

Problemas de Algoritmia Avanzada

Curso 2004–2005

25 de noviembre de 2004

Advertencia: todos los problemas descritos en este documento admiten diversas formas de ser resueltos, por lo que no debe suponerse que, necesariamente, otras formas de resolver el problema son incorrectas. En caso de duda, consúltese con los profesores de la asignatura.

Índice

1. Programación dinámica	2
2. Ramificación y poda	7
3. Simulación	8
. Soluciones de los ejercicios	10

1. Programación dinámica

EJERCICIO 1. Una empresa constructora desea subcontratar para las distintas tareas que hay que ejecutar (cada una de las cuales puede considerarse como una etapa del trabajo) a otras empresas. El coste de cada tarea depende de qué decisión se tomó en la etapa anterior, de forma que las tareas se pueden representar como un grafo multietapa en el que los nodos se disponen en niveles (etapas) y las aristas van siempre de los nodos de una etapa a los nodos de la siguiente, tal y como se refleja en el ejemplo de la figura 1. Cada arista (x, y) tiene además un coste asociado $d(x, y)$. Escribe un algoritmo de programación dinámica para encontrar la secuencia óptima de decisiones.

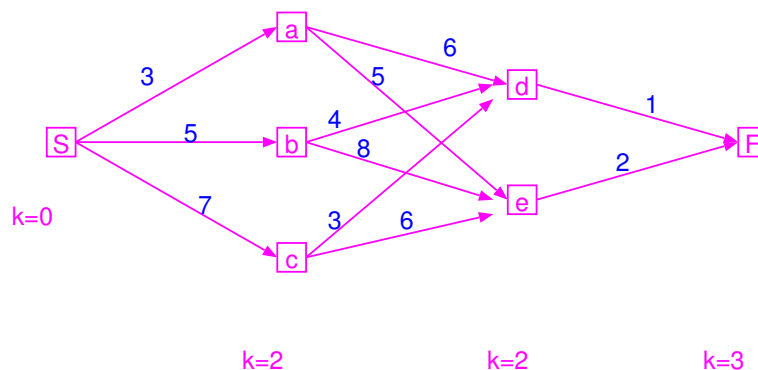


Figura 1: Un ejemplo de grafo multietapa.

EJERCICIO 2. Dada una secuencia finita de números enteros estrictamente positivos $S = \{a_1, a_2, \dots, a_N\}$, se desea extraer una subsecuencia tan larga como sea posible que sea no decreciente. Por ejemplo, si

$$S = \{7, 2, 6, 5, 6\}$$

las subsecuencias $\{2, 5, 6\}$ y $\{2, 6, 6\}$ son no decrecientes y de longitud máxima. Escribe un algoritmo recursivo y otro de programación dinámica iterativa para resolver este problema. Dibuja el árbol de llamadas que genera la función recursiva para los datos del ejemplo y marca aquellos nodos que son calculados si se usa un almacén para evitar repeticiones. Compara este número con el número de cálculos que hace el algoritmo iterativo.

EJERCICIO 3. Un robot con dos brazos debe soldar una serie de puntos r_1, \dots, r_N en un circuito. Aunque el orden de soldadura no puede cambiarse, puede elegirse qué brazo realizará la soldadura. Se quiere encontrar la

secuencia de movimientos que minimiza la longitud total de los desplazamientos realizados si inicialmente ambos brazos se encuentran en la posición r_0 y las distancias son conocidas.

d	0	1	2	3
1	100	-	100	63
2	89	100	-	60
3	45	63	60	-
4	61	108	36	50

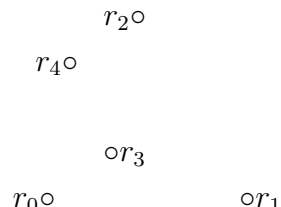


Figura 2: Un caso de circuito de soldadura.

- Comprueba que la solución voraz no es correcta para el ejemplo de la figura 2.
- Escribe una función de *programación dinámica recursiva* que permita resolver el problema y analiza su coste temporal en función de N .
- Dibuja el árbol de llamadas recursivas que se genera para los datos del ejemplo y escribe el resultado en cada nodo.
- Compara el número de cálculos con el de un esquema recursivo puro y con el de un esquema iterativo.

EJERCICIO 4. Escribe un programa `monedas(M)` que calcule el número mínimo de monedas necesarias para conseguir exactamente la cantidad M . Para ello, debemos utilizar un máximo de c_i monedas de valor v_i . Si no es posible obtener la cantidad M , el programa escribirá ∞ .

denominación	v_1	v_2	\dots	v_N
disponible	c_1	c_2	\dots	c_N

EJERCICIO 5. El personal de la universidad está organizado jerárquicamente como un árbol cuya raíz ocupa el rector. Para el vino de honor de fin de curso se quiere invitar al máximo posible de empleados de forma que no coincida ningún empleado con su superior inmediato. Escribe un algoritmo de programación dinámica para calcular el número máximo $f(x, a)$ de subordinados de la persona x a los que se puede invitar si x acude al ágape ($a = 1$) o si no acude ($a = 0$). Para ello, conocemos el conjunto $s(x)$ de subordinados inmediatos de x (nodos del árbol cuyo padre es x).

EJERCICIO 6. Al volver de Polinesia, un explorador llevaba una maleta de capacidad M completamente llena con objetos de gran valor. Cada objeto proviene de una isla diferente y los sistemas de transporte permiten viajar entre ellas según un grafo multietapa como el de la figura 1, en el que cada nodo q contiene el peso $w(q)$ del objeto situado en la isla. Escribe un algoritmo lo más eficiente posible para encontrar una ruta que permita recoger K objetos (uno por etapa) con peso total M .

EJERCICIO 7. Un problema que aparece con frecuencia en las redes de ordenadores es el de encontrar la ruta que permite transmitir los datos de un ordenador S a otro F a mayor velocidad. Las conexiones entre máquinas permiten una velocidad máxima de transmisión que supondremos idéntica en ambos sentidos. La velocidad de transmisión a través de una ruta que utiliza varias conexiones está determinada en la práctica por la conexión más lenta (véase la figura 3). Escribe un algoritmo recursivo para resolver el problema adecuado para el uso de programación dinámica.

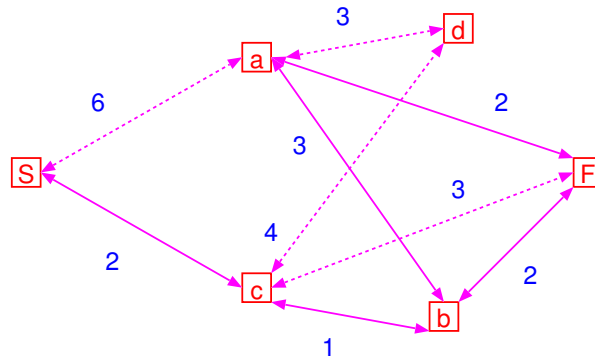


Figura 3: Ejemplo de grafo de conexiones. Un camino de velocidad máxima entre S y F aparece marcado como línea de puntos. Las conexiones no dibujadas equivalen a líneas de velocidad cero.

EJERCICIO 8. Una empresa de alquiler de vehículos debe renovar su flota de forma que ningún automóvil en activo tenga más de $M = 10$ meses y para planificar las compras dispone de la previsión del precio de los automóviles para los próximos $T = 100$ meses:

$$w_1, w_2, \dots, w_T$$

Los precios pueden subir o bajar cada mes y el objetivo es mantener los vehículos hasta el mes T (y medio) gastando el menor dinero posible en adquisiciones. Escribe un algoritmo recursivo y justifica que el coste temporal es

exponencial en función de T . Transfórmalo mediante programación dinámica recursiva y justifica el nuevo coste en función de T y M .

EJERCICIO 9. Algunas normas tipográficas establecen que las líneas deben contener entre 40 y 60 caracteres. Supongamos que no queremos partir las palabras y que conocemos los tamaños (incluidos los signos de puntuación y espacios en blanco adyacentes) s_1, s_2, \dots, s_N de las N palabras de un párrafo de M líneas. Si el objetivo es minimizar la longitud de la línea más larga del párrafo, escribe un algoritmo de programación dinámica recursiva que minimice la longitud de la línea más larga de un párrafo. A continuación, dibuja una tabla $A[N][M]$ con los valores que calcula el algoritmo anterior si la secuencia de tamaños es 20, 15, 12, 12, 24, 18, 17, 14, 10 y el número de líneas $M = 3$. ¿Cuál es la solución óptima? Transforma el algoritmo anterior en uno iterativo y compara la eficiencia de uno y otro.

