendido el problema en términos de una búsqueda ciega, decidir si se busca en profundidad o en anchura, recursiva o iterativamente, suele ser menos comprometido. En principio el algoritmo de vuelta atrás es, quizás, el más intuitivo, pero por lo demás se puede escoger cualquiera si los datos del problema no demandan algún tipo de búsqueda en particular. En cualquier caso, la especificación de la mayoría de las funciones que intervienen es independiente de la variante elegida.

Si además de encontrar una solución, ésta ha de ser óptima, entonces será necesario aplicar el esquema de ramificación y poda.

Identificación con el problema Hay que identificar los nodos del árbol, cómo se decide si un nodo es solución, y cómo se hallan los hijos de un nodo (sus `compleciones`. Además, pueden enumerarse las condiciones de poda en las que la búsqueda por una rama puede interrumpirse porque no puede conducir a ninguna solución. Si se trata de un problema de ramificación y poda, hay que explicar cómo se estima la cota para un nodo dado.

Estructuras de datos Cuando se trata de un algoritmo de vuelta atrás (es decir, de una búsqueda en profundidad mediante un algoritmo recursivo), suele ser suficiente con dar la estructura de los nodos (o ensayos). Si, por el contrario, se trata de una búsqueda iterativa, hay que comentar el uso de una pila (en el caso de búsqueda en profundidad) o de una cola (en el caso de búsqueda en anchura), sin que sea necesario entrar en detalles sobre su implementación. De igual modo, si se trata de un problema de ramificación y poda visitando en cada momento el nodo más prometedor, hay que anotar el uso de un montículo de mínimos.

A veces también es necesario guardar memoria de los nodos visitados; una lista suele ser lo más adecuado; aunque no son muy eficientes, las búsquedas ciegas en grafos suelen ser tan costosas que otras contribuciones al coste se vuelven irrelevantes.

Algoritmo completo Hay que especificar la función `compleciones`, la función `solución` y, en su caso, la función `condiciones de poda`. Por lo demás, es el esquema general escogido el que decide si la exploración es en profundidad o en anchura, recursiva o iterativa. En el caso de los algoritmos de ramificación y poda, hay que dar también las funciones cota y coste.

Estudio del coste En la mayoría de las ocasiones no se puede dar de forma exacta el coste de una búsqueda ciega en un grafo. Una buena estimación,sin embargo, suele ser el tamaño del árbol de búsqueda en función del tamaño del problema.

1.5 Estructuras de datos más útiles

En los problemas de esquemas algorítmicos es necesario utilizar continuamente algunas estructuras de datos básicas. Un conocimiento previo de las posibles estructuras más comunes, así como de los métodos o funciones asociadas a ellas, nos van a facilitar el diseño de un algoritmo.

La implementación de las estructuras de datos se realiza a partir de un *Tipo Abstracto de Datos*, que define el comportamiento de la estructura de datos sin concretar los aspectos relativos a su implementación en algún lenguaje de programación; de esta manera es posible describir las primitivas de acceso y manipulación de dicha estructura que sirvan de manera general, con las únicas diferencias de que el coste computacional de determinadas manipulaciones dependiendo de una implementación u otra puede variar considerablemente. La finalidad es tratar a la estructura de datos como un objeto dotado de métodos para el acceso y manipulación, pero siguiendo el modelo computacional de *caja negra*, según la terminología utilizada en la programación orientada a objetos, de manera que el trato con la estructura de datos elegida se hace mediante dichos métodos y sólo con ellos. De esta forma la estructura ofrece al algoritmo una única interfaz independientemente de la implementación que se tenga, permitiendo su reutilización.

Por ejemplo: Un tipo de datos *grafo* nos describe un objeto dotado de

métodos como por ejemplo funciones que nos den los sucesores de un vértice, el grado de éste, el peso o coste asociado a una arista, etc. En un algoritmo es más conveniente utilizar una declaración de variable $g : grafo$, en lugar de $g : vector[1 \ldots N, 1 \ldots N]$, que correspondería a una implementación como matriz de adyacencia. Si necesitamos hallar el peso asociado a la arista que une los nodos n_1 y n_2 en un grafo g, se utiliza el método o función $peso(g, n_1, n_2)$, en lugar de utilizar directamente $g[n_1, n_2]$.

Al obviar la implementación de las estructuras de datos, es más fácil, en general, escribir el algoritmo, así como razonar a posteriori sobre cuál es la implementación más adecuada para ese algoritmo en concreto, cómo varía el coste con cada implementación, etc.

Además, la primera declaración nos permite implementar un grafo de varias maneras distintas sin modificar nuestro código, y haciendo transperente a éste la implementación interna. Esto sigue siendo cierto al traducir los algoritmos a un lenguaje concreto de programación: para modificar las estructuras de datos con las que implementamos un algoritmo sólo es necesario cambiar la llamada al módulo que implementa el tipo de datos, conservando intacto el resto del programa.

En lo que resta de capítulo se va a describir cómo se organiza la interfaz que ofrecen algunos de las estructuras más utilizadas. No se pretende, sin embargo, ser exhaustivos ni dar una visión completa, sino dar unas líneas generales para la creación de otras estructuras de datos.

1.5.1 Funciones de manipulación de un tipo de datos

La descripción completa de una estructura de datos debe incluir necesariamente la descripción de las funciones de manipulación de los elementos de datos considerados. Al definir un árbol, una pila, un conjunto, etc. estamos definiendo una colección de elementos diferentes. Es evidente que la inserción, búsqueda, consulta, acceso, etc. de elementos en cada una de dichas estructuras puede ser totalmente diferente dependiendo del tipo de datos descrito y también de la implementación que hagamos de dicha estructura en el lenguaje de programación que utilicemos. Lo que sí es claro es que sea cual sea la implementación de dichas funciones, la portabilidad entre los diversos lenguajes está asegurada siempre que independientemente de la implementación, la interfaz que nos ofrece la estructura definida sea la misma.

La manipulación de cualquier tipo de datos se compone, comúnmente, del

siguente tipo de funciones:

Creación Se crea y se ubica la memoria necesaria para una estructura vacía.

Modificación Operaciones de acceso a la estructura de datos que pueden añadir o quitar elementos.

Consulta Acceden a la estructura de datos para consultar propiedades de ésta, recabar información sobre el estado de la estructura o sobre el valor de sus componentes.

En las secciones siguientes se describen estas y otras funciones asociadas a las listas, pilas, colas, conjuntos, grafos y montículos.

Es importante indicar que estas funciones necesitarán o no ser implementadas dependiendo del lenguaje de programación utilizado. No siempre será necesario implementar un conjunto (por ejemplo PASCAL ó MODULA-2 incorporan como básico este tipo de datos), o una lista (en el lenguaje funcional LISP es la estructura de datos por defecto) etc. Lo que se pretende es hacer hincapié en la necesidad de diferenciar la manipulación de los datos del resto del algoritmo. Tampoco es intención de este texto el de cubrir completamente la descripción de las funciones o su implementación, pero sí lo es el de incidir en la importancia que para la comprensión de un programa tiene el tratar con dichas estructuras utilizando únicamente las funciones de manejo existentes.

1.5.2 Pilas

Definición

Una pila es una lista con dinámica LIFO [1]. La forma de insertar y recuperar elementos hace que el primero en entrar sea el último en salir.

Utilización

Una pila es de utlildad cuando sea importante conservar el orden de inserción y extracción según la propiedad LIFO. Por el contrario, no es útil si el acceso a los elementos es indiscriminado o si debe existir algun tipo de orden en ella dependiendo del valor de los elementos.

[1] Last In First Out

Implementaciones

Lista enlazada Colección de registros que contienen el elemento de información y un apuntador al siguiente.

Vectores Un vector contiene los elementos de información. Se utiliza si podemos estimar el número máximo de elementos que albergamos.

Funciones de manipulación

Las funciones de manipulación de una pila son:

- Creación
  - **fun pila-vacía() dev p:pila** Devuelve una pila vacía.
- Modificación
  - **fun apila(e:elemento, p:pila) dev p:pila** Añade el elemento a la pila.
  - **fun desapila(p:pila) dev e:elemento** Devuelve el elemento situado en la cima de la pila y lo borra de ésta.
- Consulta
  - **fun vacía(p:pila) dev b:boolean** Comprueba si en la pila hay algún elemento.
  - **fun llena(p:pila) dev b:boolean** Comprueba si en la pila no caben más elementos.
  - **fun altura(p:pila) dev n:natural** Número de elementos en la pila.

El coste asociado a las operaciones según la implementación es:

	vectores	*punteros*
pila-vacía	cte	cte
apilar	cte	cte
desapilar	cte	cte
vacía	cte	cte
llena	cte	cte
altura	cte	cte

1.5.3 Colas

Definición

Una cola es una lista con dinámica FIFO [2]. Los elementos se insertan en la cola y la recuperación se efectúa en el mismo orden de inserción. El primero en entrar es el primero en salir.

Utilización

Una cola es de utilidad cuando resulte relevante que el orden de inserción y extracción en la estructura se realice según la propiedad FIFO. Análogamente al caso de las pilas, no es de utilidad usar esta estructura si el acceso a los elementos es indiscriminado.

Implementaciones

Las implementaciones posibles son las mismas que en el caso de las pilas.

En caso de utilizar vectores hay que indicar, sin embargo, la utilidad del álgebra modular para la implementación de las colas. Suponiendo que el vector es de tamaño n, las operaciones de inserción y borrado pueden hacerse utilizando el álgebra modular con módulo n.

Funciones de manipulación

En la práctica son aplicables las funciones utilizadas para las listas, aunque formalmente deben utilizarse las que se indican aquí:

- Creación
  - **fun cola-vacía() dev c:cola** Devuelve una cola vacía.
- Modificación
  - **fun encolar(e:elemento, c:cola) dev c:cola** Inserta el elemento en la cola.
  - **fun desencolar(c:cola) dev e:elemento** Devuelve el elemento situado al comienzo de la cola y lo borra de ésta.

[2] First In, First Out: el primero en entrar es el primero en salir.

- Consulta
  - **fun vacía(c:cola) dev b:boolean** Comprueba si en la cola hay algún elemento.
  - **fun llena(c:cola) dev b:boolean** Comprueba si en la cola no caben más elementos.

El coste asociado a las operaciones según la implementación es:

	vectores	*punteros*
cola-vacía	cte	cte
borrar	cte	$\mathcal{O}(n)$
encolar	cte	cte
desencolar	cte	cte
vacía	cte	cte
llena	cte	cte

1.5.4 Listas

Definición

Una lista es una colección de elementos de información organizados unidimensionalmente. Se accede a ellos mediante referencia a partir de su anterior o su siguiente y no se contempla en general un acceso aleatorio eficiente a los elementos de la lista. ES DINÁMICA (CON PUNTEROS)

Utilización

El uso más general es el de almacenar elementos de información sin poder estimar el número máximo de elementos de que disponemos. Si la forma de acceso está bien definida es posible que podamos optar por una pila o una cola. Como se ha dicho, no es útil en general si se quiere acceder a los elementos de la estructura con coste constante, ya que el recorrido de una lista es de orden lineal.

Implementaciones

Lista enlazada Colección de registros que contienen el elemento de información y un apuntador al siguiente.

Vectores Un vector contiene los elementos de información. Se utiliza si podemos estimar el número máximo de elementos que albergamos.