EJERCICIO 10. Un estudiante se ha matriculado de tres asignaturas. Su familia le ha exigido que, para irse de vacaciones, apruebe al menos una de ellas. Le quedan cuatro días para estudiar y no le da buen resultado estudiar más de una asignatura diferente el mismo día.

La tabla siguiente nos da la probabilidad $p_k(t)$ de suspender la asignatura k si se dedican t días de estudio a esa asignatura. ¿Cuántos días debe dedicar a cada asignatura?

t(días)	AA	TBO	TV
0	0.90	0.61	0.72
1	0.80	0.55	0.64
2	0.70	0.52	0.58
3	0.60	0.46	0.54
4	0.50	0.41	0.52

EJERCICIO 11. El algoritmo de Cocke, Younger y Kasami (CYK) permite determinar si una frase es correcta sintácticamente y, en ese caso, cuál es su árbol de análisis sintáctico, según una gramática escrita en forma de Chomsky. Una gramática $G = (V, T, R, S)$ en forma de Chomsky consiste en:

- un conjunto finito $V = \{A_1, A_2, \dots, A_{|V|}\}$ de símbolos variables;
- un alfabeto $T = \{a_1, a_2, \dots, a_{|T|}\}$ de símbolos terminales que forman las frases;
- un símbolo inicial $S \in V$;
- un conjunto finito R de reglas formado por

1. reglas de sustitución de una variable por un terminal del tipo $A_i \rightarrow a_k$, y
2. reglas de sustitución de una variable por dos variables como $A_i \rightarrow A_j A_k$.

Dada la gramática G , escribiremos $\lambda_2(i, j) = \{k : A_k \rightarrow A_i A_j \in R\}$ y $\lambda_1(a) = \{k : A_k \rightarrow a \in R\}$. Justifica que el algoritmo CYK puede deducirse de la aplicación de programación dinámica a un programa recursivo simple.

EJERCICIO 12. Para seguir una dieta adecuada queremos elegir una combinación de platos de un menú de forma que el precio total no supere nuestro presupuesto M , el valor proteico sea como mínimo W y el valor calórico lo más bajo posible. Para cada plato i conocemos su precio p_i , su contenido en proteínas q_i y calorías c_i . Diseña un algoritmo para encontrar la selección óptima.

EJERCICIO 13. Tenemos N programas de tamaños w_1, w_2, \dots, w_N y de un número ilimitado de discos de almacenamiento de M kilobytes (por ejemplo, $M = 1400$). Si los programas no pueden fragmentarse en discos distintos, ¿cuál es el número mínimo de discos necesario para guardar todos los programas?

EJERCICIO 14. Una empresa de suministro de agua dispone de N cubas de capacidades c_1, c_2, \dots, c_N y debe transportar M litros de agua. Para disminuir el riesgo de volcado de los camiones, las cubas deben ir lo más llenas posible. Si las cubas sólo pueden usarse una vez:

1. Escribe un algoritmo de *programación dinámica recursiva* que permita calcular $f(n, m)$, el mínimo espacio sobrante cuando se deben transportar m litros usando sólo cubas comprendidas entre la 1 y la n .
2. Dibuja el árbol de llamadas recursivas para el siguiente ejemplo: $N = 4$, $M = 100$, $c_1 = 55$, $c_2 = 40$, $c_3 = 35$, $c_4 = 15$. Escribe el valor de n, m y $f(n, m)$ en cada nodo.
3. Compara el coste de este algoritmo con uno recursivo puro y con una versión iterativa del mismo.

EJERCICIO 15.

Un problema que se plantea frecuentemente en la secuenciación de DNA es encontrar fragmentos comunes en varias cadenas. Las cadenas de DNA se pueden representar como secuencias de nucleótidos (adenina, timina, citosina y guanina) del tipo GATTACACAGATA... Escribe un algoritmo de programación dinámica recursiva para encontrar la subsecuencia común más larga a

tres cadenas x , y y z de este tipo. Justifica cuál es el coste temporal del algoritmo en función de los tamaños $|x|$, $|y|$ y $|z|$.

EJERCICIO 16. Los modelos modernos de teléfonos móviles permiten escribir mensajes de texto apretando una sola tecla por letra y resolviendo las ambigüedades automáticamente (en la medida de lo posible). Por ejemplo, si se aprieta la secuencia 6382, su interpretación más probable es la palabra META.

Diseña un programa usando técnicas de programación dinámica para encontrar la interpretación más probable de una secuencia de teclas arbitraria. Para calcular las probabilidades se construirá un modelo de bigramas, esto es, una tabla de frecuencias relativas de las letras que siguen a una dada. Por ejemplo, detrás de una q aparece una u con probabilidad 1, pero detrás de una h puede aparecer cualquier vocal con distintas probabilidades que deben sumar uno.

EJERCICIO 17. Supongamos que queremos jugar al Nim con las siguientes reglas: cada jugador puede retirar alternativamente un máximo de M fichas, siempre y cuando no repita la última jugada del contrincante (pierde el jugador que no pueda hacer una jugada). Escribe un algoritmo eficiente para calcular si disponemos de una estrategia ganadora o no.

2. Ramificación y poda

EJERCICIO 18. Describe cómo resolver el problema 7 utilizando para ello un algoritmo de ramificación y poda.

EJERCICIO 19. Para garantizar la seguridad del campus el Rector quiere instalar cámaras que vigilen todas sus calles. Una calle está vigilada por una cámara en uno de sus extremos (que llamaremos vértice) y cada cámara puede vigilar todas las calles que convergen en un vértice hasta el vértice siguiente. Describe un algoritmo para encontrar el mínimo número de cámaras necesario para vigilar todo el campus. Por ejemplo, si las calles son $(1, 2)$, $(1, 3)$, $(3, 4)$, $(3, 5)$, $(5, 6)$ y $(6, 7)$, entonces $\{1, 2, 4, 5, 7\}$ es una solución y $\{1, 3, 6\}$ es una solución óptima.

EJERCICIO 20. Supongamos que abordamos el problema 8 mediante ramificación y poda y que representamos la solución como un vector de T enteros x_1, \dots, x_T , donde $x_i = 1$ indica que hay que comprar y $x_i = 0$ que no se compra. Dado un estado $(x_1, \dots, x_t, *, *, \dots, *)$, ¿qué funciones de cota podemos emplear?

EJERCICIO 21. Dado el problema 9 cuyo objetivo es minimizar la longitud de la línea más larga del párrafo, escribe funciones de cota optimista y pesimista

para el conjunto de soluciones representado por $Z = (z_1, z_2, \dots, z_m, *, \dots, *)$ donde z_i es la longitud en caracteres de la línea i .

EJERCICIO 22. En el problema 6, el explorador quiere saber cuál es la combinación más valiosa de peso M independientemente de las rutas permitidas entre la N islas. Es decir, si el objeto i tiene peso w_i y valor v_i , debe elegirse x_i de forma que $\sum_i x_i w_i = M$ y además $\sum_i x_i v_i$ sea el máximo posible. Explica si la siguiente función es una cota válida (y por qué) para las soluciones del tipo $(x_1, \dots, x_n, *, \dots, *)$:

1. Se ordenan los objetos $n+1, \dots, N$ de mayor a menor valor en una lista $B = \{b_1, \dots, b_{N-n}\}$ y se calcula el espacio sobrante $E = M - \sum_{i=1}^n x_i w_i$.
2. Tómesese $i = 1$ y $V = \sum_{i=1}^n x_i v_i$.
3. Mientras $i \leq N-n$ y $E - w[b_i] \geq 0$ hágase $E = E - w[b_i]$, $V = V + v[b_i]$ y $i = i + 1$.
4. Devuélvase V .