Funciones de manipulación

Para la manipulación de una lista es preciso al menos implementar las siguientes funciones:

- Creación
  - **fun lista-vacía() dev l:lista** Devuelve una lista vacía.
- Modificación
  - **fun añadir(e:elemento, l:lista) dev l:lista** Añade el elemento e a la lista l
  - **fun resto(l:lista) dev l:lista** Elimina el primer elemento y devuelve el resto de la lista.
- Consulta
  - **fun vacía(l:lista) dev b:boolean** Comprueba si en la lista hay algún elemento.
  - **fun miembro(e:elemento, l:lista) dev b:booleano** Comprueba si el elemento e forma parte de la lista.
  - **fun elemento(p:posición, l:lista) dev e:elemento** Consulta el elemento de la posición p de la lista, sin modificarla
  - **fun primero(l:lista) dev e:elemento** Extrae el primer elemento de la lista, sin modificarla.

A efectos de calcular costes, los de estas operaciones se pueden estimar en:

	vectores	*punteros*
lista-vacía	cte	cte
añadir	cte	cte
resto	cte	cte
vacía	cte	cte
miembro	$\mathcal{O}(n)$	$\mathcal{O}(n)$
elemento	$\mathcal{O}(n)$	$\mathcal{O}(n)$
primero	cte	cte

Como ya hemos apuntado, una lista puede comportarse como pila o como cola. En todos los casos hay que hacer esta distinción y evitar en lo posible ambigüedades, por lo que recomendamos utilizar los términos de *pila* o de *cola* según corresponda.

1.5.5 Conjuntos

Definición

Un conjunto es una colección de elementos de información sin orden alguno.

Utilización

Se utilizan conjuntos cuando la pertenencia de un elemento de información dado es la información más relevante de la colección de elementos que estamos considerando.

No es aconsejable su uso en caso de necesitar organizar los elementos de información por orden o bien sea necesario referenciar a un elemento en una posición dada.

Implementaciones

Lista enlazada Colección de registros que contienen el elemento de información y un apuntador al siguiente.

Permite que el conjunto tenga un tamaño ilimitado, pero muchas operaciones tendrán un mayor coste debido al coste de recorrer la lista elemento a elemento.

Vectores Un vector contiene los elementos de información. Se utiliza si podemos estimar el número máximo de elementos que albergamos.

El tamaño del conjunto está limitado por el tamaño del vector. Tiene los mismos problemas de coste que la implementación mediante listas.

Vector Característico Para cada elemento existe un bit que indica si el elemento está o no en el conjunto.

Esta implementación limita también el tamaño del conjunto, pero permite que las operaciones sean mucho más eficientes debido a que se utilizan

operaciones lógicas como el AND para la intersección o el OR para la unión, entre otras. Es la que se usa en las implementaciones de conjuntos en lenguajes como PASCAL, MODULA-2, etc.

Funciones de manipulación

Hay muchas operaciones asociadas a conjuntos. Reseñamos aquí solo las más utilizadas en el texto:

- Creación
  - **fun conjunto-vacío() dev c:conjunto** Devuelve un conjunto vacío.
- Modificación
  - **fun añadir(e:elemento, c:conjunto) dev c:conjunto** Añade el elemento e al conjunto c.
  - **fun quitar(e:elemento, c:conjunto) dev c:conjunto** Elimina el elemento e del conjunto c.
- Consulta
  - **fun cardinalidad(c:conjunto) dev n:natural** Número de elementos del conjunto.
  - **fun pertenece(e:elemento, c:conjunto) dev b:booleano** Devuelve el resultado lógico de comprobar si e pertenece o no al conjunto c.
  - **fun igualdad(c_1,c_2:conjunto) dev b:booleano** Comprueba si dos conjuntos tienen los mismos elementos.
  - **fun subconjunto(c_1,c_2:conjunto) dev b:booleano** Comprueba si todos los elementos de c_1 pertenecen a c_2.
  - **fun superconjunto(c_1,c_2:conjunto) dev b:booleano** Comprueba si todos los elementos de c_2 pertenecen a c_1.

Otras operaciones interesantes son, además

- **fun intersección(c_1,c_2:conjunto) dev c:conjunto** Devuelve el conjunto formado por aquellos elementos que pertenecen simultáneamente a c_1 y a c_2.

- **fun unión(c_1,c_2:conjunto) dev c:conjunto** Devuelve el conjunto formado por aquellos elementos que pertenecen bien a c_1 o bien a c_2.
- **fun diferencia(c_1,c_2:conjunto) dev c:conjunto** Devuelve el conjunto formado por los elementos de c_1 que no pertenecen a c_2

Utilizaremos indistintamente estas funciones por su nombre o su equivalente en notación conjuntista. A efectos de calcular costes, los de estas funciones se pueden estimar en:

	vectores	*punteros*	*vector característico*
conjunto-vacío	cte	cte	cte
añadir	cte	cte	cte
quitar	$\mathcal{O}(n)$	$\mathcal{O}(n)$	cte
cardinalidad	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
pertenece	$\mathcal{O}(n)$	$\mathcal{O}(n)$	cte
subconjunto	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	cte
superconjunto	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	cte
intersección	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	cte
unión	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	cte
diferencia	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	cte

1.5.6 Grafos

Definición

Intuitivamente un grafo está compuesto por un conjunto de vértices unidos por aristas. Los vértices contienen elementos de información y éstos se relacionan entre sí mediante una arista, que puede a su vez tener un coste o *peso* asociado.

Las siguientes son algunos conceptos comunes sobre grafos:

camino Sucesión de vértices y aristas que comunican un vértice con otro.

peso Valor asociado a una arista. Indica el coste o el valor de uso de dicha arista.

ciclo Camino propio que empieza y termina en el mismo vértice.

conexo Un grafo es conexo cuando siempre hay al menos un camino entre cualesquiera dos vértices.

no dirigido Un grafo es no dirigido si la unión entre cualesquiera dos vértices adyacentes es simétrica.

Si un grafo es dirigido, las aristas se representan con una punta de flecha para indicar el orden del par de vértices que comunican.

A un grafo acíclico, conexo y no dirigido se le denomina *árbol*.

Utilización

Se utiliza para la representación de los elementos de información y de la relación entre ellos. Un ejemplo puede ser la representación de las ciudades (vértices) y las carreteras (aristas), así como la distancia (peso) que hay entre ellas.

Siempre que podamos utilizar un árbol binario o un montículo es preferible a la implementación de un grafo.

Implementaciones

Matriz de adyacencia Un registro contiene, por un lado, un vector con los elementos de información contenidos en los vértices, y, por otro lado, una matriz que puede tener valores lógicos indicando existencia o no de arista, o bien un valor que indique el peso o coste de dicha arista.

Si la matriz no es simétrica, el grafo es dirigido.

Listas de Adyacencia Una matriz de vértices contiene el valor de éstos y una lista de sus vértices sucesores.

Funciones de manipulación

- Creación
  - **fun grafo-vacío() dev g:grafo** Devuelve un grafo vacío.
- Modificación
  - **fun sucesores(v: vértice, g:grafo) dev l:lista** Devuelve una lista con los vértices adyacentes a v.
  - **fun peso(v_1,v_2: vértice, g:grafo) dev p:peso** Peso asociado a la arista que une los vértices dados.

  - **fun añadir-arista(v_1,v_2: vértice, p:peso, g:grafo) dev g:grafo** Añade una arista entre los vértices dados y le asigna el peso p.
  - **fun añadir-vértice(v: vértice, g:grafo) dev g:grafo** Añade el vértice v al grafo g.
  - **fun borrar-arista(v_1,v_2: vértice, g:grafo) dev g:grafo** Elimina la arista que une v_1 y v_2.
  - **fun borrar-vértice(v: vértice, g:grafo) dev g:grafo** Borra el vértice del grafo y todas las aristas que partan o lleguen a él.
- Consulta
  - **fun adyacente(v_1,v_2: vértice, g:grafo) dev b:boolean** Comprueba si los vértices v_1 y v_2 son adyacentes.

El coste asociado a las operaciones descritas puede consultarse en la tabla adjunta:

	matriz de adyacencia	*lista de adyacencia*
grafo-vacío	cte	cte
sucesores	$\mathcal{O}(n)$	cte
peso	cte	$\mathcal{O}(n)$
añadir-arista	cte	cte
borrar-arista	cte	$\mathcal{O}(n)$
añadir-vértice	cte	cte
borrar-vértice	$\mathcal{O}(n)$	$\mathcal{O}(n)$
adyacente	cte	$\mathcal{O}(n)$

1.5.7 Montículos

Definición

Es un tipo especial de árbol que cumple las siguientes características:

- Es binario (cada nodo tiene a lo sumo dos hijos).
- Es esencialmente completo (todo nodo interno, con la posible excepción de un nodo especial, tiene exactamente dos hijos).
- Cada uno de sus nodos incluye un elemento de información llamado *valor del nodo*, siendo éste mayor o igual que los valores de sus hijos.

Utilización

La principal utilidad de un montículo es la de estar permanentemente organizado de forma que nos proporcione de manera eficiente el elemento de mayor (o menor) valor, que es en este caso la raiz del árbol binario. A este estado se le denomina *propiedad de montículo* y debe restaurarse después de cualquier modificación del montículo.

Un ejemplo clásico de uso de montículos es el de la implementación de las colas de prioridad

Implementaciones

La representación se realiza siempre mediante un vector. Los nodos de profundidad k del árbol se colocan, leídos de izquierda a derecha, en las posiciones $T[2^k]$, $T[2^k+1]$,..., $T[2^{k+1}-1]$. El nivel 0, es decir, el nivel más inferior, no estará completo en general. Los hijos del nodo $T[i]$ son respectivamente $T[2i]$ y $T[2i+1]$. Utilizando este tipo de representación, la complejidad de las operaciones se simplifica enormemente.

Funciones de manipulación

Como hemos indicado las funciones de manipulación están encaminadas a que después de efectuar cualquier modificación debe restaurarse la propiedad de montículo.

- Creación
  - **fun montículo-vacío(v: vector) dev m:montículo** Crea un montículo con los datos contenidos en el vector v.
- Modificación
  - **fun hundir(i:indice, m:montículo) dev m:montículo** Restaura la propiedad de montículo perdida por ser $T[i]$ menor que la de alguno de sus hijos.
  - **fun flotar(i:índice, m:montículo) dev m:montículo** Restaura la propiedad de montículo perdida por ser $T[i]$ mayor que el valor del padre.

  - **fun añadir-nodo(e:elemento, m:montículo) dev m:montículo** Añade el valor e, y flota ese valor hasta restaurar la propiedad de montículo.
  - **fun borra-max(m:montículo) dev m:montículo** Elimina la raiz del montículo, es decir, el elemento de mayor valor, y restaura la propiedad de montículo.
- Consulta
  - **fun vacío(m:montículo) dev b:boolean** Comprueba si en el montículo hay algún elemento.
  - **fun max(m:montículo) dev e:elemento** Devuelve el elemento de mayor valor del montículo,es decir, el situado en la raiz.

Al contrario que en los casos anteriores, no existe más que una única implementación posible con los siguientes costes:

	coste
montículo-vacío	$\mathcal{O}(n)$
hundir	$\mathcal{O}(\log n)$
flotar	$\mathcal{O}(\log n)$
añadir-nodo	$\mathcal{O}(\log n)$
borra-max	$\mathcal{O}(\log n)$
vacío	cte
max	cte

$l_1 = 1$

$l_2 = 2$

$l_3 = 3$.

Posibles combinaciones.

a) - — — —

b) - — — —

c) — - — —

d) — — — -

e) — — - —

f) — — — -

tiempo medio de acceso.

ta) $l_1 = 1$ $l_2 = 1+3 = 4$ $l_4 = 6 \Rightarrow \frac{10}{3}$

tb) $l_1 = 1$ $l_2 = 1+2$ $l_3 = 6 \Rightarrow \frac{9}{3}$

tc) $l_1 = 2$ $l_2 = 3$ $l_3 = 6 \Rightarrow \frac{11}{3}$

td) $l_1 = 2$ $l_2 = 5$ $l_3 = 6 \Rightarrow \frac{13}{3}$

te) $l_1 = 3$ $l_2 = 4$ $l_3 = 6 \Rightarrow \frac{13}{3}$

tf) $l_1 = 3$ $l_2 = 5$ l_3 6 $= \frac{14}{3}$.