EJERCICIO 23. Tenemos N trabajos que pueden ser realizados por la máquina M1 o por la M2, si bien tardan un tiempo distinto en realizar cada trabajo. Una asignación de trabajos se representa mediante un vector de N componentes y el coste total es el máximo de las dos máquinas. Por ejemplo, la asignación $(2, 2, 1, 2)$ con la siguiente tabla de tiempos

	t1	t2	t3	t4
M1	4	4	3	5
M2	2	3	4	4

produce un coste 9. Escribe un algoritmo para calcular una cota optimista y otro para calcular una pesimista de las soluciones del tipo $(x_1, x_2, \dots, x_n, *, \dots, *)$. Explica cómo los aplicarías a este ejemplo.

3. Simulación

EJERCICIO 24. El tiempo que se tarda en realizar una tarea es una variable aleatoria y un programador local ha codificado una función (`coste`) que genera tiempos adecuadamente. Completa el algoritmo siguiente para que escriba el tiempo de espera promedio y el error del cálculo:

```
void simula ( int N ) {
    double S, S2, t;
    S = S2 = 0;
```



```

for ( int n = 1; n <= N; ++n ) {
    t = coste ();
    S += t;
    S2 *= t * t;
}
// Your code
}

```

EJERCICIO 25. ¿Es correcto el siguiente procedimiento para elegir puntos al azar uniformemente dentro de un círculo de radio R ?

1. Genérense dos valores ζ_1 y ζ_2 con un generador uniforme en $[0, 1[$.
2. Hágase $r = R\zeta_1$ y $\phi = 2\pi\zeta_2$.
3. Escribese el resultado en coordenadas cartesianas $x = r \cos \phi$; $y = r \sin \phi$.

EJERCICIO 26. Se toma un número aleatorio ζ dado por un generador uniforme (p. ej., `drand48`) y se le aplica la transformación $t = \sqrt{1 - \sqrt{1 - \zeta}}$. ¿Cuál es la densidad de probabilidad $f(t)$ de los sucesos t generados?

Soluciones de los ejercicios

Ejercicio 1. Supongamos que nuestro grafo tiene $K + 1$ etapas (desde la $k = 0$ hasta la K) y que cada etapa k está formada por un conjunto C_k de nodos. Además, supondremos que la etapa 0 o etapa inicial contiene un solo nodo ($C_0 = \{S\}$) y que la final K contiene también un sólo nodo ($C_K = \{F\}$). El resto de los conjuntos C_k tiene un tamaño arbitrario $N_k = |C_k|$.

En el ejemplo de la figura, tenemos 4 etapas ($K = 3$) con uno, tres, dos y un nodo respectivamente ($N_0 = 1, N_1 = 3, N_2 = 2, N_3 = 1$). En particular, $C_1 = \{a, b, c\}$. El coste de cada arista viene dado por la función $d(x, y)$ siguiente:

$$\begin{array}{lll} & d(a, d)=6 & \\ & d(a, e)=5 & \\ d(S, a)=3 & d(b, d)=4 & d(d, F)=1 \\ d(S, b)=5 & d(b, e)=8 & d(e, F)=2 \\ d(S, c)=7 & d(c, d)=3 & \\ & d(c, e)=6 & \end{array}$$

A continuación procederemos a resolver con bastante detalle este problema. Para ello seguiremos los siguientes pasos:

1. Describir una notación adecuada para el problema.
2. Encontrar un algoritmo recursivo que resuelve el problema, identificando claramente los casos triviales, las decisiones posibles y la reducción del problema que permiten transformar el algoritmo en uno de programación dinámica.
3. Formular una versión eficiente del algoritmo.
4. Demostrar que el algoritmo que se propone es correcto.
5. Comparar la complejidad de dicho algoritmo con la que se obtiene usando programación dinámica.

Notación Representaremos cada camino en el grafo desde S hasta F mediante un vector $\vec{x} = (x_0, x_1, x_2, \dots, x_K) \in C_0 \times C_1 \cdots \times C_K$ en el que cada componente x_k indica por qué nodo de la etapa k pasa el camino. La longitud de este camino es

$$L(\vec{x}) = \sum_{k=1}^K d(x_{k-1}, x_k).$$

En nuestro ejemplo, $\vec{x} = (S, a, d, F)$ representa un camino de longitud $d(S, a) + d(a, d) + d(d, F) = 10$. Al conjunto de caminos que van desde S hasta un nodo r de la etapa k lo denotaremos como

$$D(r) = C_0 \times C_1 \cdots \times C_{k-1} \times \{r\}.$$

Con todo esto el problema del grafo multietapa se puede enunciar de la siguiente manera: obténgase

$$\arg \min_{\vec{x} \in D(F)} L(\vec{x})$$

Algoritmo recursivo Para obtener un algoritmo recursivo, es conveniente seguir los siguientes pasos:

1. Comenzar buscando el mejor valor posible de la función que se quiere optimizar, para después encontrar el modo por el que este se obtiene.

En nuestro caso, intentaremos hallar la longitud mínima

$$L_{\min}(F) = \min_{\vec{x} \in D(F)} L(\vec{x}) \quad (1)$$

2. Englobar el problema en un conjunto más general de subproblemas.

Dado que queremos hallar el camino más corto de S a F la generalización más inmediata es pensar en encontrar el camino más corto para cualquier nodo de inicio y cualquiera de terminación. Aunque es posible hacerlo de esta manera, es suficiente con plantearse sólo los caminos que llevan desde S a un nodo arbitrario q .

3. Encontrar la solución de los casos triviales.

En este caso existe un caso realmente trivial, que es $q = S$. Dado que la distancia que hay que recorrer para llegar al nodo de partida es cero, podemos escribir

$$L_{\min}(S) = 0$$

4. Encontrar qué decisiones se pueden realizar y la reducción que se origina.

Dado el problema de encontrar el camino más corto que lleva hasta el nodo q de la etapa k , es obvio que cualquier camino que lleva a q debe pasar inmediatamente antes por un nodo r de la etapa $k-1$. La elección que se haga de r puede entenderse como una *decisión* que afecta a la solución del problema. Por otro lado, sabemos que $L_{\min}(r)$ es la longitud del camino más corto que conduce a r , luego $L_{\min}(r) + d(r, q)$ es la distancia menor necesaria para llegar hasta q pasando por r . En resumen, la opción de pasar por r reduce el problema de partida a otro más sencillo: el asociado a r .

5. Definir cómo se obtiene la mejor solución.

Dado que no existe ninguna restricción que nos obligue a pasar por un nodo r en particular, interesa elegir aquel que haga mínimo el camino hasta q . Por tanto:

$$L_{\min}(q) = \begin{cases} \min_{r \in C_{k-1}} \{L_{\min}(r) + d(r, q)\} & \text{si } q \in C_k \wedge k > 0 \\ 0 & \text{si } q = S \end{cases} \quad (2)$$

Demostración de la fórmula recursiva Para demostrar la corrección de la fórmula recursiva procederemos por inducción: la solución del caso trivial $L(S) = 0$ es evidentemente correcta, por lo que supondremos que la fórmula recursiva es válida para cualquier $i < k$ y tomaremos un nodo $q \in C_k$. Entonces, sustituyendo $L(r)$ por su definición (1) en (2) y reagrupando los términos obtenemos

$$L_{\min}(q) = \min_{r \in C_{k-1}} \{ \min_{\vec{x} \in D(r)} \{L(\vec{x}) + d(r, q)\} \}$$

Dado que $L(\vec{x}) + d(r, q)$ representa la longitud de un camino cuyas componentes son $(x_0, x_1, \dots, x_{k-2}, r, q)$ denotaremos este camino como $\vec{x} \times \{q\}$ y obtenemos

$$L_{\min}(q) = \min_{r \in C_{k-1}} \min_{\vec{x} \in D(r)} L(\vec{x} \times \{q\})$$

Finalmente, como todos los mínimos están bien definidos, podemos escribir

$$L_{\min}(q) = \min_{\vec{y} \in D(q)} L(\vec{y})$$

que es precisamente la definición del camino mínimo.