Primero el menor

Capítulo 2

Algoritmos voraces

2.1 Almacenamiento de programas en cinta

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

Elección razonada del esquema algorítmico.

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

de máquina

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

donde la constante k no afecta el resultado.

El tiempo medio de acceso es

→ Sumatorio de todos

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

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

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

Descripción del esquema usado e identificación con el problema.

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

En notación algorítmica:

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

En este caso, C es el conjunto de programas, y S no es exactamente un conjunto, sino una lista de programas, ya que nos interesa conservar la información sobre el orden en el que los programas fueron incorporados a la solución.

Las funciones sin definir del esquema son, en nuestro caso:

1. *solución(S)*: Se han incorporado todos los programas a S.
2. *objetivo(x)*: longitud del programa (en este caso ha de minimizarse, no maximizarse).

3. *completable*: Siempre es cierta.

Estructuras de datos.

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

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

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

Algoritmo completo a partir del refinamiento del esquema general.

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

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

La función *objetivo* devuelve la longitud del programa:

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

Estudio del coste.

El bucle principal se recorre exactamente n veces. Dentro de ese bucle existen tres operaciones: selección del candidato adecuado (coste $\mathcal{O}(n)$), eliminación del candidato (de nuevo, operación de coste $\mathcal{O}(n)$ si el conjunto está implementado como una lista) y adición del candidato seleccionado al conjunto solución (operación de coste constante). Tenemos, pues, dos factores n^2 que nos dan el coste global del algoritmo, $\mathcal{O}(n^2)$.

Es interesante estudiar cómo puede optimizarse el coste de un problema cambiando la implementación de las estructuras de datos. En este caso, podemos reducir el coste si implementamos el conjunto de candidatos mediante un montículo de mínimos (según la longitud de los programas). De esta forma, el programa más corto siempre estaría en la raíz, y por tanto el coste de seleccionar el candidato adecuado sería constante. Eliminarlo de C tendría un coste $\mathcal{O}(\log n)$, que es lo que se tarda en restaurar la propiedad de montículo cuando se extrae la raíz. Como el bucle principal se ejecuta n veces, ahora el coste de ejecutar ese bucle es $\mathcal{O}(n \log n)$. Al comienzo del algoritmo hay que considerar ahora una nueva instrucción para inicializar el conjunto de candidatos como un montículo de mínimos. Como esta tarea tiene un coste $\mathcal{O}(n \log n)$ (al igual que el bucle principal), el coste global del algoritmo es también $\mathcal{O}(n \log n)$.

Demostración de optimalidad

$$T = \frac{\sum_i (x_i + l_i)}{n} = \frac{\sum_i (\sum_{j=1}^{i-1} l_j) + l_i)}{n} = l_1 + \frac{n-1}{n} l_2 + \frac{n-2}{n} l_3 + \ldots + \frac{1}{n} l_n$$

Para desarrollar esa igualdad hemos tenido en cuenta que la distancia x_i no es más que la suma de las longitudes de los programas que están antes de p_i, es decir, $x_i = \sum_{j=1}^{i-1} l_j$.

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

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

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

2.2 Mezcla de cintas

Dado un conjunto de n cintas no vacías con n_i registros ordenados cada una, se pretende mezclarlas a pares hasta lograr una única cinta ordenada. la secuencia en la que se realiza la mezcla determinará la eficiencia del proceso. Diséñese un algoritmo que busque la solución óptima minimizando el número de movimientos.
Por ejemplo: 3 cintas: A con 30 registros, B con 20 y C con 10. mezclamos A con B (50 movimientos) y luego el resultado con C (60), con lo que realizamos en total 110 movimientos.

Elección razonada del esquema algorítmico.

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

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

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

Descripción del esquema usado e identificación con el problema.

Para el esquema voraz

```
fun voraz(C:conjunto) dev (S: conjunto)
S ← conjunto − vacio
mientras ¬ solución(S) ∧ C ≠ ∅ hacer
    x ← elemento que maximiza objetivo(x)
    C ← C \ {x}
```

```
    si completable( S ∪ {x})
     entonces  S ← S ∪ {x}
    fsi
fmientras
dev S
ffun
```

hemos de particularizar las siguientes funciones:

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

Estructuras de datos.

Se utilizarán vectores de n valores naturales para representar los conjuntos. Por ejemplo, para el conjunto C, se define la variable c como vector de naturales, siendo $c[i] = n_i$ la expresión de que la cinta i consta de n_i registros. El conjunto S puede representarse de forma análoga.

Para representar la ausencia de un elemento puede usarse cualquier marcador (por ejemplo el valor 0).

Algoritmo completo a partir del refinamiento del esquema general.

Retocando el esquema general tenemos:

```
fun voraz(c: vector ) dev (s: vector )
para i ← 0 hasta n hacer s[i] ← 0 fpara
i ← 0
mientras i ≤ n hacer
   x ← seleccionar(c)
   c[x] ← 0
   s[i] ← x
fmientras
dev s
ffun
```


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

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

Demostración de Optimalidad

Supongamos que el orden de mezcla de las cintas atiende a la sucesión $\{n_i\}_{i=1}^{n}$, siendo $\forall i \in \{1 \ldots n-1\} n_i \leq n_{i+1}$. El valor por minimizar es $m = \sum_{j=1}^{N}(\sum_{i=1}^{j} n_i)$, siendo N el número total de cintas por mezclar. Desarrollando queda $m = Nn_1 + (N-1)n_2 + \ldots + n_N$. Si intercambiamos cualesquiera dos elementos: $(N-j)n_{j+1}$ por $(N-i)n_{i+1}$, con $i < j$ resultará que por hipótesis $n_{j+1} \geq n_{i+1}$ y además $N - i > N - j$, con lo que necesariamente $(N-i)n_{j+1} + (N-j)n_{i+1} \geq (N-i)n_{i+1} + (N-j)n_{j+1}$, por lo que el orden propuesto es el que minimiza m.

Estudio del coste.

La función *objetivo* es $\mathcal{O}(n)$, y el bucle pricipal se repite n veces, por lo que el coste es $\mathcal{O}(n^2)$.

2.3 Tabla de distancias entre ciudades

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

Elección razonada del esquema algorítmico.

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

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

Descripción del esquema usado e identificación con el problema.

Para el esquema voraz

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


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

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

Estructuras de datos.

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

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

Algoritmo completo a partir del refinamiento del esquema general.

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

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

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

Estudio del coste.

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

2.4 Recubrimiento de vértices de un grafo

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

Elección razonada del esquema

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

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

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

Esquema general e identificación con el problema

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

En notación algorítmica puede escribirse así:

```
fun voraz(C:conjunto) dev (S: conjunto)
```

```
S ← ∅
mientras ¬ solución(S) ∧ C≠ ∅ hacer
   x ← elemento que maximiza objetivo(x)
   C ← C \ {x}
   si completable( S ∪ {x})
    entonces S ← S ∪ {x}
   fsi
dev S
ffun
```

Donde C será, en este caso, el conjunto de vértices del grafo y S será un recubrimiento del mismo. La forma más "intuitiva" de garantizar que el recubrimiento sea mínimo es tomar vértices de los que salgan muchas aristas, esto es, elegir vértices con el mayor grado.

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

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

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

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

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

Estructuras de Datos

En el problema intervienen grafos, vértices de un grafo y conjuntos de vértices. Para representar el grafo podemos utilizar cualquiera de las formas habituales, teniendo en cuenta que el uso de una matriz de adyacencia hará menos eficiente el algoritmo.

Aquí representaremos el grafo como un vector de vector de vértices, teniendo asociado cada vértice una lista enlazada con los vértices adyacentes a éste. Como los costes no juegan ningún papel en el algoritmo, los excluiremos (por comodidad) de la estructura de datos.

```
grafo = vector [1... N] de vertice

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

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

Algoritmo completo a partir del refinamiento del esquema

Adaptamos el algoritmo general a lo antedicho:

```
fun recubrimiento_minimo (G:grafo) dev (S: conjunto de vértices)
S ← ∅
mientras ¬ solucion(S) ∧ C ≠ ∅ hacer
   x ← elemento que maximiza seleccionar(x)
   C ← C \ {x}
   disminuir_grado (x,C)
   fsi
dev S
ffun
```

Las funciones son las siguientes:

solucion: El conjunto S será una solución cuando el grado de todos los elementos que restan en C sea cero:

```
fun solución (C:conjunto de vértices)  dev b: booleano
   b ← cierto
   para c en C
      si G[i].grado ≠ 0  hacer  b ← (G[i].grado = 0)
   fpara
   dev b
ffun
```

La función *seleccionar* devuelve el grado del vértice en consideración:

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

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

```
fun disminuir'grado (v:vertice, C)
para w ∈ C  en sucesores(v)  hacer
  w.grado ← w.grado − 1
fpara
ffun
```

Optimalidad

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

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

Análisis del coste

El tamaño del problema viene dado por el número de vértices del grafo. El número de veces que se ejecuta el bucle voraz es, en el peor de los casos, n. Dentro del bucle se realizan dos operaciones: encontrar el vértice de mayor grado, que tiene un coste lineal, y disminuir en uno el grado de los demás, que tiene también un coste lineal. De modo que el coste del bucle es $\mathcal{O}(n)$, y el coste del algoritmo es $\mathcal{O}(n^2)$.

2.5 Diseño de una red de carreteras

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

Elección razonada del esquema algorítmico.

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

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

¡Cuidado! No debe confundirse este problema con el de encontrar los caminos mínimos entre un vértice y el resto. Ver el problema 2.3: tabla de distancias entre ciudades.

Descripción del esquema usado e identificación con el problema.

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

En notación algorítmica puede escribirse así:

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


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