Programación dinámica del algoritmo Una implementación recursiva con programación dinámica permite mejorar la eficiencia del algoritmo:

```
int d[][]; // datos: distancias
Graph G;   // datos: grafo
int f ( GraphNode q ) {
    if ( A[q] < 0 ) // no calculado
        A[q] = MAX_INT;
    int k = G.stage(q); // etapa de q
    for ( GraphNode r = G.firstNodeIn[k-1]; G.stage(r) < k; r = G.nextNode(r) ) {
        A[q] = std::min( A[q], f(r) + d[r][q] );
    }
    return A[q];
}

int main () {
    for ( GraphNode q = G.firstNode(); q != G.lastNode(); q = G.nextNode(q) )
        A[q] = -1; // inicializa almacen
    return f( G.lastNode() );
}
```

Complejidad del algoritmo Si denotamos como $t(q)$ el coste temporal de evaluar la distancia mínima al nodo q , este coste es:

$$t(q) = \begin{cases} c_1 & \text{si } q = S \\ c_2 + \sum_{r \in C_{k-1}} t(r) & \text{si } q \in C_k \wedge k > 0 \end{cases}$$

Tenemos pues que $t(S) = c_1$. En cambio, si $q \in C_1$, entonces $t(q) = c_2 + N_0 t(S) = N_0 c_1 + c_2$, es decir, el coste temporal del cálculo del camino para todos los nodos q de la segunda etapa es idéntico.

Como consecuencia de lo anterior, si $q \in C_2$, entonces

$$t(q) = c_2 + N_1(N_0 c_1 + c_2) = N_0 N_1 c_1 + (N_1 + 1)c_2.$$

Nótese que, de nuevo, el coste temporal coincide para todos los nodos q de C_2 . Si $M = N_0 = N_1 = \dots = N_K$. Entonces, una inducción trivial nos lleva a la conclusión de que para un nodo de la etapa k

$$t(q) \simeq M^k(c_1 + c_2)$$

por lo que el coste del algoritmo recursivo es exponencial.

Si se utiliza *programación dinámica* (iterativa o no), el uso de un almacén para guardar las soluciones de los subproblemas evita llamar más de una vez a la función L con el mismo argumento, por lo que el número máximo de llamadas (y, por tanto, el coste espacial) es $N_1 + N_2 + \dots + N_K = KN$, esto es, el número total N de nodos del grafo.

Por otro lado, el cálculo de $L(q)$, siendo q un nodo de la etapa k , requiere realizar N_{k-1} operaciones (consultas al almacén y operaciones aritméticas). Por tanto, el coste total es proporcional a $t(k) = \sum_k N_k N_{k-1}$. Por tanto, podemos escribir $t(K) \in \mathcal{O}(KM^2)$. Evidentemente, esto supone una eficiencia claramente superior a la del algoritmo inicial.

Ejercicio 1

Ejercicio 2. Supongamos que $f(n, m)$ representa el tamaño máximo de las subsecuencias no decrecientes contenidas en $\{a_1, a_2, \dots, a_n\}$ formadas por elementos que no superan a m (nótese que, en general, $n \leq N$). Entonces,

$$f(n, m) = \begin{cases} 0 & \text{si } n = 0 \\ f(n-1, m) & \text{si } a_n > m \wedge n > 0 \\ \max(f(n-1, m), 1 + f(n-1, a_n)) & \text{si } a_n \leq m \wedge n > 0 \end{cases}$$

La llamada principal debe hacerse con todos los elementos, $n = N$, y sin poner ninguna restricción, esto es, $f(N, M)$ donde $M \geq \max\{a_1, \dots, a_N\}$. La función anterior puede implementarse como sigue:

```
int a[N+1]; // datos
int fr (int n, int m) {
    if( n == 0 ) return 0;
    else if( a[n] > m ) return fr(n-1, m);
    else return max( fr(n-1,m), 1 + fr(n-1, a[n]) );
}
```

Transformado mediante programación dinámica queda

```
int a[N+1];
int A[N+1,M+1]; // almacenar
int pdr (int n, int m) {
    if( A[n][m] < 0 ) { // si no está calculado
        if ( n == 0 )
            A[n][m] = 0;
        else if ( a[n] > m )
            A[n][m] = pdr(n-1, m);
        else
            A[n][m] = max( pdr(n-1, m), 1 + pdr(n-1, a[n]) );
    }
    return A[n][m];
}

int f (int N, int M) {
    for ( int n = 0; n <= N; ++n )
        for ( int m = 0; m <= M; ++m )
            A[n][m] = -1; // inicializar
    return pdr(N, M);
}
```

Su transformación en una versión iterativa requiere llenar el almacén en un orden tal que las consultas al almacén se hagan siempre a posiciones calculadas (y almacenadas) previamente. En este caso, todas las consultas se hacen a posiciones con n y con m menor, luego cualquier recorrido ascendente en n y m es adecuado y una posible versión iterativa del algoritmo es la siguiente:

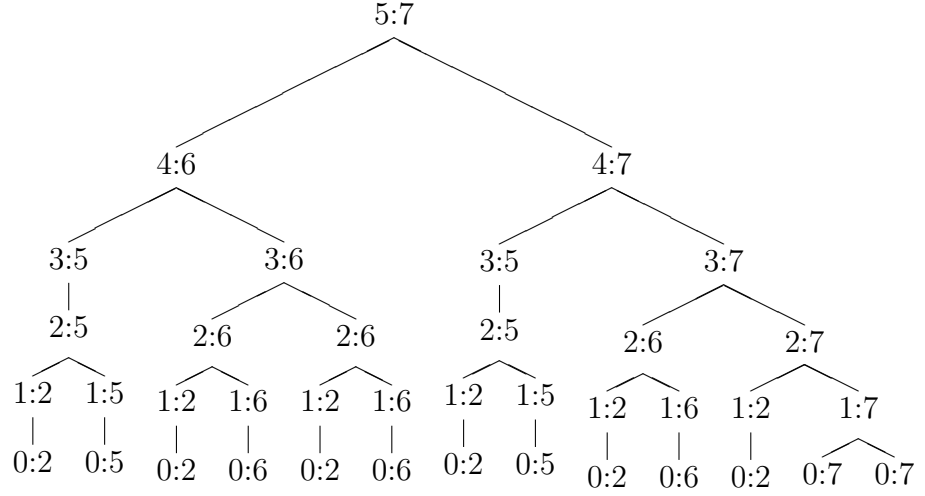
```
int a[N+1];
int A[N+1,M+1];      // almacén

int pdi (int N, int M) {
    for ( int n = 0; n <= N; ++n ) {
        for ( int m=0; m <= M; ++m ) {
            if ( n == 0 )
                A[n][m] = 0;
            else if ( a[n] > m )
                A[n][m] = A[n-1][m];
            else
                A[n][m] = max( A[n-1][m], 1+A[n-1][a[n]] );
        }
    }
    return A[N][M];
}
```

Si observamos que de los M valores posibles, en realidad sólo pueden aparecer uno por cada elemento a_n distinto, es sencillo comprobar que tanto **pdr** como **pdi** tienen un coste $\mathcal{O}(N^2)$. Por ejemplo, el cuerpo de la función **pdr** (el bloque condicional externo) sólo se ejecuta cada vez que el almacén está vacío, lo que ocurre (aproximadamente) un máximo de N^2 veces. El cuerpo de **pdr** sólo contiene operaciones de coste constante, entre ellas llamadas recursivas a **pdr** (una o dos). Por tanto, también el número de llamadas a **pdr** está acotado por el doble de N^2 y también el coste de las operaciones externas al bloque condicional.

El árbol de llamadas tiene 38 nodos, de los cuales sólo 17 son distintos (véase la figura). Por tanto, la función recursiva hace 38 llamadas, mientras que **pdr** sólo hace 17. Finalmente, **pdi** hace del orden de $(N+1) \times (M+1) = 48$ cálculos.

Ejercicio 2



Ejercicio 3. La solución voraz consiste en mover en cada instante el brazo que está más cerca del punto a soldar. En este ejemplo, la secuencia de movimientos voraz es: $r_0 \rightarrow r_1$, $r_0 \rightarrow r_2$, $r_2 \rightarrow r_3$, $r_3 \rightarrow r_4$ y la distancia total recorrida es $100 + 89 + 60 + 50 = 299$. En cambio, la distancia óptima es $d(r_0, r_1) + d(r_1, r_2) + d(r_0, r_3) + d(r_2, r_4) = 100 + 100 + 45 + 36 = 281$.

Supongamos que $f(n, k)$ es la distancia mínima que debe recorrer aún el robot tras soldar r_1, \dots, r_n si uno de los brazos está en r_k (el otro, debe estar necesariamente en r_n). Es evidente que la función $f(n, k)$ tiene solución trivial si $n = N$. En ese caso ya se han soldado todos los puntos y la distancia que falta por recorrer es 0. En general, tenemos dos opciones:

1. mover hasta el punto r_{n+1} el brazo que se encuentra en la posición r_n y resolver el problema $f(n+1, k)$;
2. mover el brazo que está en r_k y resolver el problema $f(n+1, n)$.

Por tanto, la siguiente función recursiva resuelve el problema

$$f(n, k) = \begin{cases} 0 & \text{si } n = N \\ \min(d(n, n+1) + f(n+1, k), d(k, n+1) + f(n+1, n)) & \text{si } n < N \end{cases}$$

Una implementación de $f(n, k)$ que usa programación dinámica recursiva se muestra en la figura 4.

```

struct Sol {
    double dist;    // distancia recorrida
    stack<int> s;    // la soluci3n
};

init () {
    for ( int n = 0; n < N; ++n )
        for ( int k = 0; k < N; ++k )
            A[n][k] = -1;
}

Sol f ( int n, int k ) {
    Sol r1, r2, res;
    if ( A[n][k] < 0 ) {          // consulta el almac3n
        if ( n == N ) {
            res.dist = 0;
        } else {
            r1 = f(n+1, k);
            r2 = f(n+1, n);
            if ( r1.dist + d(n, n+1) <= r2.dist + d(k, n+1) ) {
                res.dist = r1.dist + d(n, n+1);
                res.s = r1.s;
                res.s.push(n);
            } else {
                res.dist = r2.dist + d(k, n+1);
                res.s = r2.s;
                res.s.push(k);
            }
        }
        A[n][k] = res;
    }
    return A[n][k];
}

```

Figura 4: Programa recursivo para el ejercicio 3.

El coste temporal de este algoritmo es, en el peor caso, proporcional a N^2 (la justificación de este coste se basa en que no pueden realizarse más de $2N^2 + 1$ llamadas recursivas a la función).

El árbol de llamadas recursivas que se genera con esta función es el dibujado en la figura 5.

Como puede verse, el algoritmo realiza 15 llamadas en vez de las $2^{N+1} = 32$ de la función recursiva. Un algoritmo iterativo rellenaría el almacén A completo. Como hay 5 valores posibles de n y 4 posibles de k , el tamaño del almacén es 20.

Ejercicio 3

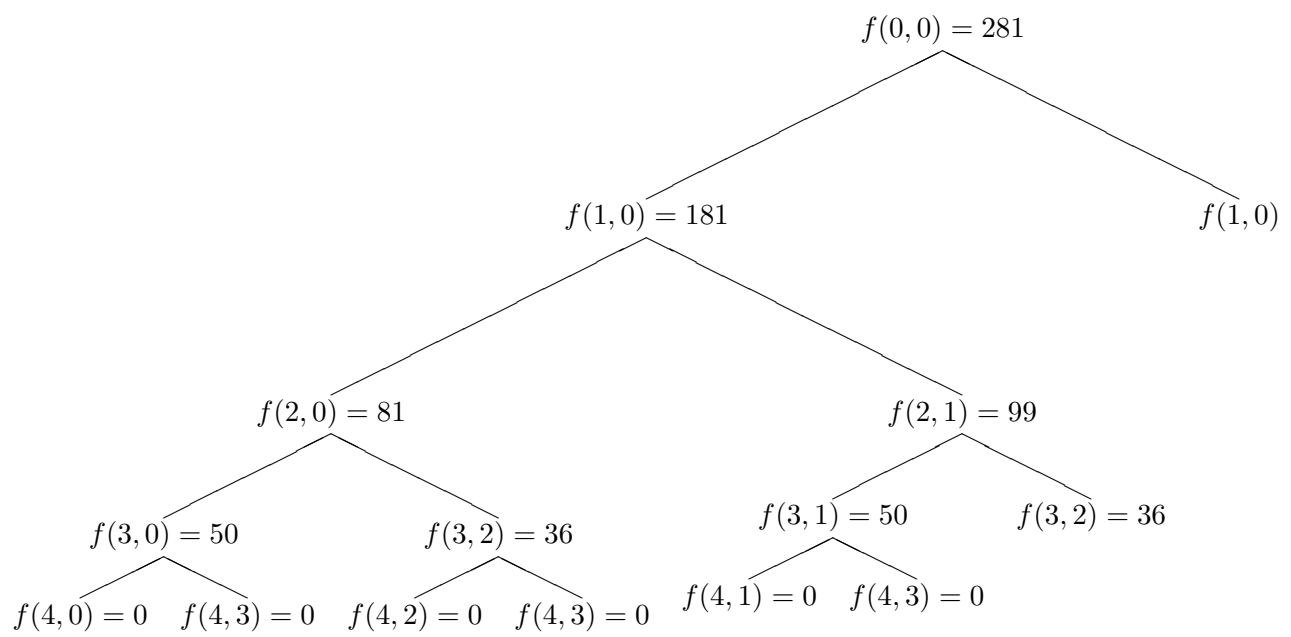


Figura 5: Árbol de llamadas recursivas del ejercicio 3.

Ejercicio 4. Este problema es semejante al clásico problema de la mochila discreto. La única diferencia es que se puede repetir (hasta c_i veces) una selección. Por tanto, una función que resuelve el problema es la siguiente:

$$f(n, m) = \begin{cases} 0 & \text{si } m = 0 \\ \infty & \text{si } n = 0 \wedge m > 0 \\ \min_{k \leq c_n: m \leq k * v_n} \{k + f(n - 1, m - k * v_n)\} & \text{en el resto de los casos} \end{cases}$$

Para realizar una implementación usando programación dinámica ha de incluirse un almacén tal y como sigue.

```
std::vector<int> v; // denominaciones
std::vector<int> c; // disponible
std::map<int, map<int, int> > A; // almacenar A[n][m]

int
monedas (M) {
    for ( n = 0; n < N; ++n ) {
        for ( m = 0; m < M; ++m ) {
            if ( m == 0 ) A[n][m] = 0;
            else if ( n == 0 ) A[n][m] = MAX_INT;
            else {
                A[n][m] = MAX_INT;
                for ( k = 0; k < c[n] && k*v[n] <= m; ++k )
                    A[n][m] = min( A[n][m], k + A[n-1][m-k*v[n]] );
            }
        }
    }
}
```

Ejercicio 4

Ejercicio 5. Si $S(x)$ es el conjunto de subordinados (directos) de la persona x y $a = 1$, entonces necesariamente $a = 0$ para todos los $y \in S(x)$ y

$$f(x, 1) = \sum_{y \in S(x)} f(y, 0).$$

Observa que la expresión anterior da cuenta del caso trivial $S(x) = \emptyset$ en el que $f(x, a) = 0$.