```
fun prim (g=(N,A): grafo) dev conjunto
S ← ∅
B ← un elemento arbitrario de N
mientras B ≠ N hacer
 buscar e={u,v} de longitud mínima tal que
  u ∈ B y v ∈ N \ B
 S ← S ∪ {e}
 B ← B ∪ {v}
dev S
ffun
```

Estructuras de datos

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

Algoritmo completo

El algoritmo de Prim no necesita ninguna modificación posterior.

Estudio del coste

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

Capítulo 3

Divide y vencerás

3.1 Cálculo de una función de Fibonacci generalizada

Dada la sucesión definida como $f_n = af_{n-3} + bf_{n-2} + cf_{n-1}$ se pide diseñar un algoritmo que calcule en tiempo logarítmico el término f_n. **Sugerencia:** Utilizad la siguiente relación:

$$\begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix} \begin{pmatrix} f_{n-3} \\ f_{n-2} \\ f_{n-1} \end{pmatrix} = \begin{pmatrix} f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix}$$

F Γ

Elección razonada del esquema algorítmico

Cada término de la sucesión se calcula en función de los tres anteriores. Por tanto, la forma trivial de calcular f_n consiste en calcular los n términos anteriores, lo que supone un coste lineal $\mathcal{O}(n)$. Sin embargo, la ecuación que se nos proporciona nos da una forma más eficiente de resolver el problema, recurriendo a la técnica divide y vencerás. Efectivamente, si llamamos

$$F \equiv \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ a & b & c \end{pmatrix}$$

tendremos

$$\begin{pmatrix} f_{n-2} \\ f_{n-1} \\ f_n \end{pmatrix} = F \begin{pmatrix} f_{n-3} \\ f_{n-2} \\ f_{n-1} \end{pmatrix} = F^2 \begin{pmatrix} f_{n-4} \\ f_{n-3} \\ f_{n-2} \end{pmatrix} = \ldots = F^{n-2} \begin{pmatrix} f_0 \\ f_1 \\ f_2 \end{pmatrix}$$

donde f_0, f_1 y f_2 no pueden calcularse en términos de los elementos anteriores y han de ser, por tanto, datos del problema.[1]

Para calcular f_n es suficiente, por tanto, con evaluar F^{n-2} y hacer una multiplicación de una matriz de 3×3 por un vector de 3 elementos (que tiene coste constante). Sabemos que la forma más eficiente de calcular exponenciaciones es mediante la técnica divide y vencerás, aplicada mediante la igualdad:

$$F^n = (F^{n \text{ div } 2})^2 F^{n \bmod 2}$$

(es decir, $F^n = (F^{n/2})^2$ si n es par, y $F^n = F(F^{(n-1)/2})^2$ si n es impar).

Al reducir un problema de tamaño n a otro de tamaño $n/2$, conseguiremos una eficiencia $\mathcal{O}(\log n)$ frente a $\mathcal{O}(n)$, como comprobaremos al final.

Descripción del esquema e identificación con el problema

El esquema *divide y vencerás* es una técnica recursiva que consiste en dividir un problema en varios subproblemas del mismo tipo. Las soluciones a estos subproblemas se combinan a continuación para dar la solución al problema original. Cuando los subproblemas son más pequeños que un umbral prefijado, se resuelven mediante un algoritmo específico. Si su tamaño es mayor, se vuelven a descomponer. El esquema es el siguiente:

```
fun divide_y_venceras (problema)
   si suficientemente-simple (problema)
   entonces  dev solucion-simple (problema)
   si no  hacer
     { p_1 ... p_k } ← descomposicion(problema)
     para cada p_i  hacer  s_i ← divide_y_venceras(p_i)  fpara
     dev combinacion(s_1 ... s_k)
```

[1] Por ejemplo, la sucesión de fibonacci es un caso particular de f con $a = b = 1$, $c = 0$ y $f_0 = 1, f_1 = 1$.


```
   fsi
ffun
```

Los elementos de este esquema se particularizan así a nuestro caso:

Tamaño umbral y solución simple: El preorden bien fundado para los problemas se deriva directamente del orden total entre los exponentes (n) como números naturales. Podemos tomar como tamaño umbral n=1, caso en que la exponenciación es trivial y se devuelve como resultado la matriz de entrada.

Descomposición: F^n se descompone en un único subproblema, $F^{n/2}$, de la mitad de tamaño. Se trata, pues, de un problema de reducción más que de descomposición. El convertir un problema de tamaño n en otro de tamaño $n/2$ es lo que nos proporcionará un algoritmo eficiente.

Combinación:La función de combinación es la ecuación mencionada más arriba.

Estructuras de datos

En el problema intervienen solamente enteros y matrices de 3×3. Los datos de entrada y salida del algoritmo son enteros; las matrices se utilizan como resultados intermedios. Daremos por conocida la multiplicación de matrices.

Algoritmo completo

Suponemos conocidos a, b, c, f_0, f_1, f_2.

```
fun f (n:entero)  dev entero
  F ← | 0 1 0 |
      | 0 0 1 |
      | a b c |
  caso  n = 0 :  dev  f_0 ;
     []  n = 1 :  dev  f_1 ;
     []  n = 2 :  dev  f_2 ;
     []  verdadero :  hacer
                 S ← exp_mat(F, n − 2)
                 s ← S (f_0, f_1, f_2)^T
                 dev  s[3]
  fcaso
```

```
fun exp_mat (M:vector[3,3]; n:entero)  dev vector[3,3]
  si n ≤ 1
  entonces  dev solucion_simple(M,n)
  si no  hacer
  p ← n div 2
  r ← n mod 2
  T ← exp_mat(M,p)
    dev combinacion(T,r,M)
  fsi
ffun
```

En los argumentos de *combinacion*, sólo el primero es un subproblema. Los otros dos son parámetros auxiliares que utiliza la función de combinación.

```
fun combinacion (T:vector[3,3]; r:entero; M:vector[3,3])  dev vector[3,3]
  dev TT * M^r
ffun
```

donde $M^1 = M, M^0 = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$.

```
fun solucion_simple (M:vector[3,3], n:entero)  dev vector[3,3]
  si n = 1  dev M
  si no  dev ( 1 0 0 )
             ( 0 1 0 )
             ( 0 0 1 )
  fsi
ffun
```

Estudio del coste

El coste del algoritmo recae sobre la exponenciación de la matriz F. Como se reduce un problema de tamaño n a otro de tamaño $n/2$, el coste cumple la ecuación de recurrencia $T(n) = T(n/2) + cte$, donde cte hace referencia al tiempo necesario para hacer una o dos multiplicaciones de matrices de 3×3. Utilizando

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

$$\Rightarrow$$

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

con $k = 0, b = 2, a = 1$, estamos en el caso $a = b^k$ y, por lo tanto, el coste del algoritmo es $\Theta(\log n)$. Una implementación directa del cálculo de la sucesión hubiera necesitado un tiempo lineal con $\mathcal{O}(n)$.

3.2 Distribución de alumnos en un aula

En una clase hay f filas y c columnas de pupitres. Delante de la primera fila se encuentra la pizarra.

- a: Diseñar un algoritmo que reparta los $f * c$ alumnos de forma que, al mirar hacia la pizarra, ninguno se vea estorbado por otro alumno más alto que él.
- b: A mitad de curso se coloca una pizarra adicional en una de las paredes adyacentes con la primera pizarra. Diseñar un algoritmo que coloque a los alumnos de forma que puedan mirar también a esa segunda pizarra sin estorbarse. Este algoritmo debe sacar partido de la colocación anterior.

Elección razonada del esquema algorítmico.

Para que nadie tenga un alumno mas alto que él al mirar a la pizarra, es necesario que, dentro de cada columna, los alumnos estén ordenados según su altura de mayor a menor (ver figura 3.1). Para resolver el apartado a) de forma eficiente basta con dividir los $f * c$ alumnos en c subconjuntos de f elementos escogidos al azar, y a continuación debe ordenarse cada uno de esos subconjuntos. Cada uno de ellos será una columna en la clase. Como algoritmo de ordenación puede escogerse cualquiera de los estudiados; nosotros utilizaremos el algoritmo divide y vencerás de *fusión*, por ser más eficiente asintóticamente en el caso peor (es $\mathcal{O}(n \log n)$).

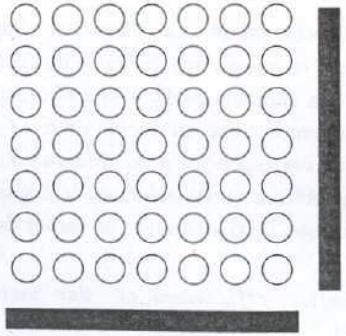


Figura 3.1: Aula con dos pizarras adyacentes.

Al colocar una segunda pizarra adyacente a la primera, los alumnos de cada fila deben estar, a su vez, ordenados entre sí. Para que estén ordenadas las columnas y las filas, es necesario ordenar a todos los alumnos de menor a mayor, y colocarlos en la clase de forma que el más bajito ocupe el pupitre que está en la intersección de las dos pizarras, y el más alto en el vértice opuesto de la clase. Por lo tanto, para obtener la disposición final de los alumnos en el apartado b) debe hacerse una ordenación de $f * c$ elementos. Pero si aprovechamos la disposición anterior no es necesario, esta vez, aplicar ningún algoritmo de ordenación: basta con realizar una fusión de los c subconjuntos ya ordenados (equivaldría al último paso de un algoritmo de ordenación por fusión en el que el factor de división fuera c). Así, la ordenación final puede obtenerse en tiempo lineal.

Descripción del esquema usado e identificación con el problema.

Dado el esquema divide y vencerás:

```
fun divide-y-venceras (problema)
   si suficientemente-simple (problema)
   entonces   dev solucion-simple (problema)
   si no   hacer
      { p_1 ... p_k } ← descomposicion(problema)
      para cada p_i hacer s_i ← divide-y-venceras(p_i) fpara
      dev combinacion(s_1 ... s_k)
   fsi
ffun
```

se particulariza para nuestro problema de la siguiente manera: *Tamaño umbral y solución simple*: El preorden bien fundado para los problemas se deriva directamente del orden total entre el tamaño de los subconjuntos problema. Podemos tomar como tamaño umbral n=1, caso en el que la ordenación es trivial y consiste simplemente en devolver el elemento.

Descomposición: Dividiremos el conjunto problema en dos subconjuntos formados por los n div 2 primeros elementos por un lado y el resto por otro.

Combinación: Es necesario fundir los dos subconjuntos ordenados, mediante un bucle que toma cada vez el menor elemento de entre los primeros de uno y otro subconjunto todavía sin seleccionar.

Estructuras de datos.

La única estructura de datos que necesitamos es una matriz de enteros de tamaño $c * f$ que almacene las alturas de los alumnos. También podemos utilizar un vector de tamaño $c * f$, sabiendo que cada f elementos representan una columna, aunque es menos natural.

Algoritmo completo a partir del refinamiento del esquema general.

Llamaremos a a la función que obtiene la ordenación requerida en el apartado a), y b a la que soluciona el apartado b), es decir, obtiene una ordenación total a partir de la que se tiene en a).

La función a es la siguiente:

```
fun a (clase: vector[1... f,1... c] de reales)
                    dev vector[1... f,1... c] de enteros
 para i ← 1 hasta c hacer
  clase[1... f,i] ← ordenacion(clase[1... f,i])
 fpara
 dev clase
ffun
```

La función de ordenación equivale a la función general divide y vencerás, que modificaremos ligeramente para incluir como argumento vectores que sean siempre del mismo tamaño:

```
proc ordenacion (v: vector[1... n];i,j: entero )
   si i ≠ j entonces hacer
     k ← (i + j div 2) ;
     ordenacion (v,i,k);
     ordenacion (v,k+1,j);
     fusion (v,i,j,2)
   fsi
fproc
```

La función de fusión tiene cuatro parámetros: el vector, el principio y final del tramo que contiene a los dos subvectores que hay que fusionar, y el factor

de división empleado. Para el apartado a) podríamos ahorrarnos este último argumento (es 2), pero esa generalización nos permitirá usar la misma función en el apartado b).