En el caso en que $a = 0$, entonces $y \in S(x)$ puede acudir o no. En el primer caso, se puede invitar a $f(y, 0)$ subordinados de y y en el segundo a $f(y, 1)$ (además de a y). Como a priori no es posible saber cuál es la mejor opción, debe calcularse recursivamente f y elegir la opción que permite más invitados. En resumen:

$$f(x, a) = \begin{cases} \sum_{y \in S(x)} f(y, 0) & \text{si } a = 1 \\ \sum_{y \in S(x)} \max(1 + f(y, 1), f(y, 0)) & \text{si } a = 0 \end{cases}$$

Una posible implementación en C++ usando programación dinámica es la siguiente:

```
int pdr (int x, bool a) {
    if ( A[x][a] < 0 ) {
        A[x][a] = 0;
        for ( std::set<int>::iterator i = S[x].begin(); i != S[x].end(); ++i )
            A[x][a] += a ? max ( pdr (*i, 0), 1 + pdr (*i, 1) ) : pdr (*i, 0);
    }
    return A[x][a];
}
```

Ejercicio 5

Ejercicio 6. Es este caso conviene calcular una función booleana $f(q, m)$ que sea cierta si es posible llegar al nodo q con una carga m . Para hallar una ruta adecuada se calcula $f(F, M)$. Trivialmente, $f(S, m)$ es cierta si $m = w(S)$ y falsa en caso contrario. En general para un nodo q de la etapa k

$$f(q, m) = \begin{cases} \text{false} & \text{if } w(q) > m \\ \bigvee_{r \in E_{k-1}} f(r, m - w(q)) & \end{cases}$$

donde E_k representa el conjunto de islas de la etapa k . Por tanto, una posible implementación iterativa es la siguiente:

```
bool pdi ( int M ) {
    int m, k;
    std::set<int>::iterator q, r;
    for ( m = 0; m <= M; ++m )
        A[ 0 ][ m ] = ( m == w[0] );
    for ( k = 1, k <= K; ++k )
        for ( q = E[k].begin(); q != E[k].end(); ++q )
            for ( m = 0; m <= M; ++m ) {
                A[ *q ][ m ] = false;
                if ( w[ q ] <= m )
                    for ( r = E[k-1].begin(); r != E[k-1].end(); ++r )
                        A[ *q ][ m ] = A[ *q ][ m ] || A[ *r ][ m-w[*q] ];
            }
    return A[ *E[K].begin() ] [ M ];
}
```

Ejercicio 6

Ejercicio 7. El problema puede plantearse del siguiente modo: dado un grafo G formado por un conjunto finito de N nodos $Q = \{a, b, c, \dots\}$ y cuyas aristas se caracterizan por un número real $v(a, b)$, se trata de obtener un camino entre dos nodos en el que el valor mínimo de v es lo mayor posible. Dado que transmitir unos datos al mismo nodo tienen un coste nulo (tiempo cero), podemos asumir que $v(q, q) = \infty$. Por el contrario, si una arista (p, q) no pertenece al grafo, podemos tomar $v(p, q) = 0$. Una forma natural de representar las rutas entre dos ordenadores es mediante una lista de los nodos intermedios. Por ejemplo, las rutas (S, a, d, c, F) y (S, a, b, F) permiten llegar desde S a F . Entonces, el problema consiste en obtener

$$\arg \max_{\vec{x} \in D} \phi(\vec{x})$$

donde ϕ representa la velocidad de transmisión por el camino \vec{x} .

Algoritmo recursivo Seguimos los siguientes pasos que nos guían en la búsqueda de un procedimiento recursivo adecuado:

1. Se reduce el problema a hallar el valor óptimo:

$$\max_{\vec{x} \in D} \phi(\vec{x})$$

2. Se generaliza el problema al de obtener un camino óptimo de longitud no superior a n para llevar los datos desde S a cualquier nodo destino q . Esto es:

$$V_{\max}(q, n) = \max_{\vec{x} \in D_n(q)} \Phi(\vec{x}) \quad (3)$$

La generalización para todos los nodos es bastante intuitiva. El hecho de considerar caminos hasta una cierta longitud es conveniente para evitar las repeticiones de nodos en el camino, esto es, la aparición de bucles.

3. Se resuelven los casos triviales. Si la longitud del camino n es uno, el único camino de S a q es (S, q) y la velocidad de transmisión es $v(S, q)$. Luego,

$$V_{\max}(q, 1) = v(S, q)$$

4. Para construir un camino de S a q de longitud n como máximo, la decisión de traer los datos desde el nodo r (a donde habrán llegado anteriormente desde S usando, como máximo, $n - 1$ conexiones y a una velocidad $V_{\max}(r, n - 1)$) nos conduce a que la velocidad de transmisión es $\min(V_{\max}(r, n - 1), v(r, q))$.

5. Juntando todos estos elementos, la solución a nuestro problema puede encontrarse de la siguiente forma:

$$V_{\text{máx}}(q, n) = \begin{cases} v(S, q) & \text{si } n = 1 \\ \max_{r \neq q} \min(V_{\text{máx}}(r, n-1), v(r, q)) & \text{si } n > 1 \end{cases} \quad (4)$$

Aunque resulta evidente que pueden imponerse más restricciones a r en la fórmula anterior, dejaremos este aspecto para más adelante.

La incorporación de un almacén para mejorar la eficiencia puede realizarse de forma análoga a la del problema 1.

Complejidad del algoritmo El coste temporal de calcular la velocidad máxima hasta un nodo q pasando como máximo por n conexiones es

$$t(q, n) = \begin{cases} c_1 & \text{si } n = 1 \\ c_2 + \sum_{r \neq q} t(r, n-1) & \text{si } n > 1 \end{cases}$$

Si abreviamos $N = |Q| - 1$, es evidente que

$$t(q, n) \geq c_1 N^{n-1}$$

Por tanto, como la solución del problema viene dada por $V_{\text{máx}}(F, N)$, podemos concluir que la complejidad de la solución recursiva es peor que N^N .

La programación dinámica modifica el esquema añadiendo una tabla $A(q, n)$ de almacenamiento de soluciones cuyo tamaño crece con N^2 . No se realizan, por tanto, mas de N^2 llamadas a la función y, en cada una, se calcula el mínimo de N elementos (que se consultan en la tabla). Por ello, el coste del algoritmo, usando programación dinámica, es $\mathcal{O}(N^3)$.

Comentario. El problema anterior puede resolverse de otra forma, usando también programación dinámica, con un coste N^2 , inferior al obtenido anteriormente. Para ello, basta con utilizar un procedimiento similar al que usa el algoritmo de Floyd y Warshall. Básicamente, consiste en redefinir el significado del parámetro n : en vez de representar la longitud del camino, representa el índice máximo de los nodos intermedios. Esto es, $D_n(q)$ representa todos los caminos que llegan a q sin pasar por el nodo n ni por ningún otro nodo $m > n$. Ejercicio 7

Ejercicio 8. Supongamos que $f(t, m)$ es la inversión mínima para mantener la flota a partir del mes t (y hasta el final del mes T) si en ese momento la antigüedad de los vehículos es m . Esto significa que t es un entero entre 1 y T y m un valor entre 0 y M . Supondremos también que no consideramos el mantenimiento después del mes T y, por tanto,

$$f(T + 1, m) = 0$$

para cualquier valor de m (también es el resultado cero si $t > T - M + m$).

Por otra parte, en el mes $t > 0$ puede decidirse renovar ($x_t = 1$) o no ($x_t = 0$) la flota. Como no sabemos a priori cuál de las dos opciones es la óptima, debemos comparar ambas y tomar la que produce una inversión mínima. Si no se renueva, es obvio que la inversión es la misma que para los $t + 1$ meses anteriores y que la antigüedad en $t + 1$ es $m + 1$. Si se toma la decisión contraria, entonces la inversión aumenta en w_t , pero la antigüedad en $t + 1$ será 1. En resumen:

$$f(t, m) = \min(f(t + 1, m + 1), w_t + f(t + 1, 1))$$

Sin embargo, la fórmula anterior no es correcta debido a dos causas:

1. la recursión no se detiene debido a que no se han incluido los casos triviales;
2. si $m = M$ es preciso renovar.

Por ello, la fórmula correcta es:

$$f(t, m) = \begin{cases} 0 & \text{si } t > T \\ w_t + f(t + 1, 1) & \text{si } t \leq T \wedge m = M \\ \min(f(t + 1, m + 1), w_t + f(t + 1, 1)) & \text{en los demás casos} \end{cases}$$

Implementado como código:

```
int w[T+1]; // datos

int fr (int t, int n) {
    if( t > T ) return 0;
    else if( m == M ) return w[t] + fr(t+1, 1);
    else return min( fr(t+1, m+1), w[t] + fr(t+1, 1) );
}
```

Es evidente que en el peor caso, el coste temporal del algoritmo crece exponencialmente con M : si N es grande, siempre se realizan dos llamadas recursivas que generan un árbol binario de profundidad M , esto es, con un número de nodos del orden de 2^M .

Transformado mediante programación dinámica queda:

```
int w[T+1];
int A[T+1][M+1];          // almacén

int
pdr (int t, int m) {
    if( A[t][m] < 0 ) {    // si no está calculado
        if ( t > T )
            A[t][m] = 0;
        else if ( m == M )
            A[t][m] = w[t] + pdr(t+1, 1);
        else
            A[t][m] = min( pdr(t+1, m+1), w[t] + pdr(t+1, 1);
    }
    return A[t][m];
}

int
f (int T, int M) {
    for (int t = 0; t <= T; ++t)
        for (int m = 0; m <= M; ++m)
            A[t][m] = -1;    // inicialización

    return pdr(T, M);
}
```

El coste temporal es ahora distinto. La función principal tiene un bucle doble con coste TM . La función recursiva no puede ser llamada más de $(T+1)(M+1) + 1$ veces: una desde la función principal y dos por cada valor del almacén que puede ser cero en la línea recursiva. Por tanto, el coste temporal es $\mathcal{O}(TM)$.

Una versión iterativa del algoritmo anterior requiere que $A[t+1][m+1]$ y $A[t+1][0]$ hayan sido calculados previamente. Esto es sencillo de conseguir, ya que basta con que los elementos se calculen por orden decreciente de t :

```

int w[T+1];
int A[T+1][M+1];          // almacen

int pdi (int T, int M) {
    for (int t = T; t >= 0; --t) {
        for (int m = 0; m <= M; ++m) {
            if ( t > T )
                A[t][m] = 0;
            else if ( m == M )
                A[t][m] = w[t]+A[t+1][1];
            else
                A[t][m] = min( A[t+1][m+1], w[t] + A[t+1][1] );
        }
    }
    return A[T][M];
}

```

Es evidente que el coste es ahora proporcional al número de iteraciones del algoritmo, esto es, a TM . Por otro lado, es posible utilizar menos memoria si se guardan sólo los resultados de la fila $t + 1$ de la matriz. Esto se puede conseguir sustituyendo $A[t]$ por $A[t\%2]$ y $A[t + 1]$ por $A[1 - t\%2]$ en el algoritmo anterior.

Ejercicio 8

Ejercicio 9. Sea la función $f(n, m)$ que devuelva la longitud mínima de la línea más larga cuando las n primeras palabras se distribuyen en m líneas. Es evidente que

$$f(n, 1) = \sum_{n=1}^N s_n$$

y que

$$f(0, m) = 0$$

son casos triviales de resolver.

En general, si k palabras forman la línea m , entonces, la línea más larga puede ser la m , cuya longitud es $\sum_{i=n-k+1}^n s_i$, o una de las $m-1$ anteriores, en cuyo caso la longitud máxima es $f(n-k, m-1)$. Como no podemos prever qué valor de k es el óptimo, entonces debemos escribir

$$f(n, m) = \begin{cases} 0 & \text{si } n = 0 \\ \sum_{n=1}^N s_n & \text{si } m = 1 \\ \min_{k=1, \dots, n} \max(f(n-k, m-1), \sum_{i=n-k+1}^n s_i) & \text{en otro caso} \end{cases}$$

Para implementar esta función de forma eficiente podemos usar programación dinámica como sigue:

```
int L (int i, int j ) {
    int l = 0;
    for ( int k = i; k <= j; ++k )
        l += s[k];
    return l;
}

int pdi ( int N, int M ) {
    int n, m;
    for ( m = 0; m <= M; ++m )
        A[0][m] = 0;
    for ( n = 1; n < N; ++n ) {
        A[n][1] = L(1, N);
        for ( m = 2; m <= M; ++m ) {
            A[n][m] = MAX_INT;
            for ( k = 1; k <= n; ++k )
                A[n][m] = min ( A[n][m], max ( A[n-k][m-1], L(n-k+1, n) ) );
        }
    }
    return A[N][M];
}
```

Ejercicio 9

Ejercicio 10. Aplicaremos las líneas de análisis habituales:

1. Calcularemos la probabilidad de que se suspendan todas las asignaturas y la estrategia que minimiza este peligro. En este caso, tenemos $M = 4$ días y $N = 3$ asignaturas. Una estrategia consiste en asignar un cierto número de días x_k a cada asignatura ($k = 1, \dots, N$) de modo que $\sum_{k=1}^N x_k = M$.
2. Consideramos el problema más general de calcular la probabilidad mínima $f(n, m)$ de suspender las asignaturas $1, 2, \dots, n$ si disponemos de m días para ellas:

$$f(n, m) = \min_{\vec{x} \in D(n, m)} \prod_{k=1}^n p_k(x_k)$$

$$\text{con } D(n, m) = \{\vec{x} \in (0, \dots, m)^n : \sum_{k=1}^n x_k = m\}.$$

3. Los siguientes casos son triviales: $f(1, m) = p_1(m)$ y $f(n, 0) = \prod_{k=1}^n p_k(0)$.
4. Planteamos la relación de recurrencia (para $n > 1$):

$$\begin{aligned} f(n, m) &= \min_{0 \leq x_n \leq m} \left(p_n(x_n) \min_{\vec{x} \in D(n-1, m-x_n)} \prod_{k=1}^{n-1} p_k(x_k) \right) = \\ &= \min_{0 \leq x_n \leq m} p_n(x_n) f(n-1, m-x_n) \end{aligned}$$

Nótese que el segundo caso trivial se puede deducir del primero y de la relación de recurrencia.

Puede comprobarse que los resultados obtenidos coinciden con los que se obtienen si se utiliza un esquema paralelo al del problema de la mochila para minimizar

$$\sum_{k=1}^N \log p_k(x_k)$$

con la restricción

$$\sum_{k=1}^n x_k = M.$$

Una implementación con programación dinámica iterativa puede hacerse como sigue:

```

double f (int N, int M) {
    double A[N][M];
    for ( m = 0; m <=M; ++m )
        A[1][m] = p[1][m];
    for ( int n = 2; n <= N; ++n ) {
        for ( m = 0; m <= M; ++m ) {
            A[n][m] = MAX_DOUBLE;
            for ( x = 0; x <= m; ++x )
                A[n][m] = std::min( A[n][m], p[n][x] * f( n - 1, m - x ) );
        }
    }
    return A[N][M];
}

```

Ejercicio 10

Ejercicio 11. Para resolver el problema, seguimos los pasos habituales:

1. Inicialmente, buscaremos qué variables permiten generar la cadena y, más tarde, cuál es su árbol de análisis sintáctico.
2. La familia de problemas que intentaremos resolver es, dada la cadena $w \in T^*$, obtener el conjunto de variables $C(i, j) \subset V$ que permiten generar la subcadena $w_i \dots w_j$. La solución al problema original viene dada por $C(1, |w|)$.
3. Los casos triviales son aquellos en los que la cadena es de longitud uno y vienen resueltos directamente por la función λ_1 , ya que $C(i, i) = \lambda_1(w_i)$.
4. Para los casos no triviales, esto es, cuando $j > i$, podemos utilizar la relación de recurrencia:

$$C(i, j) = \bigcup_{i \leq k < j} \lambda_2(C(i, k), C(k + 1, j))$$

Luego la fórmula recursiva que resuelve el problema del análisis sintáctico de una cadena es

$$C(i, j) = \begin{cases} \lambda_1(w_i) & \text{si } i = j \\ \bigcup_{i \leq k < j} \lambda_2(C(i, k), C(k + 1, j)) & \text{si } j > i \end{cases} \quad (5)$$

Ejercicio 11

Ejercicio 12. Basta con optimizar una función $f(n, m, w)$ que represente el número de calorías mínimo eligiendo platos del 1 al n con presupuesto máximo m y contenido protéico mínimo w . Obviamente,

$$f(n, m, w) = \begin{cases} \infty & \text{si } n = 0 \wedge w > 0 \\ 0 & \text{si } n = 0 \wedge w = 0 \\ f(n-1, m, w) & \text{si } p - n > m \\ \min(f(n-1, m, w), c_n + f(n-1, m - p_n, w - q_n)) & \text{en los demás casos} \end{cases}$$

La implementación de esta función mediante programación dinámica puede hacerse como sigue:

```
int
pdr ( int n, int m, int w ) {
    if ( A[n][m][w] < 0 ) {
        if ( n == 0 )
            A[n][m][w] = (w > 0)? MAX_INT : 0;
        else if ( p[n] <= m )
            A[n][m][w] = min ( pdr ( n - 1, m, w ),
                               c[n] + pdr( n - 1, m - p[n], w - q[n]) );
        else
            A[n][m][w] = pdr( n - 1, m, w );
    }
    return A[n][m][w];
}
```

Ejercicio 12

Ejercicio 13. Evidentemente, dadas las suposiciones anteriores, es necesario para que exista al menos una solución que se cumpla la condición $w_n \leq M$ para todos los valores de n .