```
proc fusion (v: vector[1... n];inicio,final,factor: entero )
{ Inicializa el vector solución }
v' ← vector[1... n]
{ Inicializa punteros al comienzo de los vectores por fusionar }
  para k ← 1 hasta factor hacer
  i_k ← inicio + [illegible]
  fpara
I ← { i_1 ... i_factor }
{ Selecciona el menor elemento de entre los principios de vector
  para incluirlo en la solucion, y a continuacion lo borra para
  que no vuelva a ser considerado }
  para h ← inicio hasta final hacer
 i_x ← elemento que maximiza v[i_x]
 v'[h] ← v[i_x]
  si i_x < inicio + factor * x − 1
    hacer i_x ← i_x + 1
    si no hacer I ← I \ { i_x }
  fsi
 fpara
v ← v'
fproc
```

La función b debe conseguir una ordenación completa del conjunto de alumnos, pero debe tener en cuenta que ya existe una ordenación parcial entre ellos: los alumnos de cada columna están ordenados entre sí. Si representamos el conjunto de los alumnos como un vector de $c * f$ elementos, en el que los f primeros elementos corresponden a la primera columna, los f siguientes a la segunda, etc, el problema queda solucionado llamando a la función de fusión definida anteriormente, pero utilizando un factor de división c en lugar de 2:

```
proc b (v: vector[1... c*f], entero: c) dev vector[1... c*f]
fusion (v,1,c*f,c)
fproc
```

Estudio del coste.

apartado a

La ordenación por fusión tiene un coste que cumple:

$$T(n) = 2T(n/2) + cte * n$$

ya que el algoritmo de fusión tiene un coste $\mathcal{O}(n)$ (consta de dos bucles consecutivos). De esa igualdad se obtiene un coste $\mathcal{O}(n \log n)$. Como se realizan c ordenaciones de f elementos cada una, el coste total es $\mathcal{O}(c(f \log f))$. Mediante una ordenación total habríamos resuelto también el problema, pero con un coste $\mathcal{O}(n \log n) \rightarrow \mathcal{O}(cf \log cf)$ (ya que el tamaño del problema sería $c * f$).

apartado b

Se resuelve mediante una llamada al algoritmo de fusión, que tiene un coste lineal; como el tamaño del problema es $c * f$, el coste es $\mathcal{O}(c * f)$. El coste es mucho menor que en el caso en que no aprovecharamos la ordenación parcial que se obtiene en el apartado a.

3.3 Cálculo de una función exponencial

El tiempo de ejecución de un algoritmo viene dado por $T(n) = 2^{n^2}$. Encontrar una forma eficiente de calcular $T(n)$, suponiendo que el coste de multiplicar dos enteros es proporcional a su tamaño en representación binaria.

Elección razonada del esquema algorítmico.

Una forma eficiente de calcular exponenciaciones es aplicando la técnica divide y vencerás y usando la relación

$$a^n = (a^{n \text{ div } 2})^2 a^{n \bmod 2}$$

(es decir, $a^n = (a^{n/2})^2$ si n es par, y $a^n = a(a^{(n-1)/2})^2$ si n es impar).

Descripción del esquema usado e identificación con el problema.

El esquema general

```
fun divide_y_venceras (problema)
  si suficientemente-simple (problema)
  entonces  dev solucion-simple (problema)
  si no  hacer
    { p_1 ... p_k } ← descomposicion(problema)
    para cada p_i hacer s_i ← divide_y_venceras(p_i) fpara
    dev combinacion(s_1 ... s_k)
  fsi
ffun
```

se particulariza a este problema de la siguiente manera:

Tamaño umbral y *solución-simple*: El preorden bien fundado para los problemas se deriva directamente del orden total entre los exponentes (n) como números naturales. Podemos tomar como tamaño umbral n=1, caso en que la exponenciación es trivial y se devuelve como resultado el mismo dato de entrada.

Descomposición: 2^{n^2} se descompone en un único subproblema, $2^{n^2/2}$, de la mitad de tamaño. Se trata, pues, de un problema de reducción más que de descomposición. El convertir un problema de tamaño n en otro de tamaño $n/2$ es lo que nos proporcionará un algoritmo eficiente.

Combinación:La función de combinación viene dada por la fórmula anterior para $a = 2$:

$$2^{n^2} = (2^{n^2} \text{ div } 2)^2 2^{n^2 mod\ 2}$$

Estructuras de datos.

Las únicas estructuras de datos necesarias en el problema son los enteros. Según las condiciones del problema, supondremos que se dispone de un algoritmo de multiplicación de enteros de coste proporcional al tamaño de éstos en representación binaria. En particular, 2^n se representa en binario como un 1 seguido de n ceros, de forma que el coste de la operación $(2^n)^2$ es proporcional a $n+1$.

Algoritmo completo a partir del refinamiento del esquema general.

```
fun T (n:entero)  dev entero
 m ← n * n
  dev exp(2,m)
ffun
```

```
fun exp (a:entero; n:entero)  dev entero
  si n ≤ 1
  entonces  dev solucion_simple(a,n)
  si no  hacer
   p ← n div 2
   r ← n mod 2
   t ← exp(a, p)
    dev combinacion(t,r,a)
  fsi
ffun
```

En los argumentos de *combinacion*, sólo el primero es un subproblema. Los otros dos son parámetros auxiliares que utiliza la función de combinación.


```
fun combinacion (t:entero; r:entero; a:entero)  dev entero
  dev t * t * a^r
ffun
```

```
fun solucion_simple (a:entero, n:entero)  dev entero
  si n = 1  entonces  dev a
  si no  dev 1
  fsi
ffun
```

Estudio del coste.

El coste de calcular una exponenciación según este algoritmo cumple la siguiente relación:

$$C(n) = C(n/2) + cte * n$$

Es decir, calcular a^n se reduce a calcular a^n div 2 y multiplicar dos o tres enteros cuyo tamaño es del orden de n. Las operaciones n div 2 y $n mod\ 2$ tienen un coste constante sobre números en representación binaria.

Utilizando

$$T(n) = \begin{cases} cn^k & \text{, si } 1 \le n < b \\ aT(n/b) + cn^k & \text{, si } n \ge b \end{cases}$$

$$\Rightarrow$$

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

para $b = 2, a = 1, k = 1$ estamos en el caso $a < b^k$. Por lo tanto, el orden de complejidad de nuestro algoritmo de exponenciación es $\Theta(n)$. Como la cantidad que necesitamos calcular es $T(n) = 2^{n^2}$, esa operación tendrá un coste $\Theta(n^2)$. Si nos hubiéramos limitado a realizar n^2 multiplicaciones, en lugar de usar la técnica divide y vencerás, el orden de complejidad hubiera sido $O(n^3)$.

3.4 Las torres de Hanoi

Se tienen 3 varillas y n discos agujereados por el centro. Los discos son todos de diferente tamaño y en la posición inicial se tienen los discos insertados en el primer palo y ordenados en tamaños decrecientes desde la base hasta la punta de la varilla . El problema consiste en pasar los discos de la 1^a a la 3^a varilla observando las siguientes reglas: Se mueven los discos de uno en uno y nunca un disco puede colocarse encima de otro menor que éste.

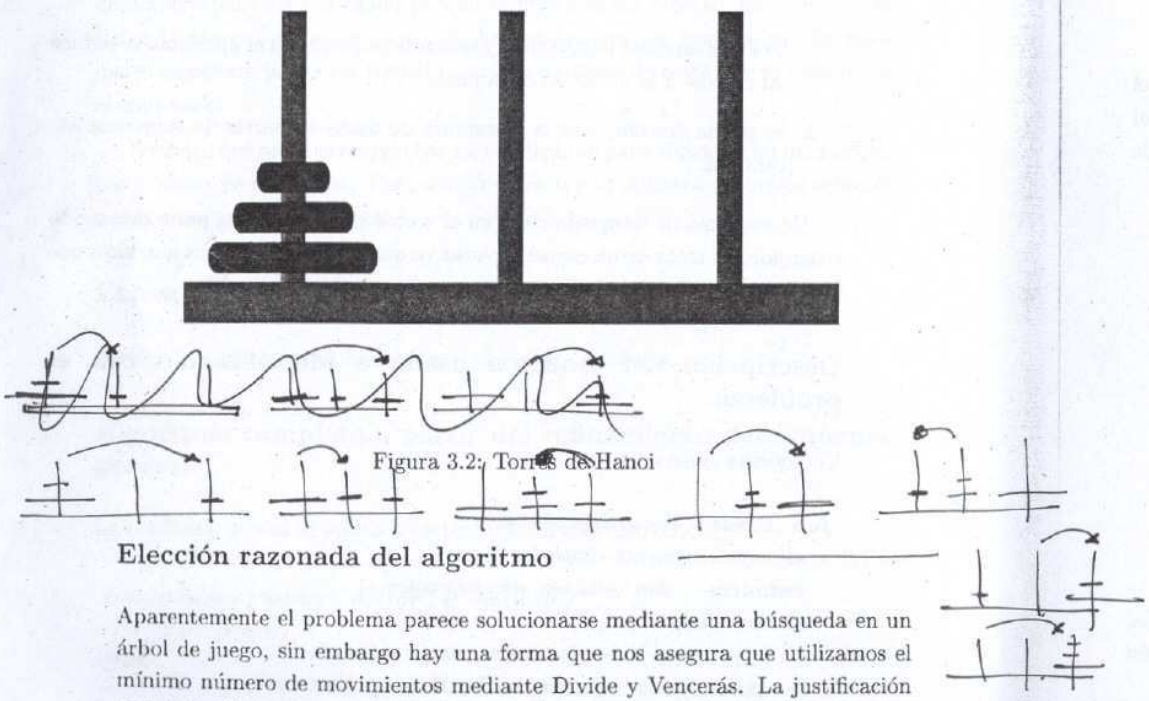


Figura 3.2: Torres de Hanoi

Elección razonada del algoritmo

Aparentemente el problema parece solucionarse mediante una búsqueda en un árbol de juego, sin embargo hay una forma que nos asegura que utilizamos el mínimo número de movimientos mediante Divide y Vencerás. La justificación de la elección se basa por tanto en la eficiencia más que en la imposibilidad de utilizar el esquema de Vuelta Atrás. Por otra parte hay que indicar que un esquema voraz es imposible de aplicar, ya que la solución no consta de unos elementos elegidos de un conjunto de candidatos, sino de los movimientos efectuados sobre ellos. Para que un conjunto de candidatos de tal naturaleza fuera posible, habría que añadir a cada movimiento información acerca del número de discos que descansan sobre cada varilla, lo cual es tanto como enumerar todas

las posibles combinaciones del juego.

Descripción del esquema usado e identificación con el problema

El esquema general es:

```
fun Divide y Vencerás(problema)
  si suficientemente-simple(problema)
  entonces  dev solución-simple(problema)
  si no  hacer
    { p_1 ... p_k } ← descomposicion(problema)
    para cada p_i  hacer  s_i ← Divide y Venceras (p_i)
    fpara
    dev combinar( s_1 ... s_k)
  fsi
ffun
```

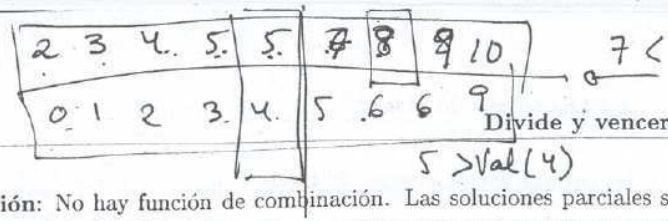
y se particulariza de la siguiente forma:

Tamaño umbral y solución simple: El preorden se establece sobre los tamaños de los sucesivos problemas en la cadena de llamadas recursivas. El único problema verdaderamente simple es el de solucionar el pasatiempo para un único disco, por lo que el tamaño umbral es $n = 1$.

Descomposición: Cualquier solución para pasar n discos del poste 1 al 3 pasa por:

1. Conseguir pasar $n-1$ discos del poste 1 al poste 2
2. Pasar el disco que queda -que además será el de mayor radio de los que había en el poste 1- del poste 1 al poste 3
3. Pasar por último los $n-1$ discos que habíamos puesto en el poste auxiliar 2, de dicho poste al poste 3.

En general, si tenemos k discos en i y queremos pasarlos a j, siendo $j, i \in \{1,2,3\}$, habrá que calcular $6-i-j$ para hallar el número del poste restante que se utilizará como auxiliar y se siguen los pasos anteriores.



Combinación: No hay función de combinación. Las soluciones parciales son válidas en sí mismas. En el caso que nos ocupa éstas son movimientos expresados mediante *origen* y *destino*.

Estructuras de Datos

No hay estructuras complejas de datos. Se utilizarán únicamente tipos de datos estándar.

Algoritmo completo a partir del refinamiento del esquema general

Los parámetros del algoritmo serán la varilla origen, la varilla destino y el número de discos que deben ser movidos del origen al destino. La llamada inicial tendrá como origen la varilla 1, como destino la varilla 3 y con un número n de discos.

El algoritmo refinado se expone a continuación:

```
fun Hanoi(origen,destino,n)
  si n=1
  entonces  escribe "Mover disco de poste origen a poste destino"
  si no  hacer
    Hanoi(origen,6-destino-origen,n-1)
    Hanoi(origen,destino,1)
    Hanoi(6-origen-destino,destino,n-1)
  fsi
ffun
```

La función de combinación no es necesaria ya que cada descomposición del problema en un subproblema produce un movimiento y dos llamadas a la función

Estudio del Coste

Debemos plantear la ecuación de recurrencia. En el caso que nos ocupa, al entrar a la función tenemos una instrucción **si ... entonces** con coste 1 seguida de tres llamadas a la función, de manera que $T(n) = 2T(n-1) + T(1) + 1$. Simplificando $T(1)$ tenemos finalmente que $T(n) = 2T(n-1) + 2$ y resolviendo se obtiene que el orden es complejidad es $\mathcal{O}(2^n)$.

3.5 Elemento de un vector igual a su índice

> Se tiene un vector de enteros no repetidos y ordenados de menor a mayor. Diseñar un algoritmo que compruebe en tiempo logarítmico si existe algún elemento del vector que coincida con su índice.

Elección razonada del esquema algorítmico.

Se trata del esquema de Divide y Vencerás por dos motivos:

1. Se exige un coste logarítmico y esto sólo es posible si el problema se reduce -al menos- a la mitad en cada paso.
2. Se puede descomponer la estructura de datos en partes de su misma naturaleza.

Un esquema de búsqueda ciega en el vector carece por otra parte de sentido y tampoco se trata de un esquema voraz ya que no hay *candidatos* que haya que escoger de ningún tipo de conjunto.

Descripción del esquema usado e identificación con el problema.

El esquema general es:

```
fun Divide y Vencerás(problema)
  si suficientemente-simple(problema)
  entonces  dev solución-simple(problema)
  si no  hacer
    { p1 ... pk } ← descomposicion(problema)
    para cada pi  hacer  si ← Divide y Venceras (pi)
    fpara
    dev combinar( s1 ... sk)
  fsi
ffun
```

Se trata de un problema en el que no hay que efectuar una combinación posterior de las soluciones. A este tipo de problemas se los denomina de *reducción*

o *simplificación*, dentro del esquema de Divide y Vencerás.

Lo más importante en este tipo de problemas de simplificación es que hay que tener en cuenta con qué mitad del vector nos quedamos para seguir buscando. Este criterio es prácticamente lo único que tenemos que decidir, ya que el algoritmo debe solo devolver *falso* o dar un índice del vector.

Supongamos que es i el elemento central del vector v, o bien i, j los elementos centrales en caso de que tenga un número par de elementos. Si $v[i] < i$ sabemos que podemos descartar la parte izquierda del vector como susceptible de contener en ella algun elemento tal que $v[i] = i$. Lo mismo le ocurre a la derecha para la condición contraria. Esto ocurre por dos motivos: primero porque se trata de números enteros y segundo porque se prohiben las repeticiones. Se deja al lector la comprobación de que estas dos condiciones son imprescindibles para que el algoritmo pueda ser resuelto con los requisitos de coste que se indican en el enunciado.

Lo único que resta es comprobar las condiciones para vectores con un número par e impar de elementos. Para dividir el vector se utilizará un índice sobre él que se pasa como argumento.

Estructuras de datos.

No hay más estructuras de datos que el vector de enteros.

Algoritmo completo a partir del refinamiento del esquema general.

La condición trivial se aplica a vectores de un solo elemento:

```
fun solucion-trivial(v: vector ; i: natural )
    dev (v[i]=i)
ffun
```

El esquema refinado es:

```
fun indice(v: vector ; i,j: natural )
  si i=j entonces solucion-trivial(v,i)
  si no hacer
   k ← (i+j) div 2
   si v[k]≥k entonces indice(v,i,k)
   si no indice(v,k+1,j)
ffun
```


Estudio del coste.

La ecuación de recurrencia del algoritmo es $T(n) = T(n/2) + 1$. Su resolución es, como esperábamos, $T(n) \in \mathcal{O}(logn)$

Capítulo 4

Algoritmos de exploración en grafos

4.1 Generar palabras con restricciones

Dado el conjunto de caracteres alfabéticos, se quieren generar todas las palabras de cuatro letras que cumplan las siguientes condiciones: *a)* La primera letra debe ser vocal. *b)* Sólo pueden aparecer dos vocales seguidas si son diferentes. *c)* No puede haber ni tres vocales ni tres consonantes seguidas. *d)* Existe un conjunto C de parejas de consonantes que no pueden aparecer seguidas.

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.

Descripción del esquema usado e identificación con el problema.

El esquema de backtracking o vuelta atrás es una técnica algorítmica para realizar búsquedas en grafos ó árboles. Si en algún punto del recorrido se dan condiciones que hagan inútil seguir el camino tomado, se retrocede a la última decisión y se sigue la siguiente opción. El esquema general es el siguiente:

```
fun vuelta-atrás (ensayo)
   si válido (ensayo)
   entonces   dev ensayo
   si no   para hijo ∈ compleciones(ensayo)
           hacer   si condiciones de poda(hijo)
                   entonces vuelta-atrás(hijo)
   fsi
ffun
```

En el caso que nos ocupa, un *ensayo* es una palabra de 4 letras. Un ensayo 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). 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 tres 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 cuatro niveles.

A continuación se describen las estructuras de datos necesarias y se refina el algoritmo anterior.

Estructuras de datos

Para representar una palabra 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*.

La tupla se puede definir así:


```
ensayo =  tupla
  palabra:  arreglo [1..4]  de  a..z
  indice:   entero
  ftupla
```

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
- No 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 palabra
 lista-compleciones ← lista-vacía
  para  cada letra  en  {a,b,c,..,z}  hacer
  hijo ← añadir-letra (e,letra);
  lista-compleciones ← añadir(lista-compleciones,hijo)
  fpara
  dev lista-compleciones
ffun
```

Donde la función *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(nuevo-ensayo.palabra,l);
  nuevo-ensayo.indice ← nuevo-ensayo.indice +1
  dev nuevo-ensayo
ffun
```

Con respecto a la función *condiciones de poda*, ésta queda como sigue:

```
fun condiciones de poda (e:ensayo) dev bool
  dev condicion1(e) ∧ condicion2(e) ∧
    condicion3(e)∧ condicion4(e)
ffun
```

La primera condición dice que la primera letra ha de ser una vocal:

```
fun condicion1 (e:ensayo) dev bool
  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 bool
  dev pertenece(l,{a,e,i,o,u})
ffun

fun consonante (l:letra) dev bool
  dev ¬ vocal (l)
ffun
```

La segunda condición dice que sólo pueden aparecer dos vocales seguidas si son diferentes. Podemos implementarla reescribiendola así: la segunda condición se cumple si la palabra tiene menos de dos letras, o si la última no es vocal, o si la última no es igual a la penúltima. Sólo comprobamos sobre las dos últimas porque el resto de la palabra ha debido pasar ya las condiciones de poda en el momento de ser generado.


```
fun condicion2 (e:ensayo) dev bool
  dev e.indice < 2 ∨
  consonante(ensayo.palabra[e.indice]) ∨
  e.palabra[e.indice] ≠ e.palabra[e.indice-1]
ffun
```

La tercera condición consiste en que no haya tres 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 reescribirse así: la condición se cumple si la palabra tiene menos de tres letras, o bien la última y la penúltima son de distinto tipo, o bien la última y la antepenúltima son de distinto tipo:

```
fun condicion3 (e:ensayo) dev bool
 i ← e.indice;
  dev i < 3 ∨
    vocal(e.palabra[i-1]) ≠ vocal(e.palabra[i]) ∨
    vocal(e.palabra[i-2]) ≠ vocal(e.palabra[i])
ffun
```

Por último, la última condición nos dice que no debe aparecer ninguna pareja de entre las de una lista negra C. De nuevo hay que comprobarlo sólo para las dos últimas letras:

```
fun condicion4 (e:ensayo) dev bool
 i ← e.indice;
  dev ¬pertenece((e.palabra[i-1],e.palabra[i]),C)
ffun
```

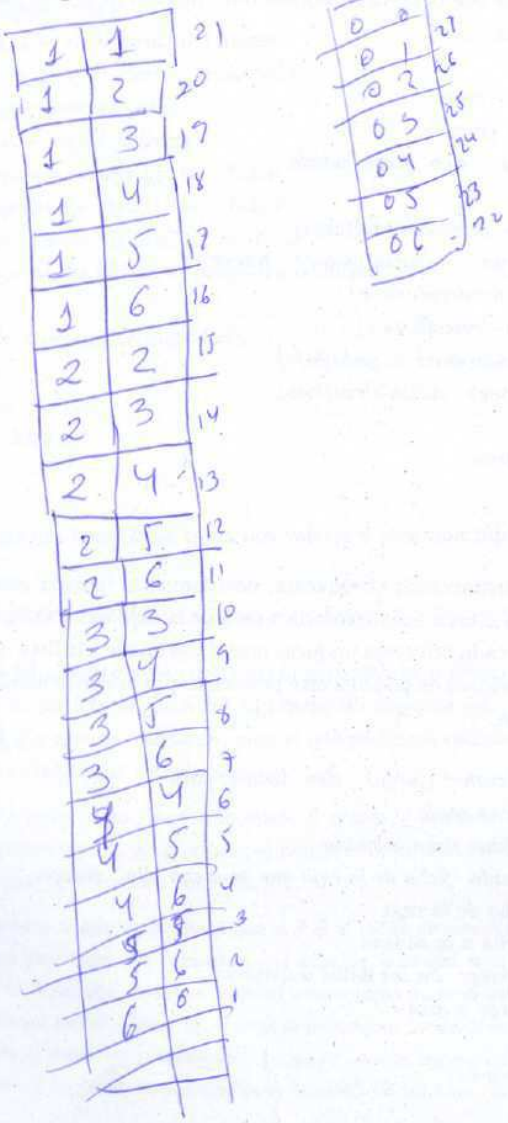
Sólo nos resta escribir la función principal, que debe llamar a *vuelta-atrás* tomando como argumento *ensayo-vacío*:

```
fun caracteres dev lista de ensayo
  dev vuelta-atrás (ensayo-vacío)
ffun
```

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 estará 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 cuatro condiciones que teníamos. Este factor exponencial quiere decir que el problema es irresoluble, en la práctica, para valores grandes de n.



4.2 Cadena de fichas del Dominó

Las 28 fichas de dominó son de la forma (i, j) con $i, j = 0 \ldots 6$. 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 la $(2, 4)$. Diséñese un algoritmo que produzca todas las cadenas permisibles que contengan todas las fichas del dominó.

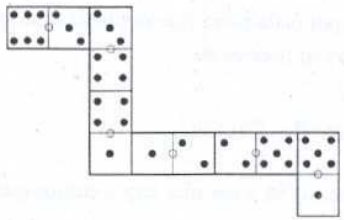


Figura 4.1: Comienzo de una partida de dominó

Elección razonada del esquema algorítmico

Se justifica la técnica de algoritmo por retroceso o *backtracking* por que:

- No hay un 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 descomponible en subproblemas idénticos cuyas soluciones puedan combinarse para dar la solución global.

Descripción del esquema usado e identificación con el problema

El esquema general de los algoritmos de retroceso es:

```
fun vuelta-atrás (ensayo)
  si válido (ensayo)
```

```
    entonces   dev ensayo
    si no   para hijo ∈ compleciones(ensayo)
            hacer   si condiciones de poda(hijo)
                    entonces vuelta-atrás(hijo)
    fsi
ffun
```

En nuestro caso:

válido: Cuando se hayan colocado todas las piezas. **compleciones**: se crea un nuevo ensayo por cada pieza que se pueda añadir en ese momento. **condiciones de poda**: No son necesarias.

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: Una matriz booleana simétrica. Si una ficha está en la caja, en la posición (i,j) y (j,i) habrá un valor *cierto*. La ficha (i, j) está en la caja si $caja[i, j] = cierto$
- Cadena de fichas: Un vector con rango 1..28 que contenga fichas. Una ficha será un registro con dos valores numéricos i y j con rango $0 \ldots 6$.

La estructura queda:

```
juego =   tupla
    caja:   arreglo [0..6, 0..6] de   bool
    cadena:   arreglo [1..28] de ficha
    ultima:   entero
ftupla
```


Refinamiento del Algoritmo

Una vez detalladas las estructuras de datos, el algoritmo se puede refinar instanciando el esquema al problema planteado.

Suponemos que la función *compleciones* nos devuelve una lista. El esquema refinado queda:

```
fun domino (juego)
    si válido (juego)
    entonces   dev juego.cadena
    si no
    lista-c ← compleciones(juego)
      mientras ¬ vacia(lista-c)   hacer
      hijo ← primero(lista-c)
      lista-c ← resto(lista-c)
        si condiciones de poda(hijo)
        entonces vuelta-atrás(hijo)
        fsi
      fmientras
    fsi
ffun
```

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.

```
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
 ultima-ficha ← juego.cadena[última]
 j ← ultima-ficha.j
  para i ← 0  hasta 6  hacer
   si caja[j,i] = cierto  entonces
   juego-nuevo ← juego
   ficha ← crear-ficha(j,i)
   juego-nuevo.caja[i,j] ←  falso
   juego-nuevo.caja[j,i] ←  falso
   juego-nuevo.cadena[j+1] ← ficha
   juego-nuevo.ultima ← juego-nuevo.ultima + 1

   lista-c ← añadir(juego,lista-c)
   fsi
  fpara
  dev lista-c
ffun
```

La función *crear-ficha* toma dos valores y crea una tupla del tipo indicado.

Análisis 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 que este es constante, pero sí que podemos estimar el coste analizando cómo es el árbol que recorremos.

Como cada nodo tiene a lo sumo 6 ramas y además el último nodo tiene solo una alternativa, al igual que el penúltimo, de manera que el coste se puede estimar muy por encima como de 6^{26}.

En general se puede considerar que si β es el *índice de ramificación* del árbol y γ es el número de elementos que componen una solución, o lo que es lo mismo, la profundidad del árbol de búsqueda, vamos en realidad a tener como máximo que realizar β^γ pasos para alcanzar todas las soluciones. Si además se introducen condiciones de poda, éstas influyen reduciendo el índice de ramificación. La poda además dependerá de la profundidad p y del número n de piezas disponibles para completar la solución. El *coeficiente de poda* es

un valor entre 0 y 1, si el coeficiente de poda es cero, no existen condiciones de poda y la búsqueda es totalmente ciega, si es 1 no habrá ramificación. La expresión general sería $[\beta(1-\xi(n,p))]^\gamma$. En este problema $\beta = 6$, $\gamma = 28$ y $\xi(n,p)$ es una función que varía entre 0 y $\beta - 1/\beta$.

4.3 Recorrido del Caballo de Ajedrez

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.

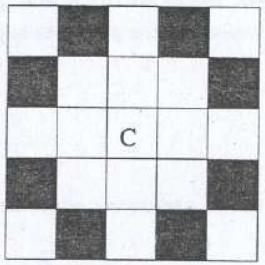


Figura 4.2: Las casillas grises son las alcanzables por un movimiento de caballo desde la casilla C

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 menor tamaño sin que se desvirtúe el mismo.

Descripción del esquema usado e identificación con el problema

El esquema de Vuelta Atrás utiliza el retroceso como técnica de exploración en 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 válido (ensayo)
   entonces   dev ensayo
   si no   para hijo ∈ compleciones(ensayo)
           hacer   si condiciones de poda(hijo)
                   entonces vuelta-atrás(hijo)
   fsi
ffun
```

La particularización del esquema al problema del caballo puede hacerse así:

válido: Un ensayo será válido 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.

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.

- 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.

Algoritmo completo a partir del refinamiento del esquema general

Como hemos indicado antes, un *ensayo* consta de tres 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(ensayo)
 lista ← lista vacía;
   si condiciones de poda (ensayo)
   entonces   hacer
     para cada rama   hacer
      w ← generar ensayo (rama)
      lista← insertar(lista,w);
     fpara
   fsi
   dev lista
ffun
```

La única condición de retroceso posible es la de 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 en 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 puesto que deben ser válidas) y devolver una lista vacía para que la función retroceda.

NOTA: Al hablar de condiciones de poda nos referimos a que *anticipemos* que no hay solución por ese camino, no que hayamos llegado a un punto mu erto. En este sentido la condición de "Caballo sin casillas libres a su alrededor" no es un a condición de poda ya que no es un criterio objetivo que anticipe un camino erróneo. En el caso de este problema no hay en realidad condiciones de poda sino únicamente condici ones de retroceso y por tanto esta función no tiene sentido en esta implementación.

Los movimientos posibles del caballo pueden verse en la figura 4.2. 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 válidas se pu ede hacer mediante un bucle con la condición anterior para formar los ocho nuevos movimientos posibles.

Para crear los nuevos tableros a partir de un movimie ntos válido se utiliza la función de generar ensayo. Esta función crea una nue va variable ensayo y genera:

- Un nuevo registro de última casilla ocupada con la p osición obtenida del bucle.

- 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 efectuado.

- Incrementamos en 1 el valor del número de movimi entos efectuados que contenía el anterior ensayo.

```
fun compleciones(ensayo)
  lista ← lista-vacía;
  (i, j)  ← ensayo.ultima-posición;
  N ← ensayo.numero-movs;
  ultimo-tablero ← ensayo.tablero;
 /* Generamos todas las posibles ramas */
  para i ← -2  hasta 2  hacer
  para j ← -2  hasta 2  hacer
    si   ( abs(i) + abs(j) = 3) ∧
         ( ensayo.tablero[i, j] = 0)
    entonces   hacer
         nuevo-tablero ← ultimo-tablero;
```

```
            nuevo-tablero[i,j] ← N+1;
            nueva-posición ← (i,j);
            nuevo-ensayo ← < nuevo-tablero,nueva-posición,N+1 >

            lista ← añadir(lista,nuevo-ensayo);
        fsi
    fpara
    fpara
    dev lista
ffun
```

Por último, un tablero es válido si hemos realizado N^2 movimientos, con lo que la función *válido* queda como sigue:

```
fun válido (ensayo)
    dev (ensayo.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 (ensayo)
    si válido (ensayo )
    entonces escribe ensayo
    si no hacer
        lista ← compleciones(ensayo)
        mientras no vacía(lista) hacer
            w ← PrimerElemento(lista)
            saltocaballo(w)
            lista ← Resto(lista)
        fmientras
    fsi
ffun
```

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 de 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.

4.4 Cuadrado Mágico de suma 15

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 direcciones, incluidas las diagonales. Diseñar un algoritmo que busque todas las soluciones posibles.

2	7	6
9	5	1
4	3	8

Figura 4.3: Una solución para el problema del cuadrado mágico

Elección razonada del esquema algorítmico.

El pasatiempo requiere tomar números y colocarlos de forma que encajen en la cuadrícula de manera que cumplan las condiciones del enunciado. En la figura 4.4 puede verse una solución. Aunque hay claramente un conjunto de elementos candidatos, no hay sin embargo forma de prever que podamos cometer un error en la elección y por tanto vernos obligados a deshacer una decisión tomada. El esquema correcto es el de Vuelta Atrás, ya que por otra parte no es posible descomponerlo en subproblemas de su misma naturaleza.

Descripción del esquema usado e identificación con el problema.

El esquema general de Vuelta Atrás es al siguiente:

```
fun vuelta-atrás (ensayo)
  si válido (ensayo)
  entonces   dev ensayo
  si no   para hijo ∈ compleciones(ensayo)
          hacer   si condiciones de poda(hijo)
                  entonces vuelta-atrás(hijo)
  fsi
ffun
```


Condiciones de Poda

La búsqueda de una combinación correcta de números en las casillas no es totalmente ciega. Tenemos varios criterios con los que podar el árbol que recorre el algoritmo.

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 s tal que $s < 6$ o bien $s > 14$, ya que en estos casos ninguna combinación posterior encontrará solución.

Otras condiciones pueden limitar la presencia en la casilla central de determinados números, p.ej. 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 podemos estar descartando soluciones válidas. En este problema no se impondrán condiciones al número ubicado en la casilla central, pero se deja como ejercicio la implementación de esta condición de poda y su demostración.

Compleciones

Un posible esquema principal para la función compleciones es el siguiente:

```
fun compleciones(ensayo)
 lista ← lista-vacía;
  si condiciones de poda (ensayo)
  entonces   hacer
    para cada rama válida   hacer
     w ← generar ensayo (rama)
     lista ← insertar(lista,w);
    fpara
  fsi
  dev lista
ffun
```

La función *Condiciones de Poda* devuelve un valor cierto si un determinado nodo (rama) es susceptible de llegar a alcanzar una solución válida.

Como hemos expresado anteriormente vamos a evitar explorar ramas que:

- Contengan 2 elementos y sumen una cantidad c donde $c > 14$
- Contengan 2 elementos y sumen una cantidad c donde $c < 6$

El resto de las combinaciones vamos a considerarlas válidas para seguir buscando.

NOTA: Como hemos expresado antes, Es importante establecer criterios de poda que nos ayuden lo más posible a reducir el tamaño del árbol de búsqueda, pero no es útil apoyarse en razonamientos intuitivos, salvo que estos sean demostrables.

Para comprobar las condiciones expuestas creamos unas funciones auxiliares que verifiquen las sumas en horizontal, en vertical y en 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 )
  s ← 0;
  para i ← 1 hasta 3 hacer
    s ← s + matriz[fila,i];
  fpara
  dev s
```

Para las columnas se utiliza una función análoga. Asimismo se construyen otras dos funciones que compruebe el número de elementos distintos de cero en una fila y en una columna dada:

```
fun num-f(matriz: vector [1..3,1..3] de natural ; fila: natural )
fun num-c(matriz: vector [1..3,1..3] de natural ; columna: natural )
```

Estructuras 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 por tanto de:

- Una matriz de 3 × 3 de números naturales. Ej. *cuadricula*
- Un conjunto de números disponibles . ~~Ej. candidatos~~
- Un número natural n que indique cuántos valores hay colocados en la cuadrícula. Ej. *num-ocupadas*

Se define *TipoCuadrado* como un registro de lo anterior.

Algoritmo completo a partir del refinamiento del esquema general.

La función *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.

La función compleciones debe generar una lista de compleciones que tenga en cuenta lo anterior. El esquema general un poco más refinado es:

```
fun compleciones(cuadrado)
 lista ← lista'vacía;
   para i ← 1 hasta 9 hacer
     si i en cuadrado.disponibles entonces
       w ← generar-ensayo(cuadrado,i);
       lista ← insertar(lista,w);
     fsi
   fpara
  dev lista
ffun
```

Faltan por especificar las funciones de *condición 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 válidas, es decir, cualquier

combinación de una cuadrícula en la que hayamos puesto ya 3 números y en la que éstos no sumen 15 es descartada.

La función *generar-ensayo* se limita a tomar la información necesaria y generar la estructura de datos resultante de integrarla. Con un cuadrícula, una posición dentro de ella y el valor numérico que debe insertar, forma un nuevo nodo

```
fun generar-ensayo(cuadrado: tipoCuadrado, i,j: natural ;n: natural )

   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
```

La implementación de las funciones de poda queda como sigue:

```
fun condiciones de poda(cuadrado: tipoCuadrado) dev bool
  para f ← 1 hasta 3 hacer
     si suma-fila(cuadrado.cuadricula,f) < 6 ∧ entonces dev Falso
     si num-f(cuadrado.cuadricula,f) < 3 ∧
      suma-fila(cuadrado.cuadricula,f) > 14 entonces dev Falso
  fpara
  para c ← 1 hasta 3 hacer
     si suma-columna(cuadrado.cuadricula,c) < 6 entonces dev Falso
     si num-c(cuadrado.cuadricula,c) < 3 ∧
       suma-columna(cuadrado.cuadricula,f) > 14 entonces dev Falso
  fpara
  dev Cierto
ffun
```

Y la implementación final de *compleciones* queda:

```
fun compleciones(cuadrado)
 lista ← lista vacía;
```



```
 si condiciones de poda(cuadrado) entonces
   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-valido(cuadrado.cuadrícula)
       entonces
        w ← generar-ensayo(cuadrado,i,j,n);
        lista ← insertar(lista,w);
      fsi
     fpara
    fpara
   fpara
  fsi
ffun
```

Como hemos indicado antes, la función *parcialmente-válido* comprueba que todas aquellas filas, columnas o diagonales con 3 valores sumen 15. Su implementación se deja al lector como ejercicio.

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 de 9^2. Para el siguiente número hay 8^2 y así sucesivamente. De nuevo tenemos con esto una cota irreal; primero porque estamos descartando un enorme número de ramas que no cumplen con las condiciones exigidas de suma 15, y segundo porque en realidad no estamos teniendo en cuenta la simetría de la cuadrícula.

4.5 Paso de un número a otro mediante dos operaciones

Se consideran las funciones $m(x) = 3x$ y $d(x) = x$ div 2 (donde ' div ' representa la división entera). Diseñar un algoritmo que, dados dos números a y b, encuentre una forma de llegar de a a b mediante aplicaciones sucesivas de $m(x)$ y $d(x)$. Por ejemplo, se puede pasar de 7 a 2 mediante

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

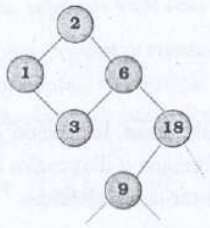


Figura 4.4: Vista parcial del grafo de búsqueda

Elección razonada del esquema algorítmico.

El esquema voraz no es aplicable, ya que no se trata de un problema de optimización. El esquema 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 número de origen a, y deberemos explorar el árbol 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.

En la figura 4.5 puede verse el aspecto del árbol de búsqueda.

Descripción del esquema usado e identificación con el problema.

El esquema de vuelta atrás es:

```
fun vuelta-atrás (ensayo)
   si válido (ensayo)
   entonces   dev ensayo
   si no   para hijo ∈ compleciones(ensayo)
           hacer   si condiciones de poda(hijo)
                   entonces vuelta-atrás(hijo)
   fsi
ffun
```

En nuestro caso, cada ensayo estará constituido, fundamentalmente, por una serie de aplicaciones de $m(x)$ y $d(x)$. Habrá dos compleciones posibles: aplicar m o aplicar d. Para no caer en ciclos, debemos anotar en una lista los nodos ya visitados, y desechar una exploración que nos lleve a alguno de ellos de nuevo. Esta variable debe ser global. También debemos tener cuidado de eliminar una rama si se llega al cero.

En principio, hay que evitar el uso de variables globales, ya que afectan a la transparencia de un programa. En este caso, si usa-ramos un parámetro que acumulara los nodos visitados, repetiríamos búsquedas hechas en otras ramas, ya que el árbol está lleno de bucles y ramas interconectadas. Por eso aceptaremos el uso de una variable global que acumule los nodos visitados.

Además, hay que evitar las búsquedas infinitas no cíclicas. Todas las ramas que consisten en multiplicar indefinidamente son infinitas; para evitarlas, basta con situar el ensayo que consiste en dividir en el primer lugar de la lista de compleciones para cada nodo.

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 una estructura con los siguientes registros: dos enteros que representen el número en que nos encontramos y el número objetivo, y una lista de caracteres con las funciones que hemos aplicado hasta llegar a ese ensayo.

```
ensayo =  tupla
  origen:  entero
  destino:  entero
  camino:  lista de caracter
```

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* y *compleciones*):

```
fun válido (e:ensayo)
  si e.origen = e.destino
   entonces  dev cierto
   si no  dev falso
  fsi
ffun

fun compleciones (e:ensayo)
 lista-compleciones ← lista-vacía

 hijo1 ← e;
 hijo1.origen ← e.origen * 3;
 hijo1.camino ← añadir("m", hijo1.camino);
 añadir(hijo1,lista-compleciones);
 añadir(hijo1.origen,nodos-visitados)

 hijo2 ← e;
 hijo2.origen ← e.origen div 2;
 hijo2.camino ← añadir("d", hijo2.camino);
 añadir(hijo2,lista-compleciones)
 añadir(hijo2.origen,nodos-visitados)

   dev lista-compleciones

ffun
```



```
fun condiciones-de-poda (ensayo)
  si ensayo.origen en nodos-visitados ∨ ensayo.origen = 0
   entonces  dev falso
   si no  dev cierto
  fsi
ffun
```

Necesitamos definir una función adicional que inicialice la variable global *nodos-visitados* y el primer ensayo antes de llamar a la función recursiva *vuelta-atrás*:

```
fun conecta (a,b:  natural )
 e ← inicializar-ensayo;
 e.origen ← a;
 e.destino ← b;
 e.camino ← lista-vacia;
 nodos-visitados ← lista-vacia;
 vuelta-atrás(e)
```

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; obviamos su implementación.

Estudio del coste.

No es posible deducir de forma sencilla cuál es el coste de este algoritmo, ya que el árbol de análisis es potencialmente infinito. De hecho, ¡ni siquiera es fácil definir cual es el tamaño del problema ! Dos números pequeños, pero próximos entre sí, pueden ser más difíciles de conectar que dos números muy grandes. De hecho, este algoritmo halla una solución, pero no la más corta, de forma que sus resultados podrían ser arbitrariamente largos.

4.6 Reparto de tareas entre mensajeros

Una empresa de mensajería dispone de tres motoristas en distintos puntos de la ciudad, y tiene que atender a tres clientes en otros tres 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	20

Diseñar un algoritmo que distribuya un cliente para cada motorista de forma que se minimice el coste total (en tiempo) de atender a los tres clientes.

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 decisiones; por tanto, hemos de desestimar el esquema voraz en 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 del 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 al mínimo de cada una de las tareas no asignadas (30, 20 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 $40 + 20 + 20 = 80$ minutos. En lugar de explorar el árbol en anchura o en profundidad, examinaremos en cada momento el nodo más prometedor, es decir, aquel cuya cota inferior sea mínima. 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 ya obtenida, el algoritmo habrá terminado, puesto que ya no será posible encontrar soluciones mejores.

Descripción del esquema usado e identificación con el problema.

El esquema de ramificación y poda es una variante de búsqueda ciega en un árbol, en la que se busca una solución que sea óptima de acuerdo a un criterio dado (para concretar, que minimice 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 será 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, en general, el número de nodos que es necesario visitar.

El algoritmo puede esquematizarse así:

```
fun ramificación-y-poda (ensayo) dev ensayo
 m ← montículo-vacío;
 cota-superior ← cota-superior-inicial;
 solución ← solución-vacía;
 añadir-elemento(ensayo,m);
  mientras ¬vacío(m) hacer
  nodo ← extraer-raiz(m)
   si válido(nodo) entonces hacer
    si coste(nodo) < cota-superior entonces hacer
     solución ← nodo;
     cota-superior ← coste(nodo)
    fsi
   si no
    si cota-inferior(nodo) ≥ cota-superior entonces
     dev solución
    si no
     para cada hijo en compleciones(nodo) hacer
      si condiciones-de-poda(hijo) y cota-inferior(hijo) < cota-superior
       entonces añadir-nodo(hijo,m)
      fsi
     fpara
    fsi
```

```
  fsi
 fmientras
ffun
```

En nuestro caso, *cota-inferior(nodo)* devolverá la suma de los costes de las tareas asignadas y los costes mínimos de las no asignadas. La *cota-superior* será, en cada momento, el coste de la mejor solución encontrada. La *cota-superior-inicial* puede ser una solución cualquiera: por ejemplo, la diagonal principal $30+20+20=70$. La función *compleciones* puede funcionar tomando la primera tarea sin asignar y generando tantos nodos como mensajeros disponibles hay para esa tarea. Respecto a las *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 sería, por ejemplo, que no haya dos mensajeros efectuando la misma tarea).

Estructuras de datos.

En principio, 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 esa posición con la tarea que se ha asignado. Sin embargo, optamos por una estructura más rica que nos evite repetir cálculos (coste de la asignación parcial, cota inferior, etc.). Es la siguiente:

```
ensayo = tupla
 asignaciones : vector [3] de natural
 tareas-no-asignadas : lista de natural
 ultimo-mensajero-asignado : natural
 coste : natural
```

Algoritmo completo a partir del refinamiento del esquema general.

Las funciones que hemos de especificar son:

```
fun compleciones (e:ensayo) dev lista de ensayo
matriz-de-costes ← ( 30 40 70 )
                   ( 60 20 10 ) ;
                   ( 40 90 20 )
lista-compleciones ← lista-vacía;
para cada tarea en e.tareas-no-asignadas hacer
  complecion ← e;
  mensajero ← e.ultimo-mensajero-asignado + 1;
  complecion.asignaciones[mensajero] ← tarea;
  complecion.coste ← e.coste + matriz-de-costes[tarea,mensajero];
  complecion.tareas-no-asignadas ← eliminar(tarea,e.tareas-no-asignadas);
  complecion.ultimo-mensajero-asignado ← mensajero;
  lista-compleciones ← añadir(complecion,lista-compleciones)
fpara
dev lista-compleciones

fun cota-inferior (e:ensayo) dev natural
vector-de-mínimos ← [30,20,20];
 dev e.coste + ∑_{i=e.ultimo-asignacion+1}^{3} vector-de-mínimos[i]
ffun

fun válido (e:ensayo) dev bool
 dev e.ultimo-mensajero-asignado = 3
ffun
```

Adaptando el esquema general para incluir los datos constantes del problema, el algoritmo queda como sigue:

```
fun tareas-y-mensajeros (e:ensayo) dev ensayo
 m ← montículo-vacío;
 cota-superior ← 70;
 añadir-elemento(ensayo,m);
  mientras ¬vacío(m) hacer
  nodo ← extraer-raiz(m)
   si válido(nodo) entonces hacer
    si nodo.coste < cota-superior entonces hacer
     solución ← nodo;
     cota-superior ← coste(nodo)
    fsi
   si no
    si cota-inferior(nodo) ≥ cota-superior entonces
       dev solución
    si no
```

```
        para cada hijo en compleciones(nodo) hacer
         si cota-inferior(hijo) < cota-superior
          entonces añadir-elemento(hijo,m)
         fsi
        fpara
       fsi
      fsi
     fmientras
    ffun
```

Estudio 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 $\mathcal{O}(n!)$, ya que cada nodo de nivel k puede expandirse con los $n-k$ mensajeros que restan sin asignar.