Para resolver este problema, estudiaremos primero el caso en el que el número de discos es limitado por un valor máximo K . Si somos capaces de resolver este problema, podemos realizar una búsqueda binaria en el rango $1, \dots, N$ (dado que siempre es posible guardar cada programa en un disco, la solución estará entre 1 y N). Dicha búsqueda puede realizarse con un coste adicional $\log(N)$.

Notación Cada posible solución del problema es un vector $\vec{x} = (x_1, \dots, x_N)$ cuyas componentes indican en qué disco ha sido almacenado el programa n . Por tanto, cada componente x_n está en el rango $R = \{1, \dots, K\}$ y $\vec{x} \in R^N$. Por otro lado, el contenido del disco k es

$$C(k, \vec{x}) = \sum_{n: x_n = k} w_n \quad (6)$$

En este problema no se trata de optimizar directamente el espacio, sino simplemente de saber si es posible acomodar todos los programas en los K discos. Por ello, definimos la función lógica

$$F(\vec{x}) = \bigwedge_{k=1}^K (C(k, \vec{x}) \leq M)$$

que nos dice si \vec{x} es una solución válida (esto es, satisface las restricciones de espacio). Por tanto, el objetivo es encontrar \vec{x} tal que $F(\vec{x})$ es cierto.

Algoritmo recursivo Para resolver el problema seguimos los pasos habituales:

1. Nos limitamos a encontrar si existe algún valor de \vec{x} tal que $F(\vec{x})$ es verdadero, sin preocuparnos en un principio de cuál es vector \vec{x} que lo permite.
2. Englobamos el problema en uno más general: cómo introducir n programas (del 1 al n) en K discos cuyas capacidades son m_1, m_2, \dots, m_K . La solución vendrá dada por una función lógica

$$f(n, m_1, \dots, m_K) = \bigvee_{\vec{x} \in R^N} F(\vec{x}, m_1, \dots, m_K) \quad (7)$$

donde F es cierto si \vec{x} es una posible solución del problema:

$$F(\vec{x}, m_1, \dots, m_K) = \bigwedge_{k=1}^K (C(k, \vec{x}) \leq m_k)$$

3. Existe un caso de solución trivial: si $n = 0$ (no hay que almacenar ningún programa), entonces la función $f(0, m_1, \dots, m_K)$ es trivialmente cierta.
4. La decisión de introducir el programa n en el disco k (lo cuál sólo es posible si $m_k \geq w_n$) reduce el problema a estudiar si es posible introducir los $n - 1$ primeros programas en K discos de capacidades respectivas $m_1, m_2, \dots, m_k - w_n, \dots, m_K$.
5. La solución del problema puede enunciarse, por tanto, de la siguiente forma:

$$f(n, m_1, \dots, m_K) = \begin{cases} \text{cierto} & \text{si } n = 0 \\ \bigvee_{k: m_k \geq w_n} f(n - 1, m_1, \dots, m_k - w_n, \dots, m_K) & \text{si } n > 0 \end{cases} \quad (8)$$

Demostración Sustituyendo f en (8) de acuerdo con la definición (7) se obtiene

$$f(n, m_1, \dots, m_K) = \bigvee_{k: m_k \geq w_n} \bigvee_{\vec{x} \in R^{n-1}} F(\vec{x}, m_1, \dots, m_k - w_n, \dots, m_K)$$

Dado que $\vec{x} \times \{k\} \in R^n$ y que $m_k \geq w_n$ implica que $C(k, \vec{x} \times \{k\}) \leq m_k$ podemos reescribir la ecuación anterior de la siguiente forma

$$f(n, m_1, \dots, m_K) = \bigvee_{\vec{x} \in R^n} F(\vec{x}, m_1, \dots, m_K)$$

que es precisamente la definición propuesta en (7).

La implementación según un esquema de programación dinámica es muy semejante a la del clásico problema de la mochila (véanse las notas del curso): tan sólo hay que reemplazar el almacén bidimensional $A[n][m]$ por uno K -dimensional $A[n][m1][m2] \dots [mK]$.

Complejidad del algoritmo En el peor caso, la ecuación (8) efectúa K llamadas distintas a la función f con primer argumento $n - 1$. Por tanto, el coste temporal de resolver la función con primer argumento n es

$$t(n) = \begin{cases} c_1 & \text{si } n = 0 \\ Kt(n-1) + c_2 & \text{si } n > 0 \end{cases}$$

lo que conduce, trivialmente a una complejidad $\Theta(K^N)$.

En cambio, el uso de programación dinámica garantiza que la función f sólo se evalúa una vez por cada combinación de argumentos distinta que puede aparecer. En particular, n puede variar entre 1 y N y m_k entre 0 y M . Si los valores w_n son enteros, entonces sólo existen $M + 1$ valores diferentes posibles de m_k y, por tanto, un máximo de $N(M + 1)^K$ elementos. Cada uno de estos se obtiene a partir de otros K , luego la complejidad resultante es $\mathcal{O}(KNM^K)$. Nótese que el problema de la mochila tradicional corresponde al caso particular $K = 1$. Por otro lado, en el caso en que los valores de w_k son reales no es posible establecer una cota mejor que la del método recursivo.

Comentario Volviendo al problema original, si podemos comprobar que los N programas caben en K discos en un tiempo $\mathcal{O}(KNM^K)$, podemos diseñar un procedimiento de búsqueda binaria que obtenga el número mínimo k de discos necesarios para guardar los N programas con un coste global $\mathcal{O}(N^2 \log(N)M^K)$. Se deja este diseño como ejercicio a los lectores.

Ejercicio 13

Ejercicio 14. El problema es trivial si sólo hay una cuba ($n = 1$), ya que entonces bien su capacidad es suficiente para transportar el agua o bien es demasiado pequeña. En este último caso el problema no tiene solución y lo indicaremos con una respuesta desmesurada (∞). Por contra, si $n > 1$, siempre es posible optar entre utilizar la cuba n o no y una fórmula recursiva adecuada es;

$$f(n, m) = \begin{cases} \infty & \text{si } n = 1 \wedge c_1 < m \\ c_1 - m & \text{si } n = 1 \wedge m \leq c_1 \\ \min(f(n-1, m), f(n-1, m - c_n)) & \text{si } n > 1 \wedge c_n < m \\ \min(f(n-1, m), c_n - m) & \text{si } n > 1 \wedge m \leq c_n \end{cases}$$

Una implementación usando programación dinámica iterativa es la siguiente:

```
int pdi ( int N, int M ) {
    int n, m;
    for ( m = 0; m <= M; ++m )
        A[1][m] = c[1] < m ? MAX_INT : m - c[1];
    for ( n = 2; n <= N; ++n )
        for ( m = 0; m <= M; ++m )
            A[n][m] = min ( A[n-1][m],
                           c[n] < m ? A[n-1][ m-c[n] ] : m - c[n] );
    return A[N][M];
}
```

Ejercicio 14

Ejercicio 15. Podemos escribir un programa basado en la siguiente función

$$f(m, n, p) = \begin{cases} 0 & \text{si } mnp = 0 \\ 1 + f(m-1, n-1, p-1) & \text{si } x[m] = y[n] = z[p] \\ \min(f(m-1, n, p), f(m, n-1, p), f(m, n, p-1)) & \text{en otro caso} \end{cases}$$

Ejercicio 15

Ejercicio 16. Tenemos una tabla de probabilidades $p(Y|X)$ de cada bigrama XY que nos dice cuál es la probabilidad de que a la letra X le siga la letra Y . Por otro lado tenemos el siguiente relación ρ que asigna un conjunto de letras a cada número:

1	-
2	ABC
3	DEF
4	GHI
5	JKL
6	MNÑO
7	PQRS
8	TUV
9	WXYZ

donde $-$ representa el espacio en blanco. ¿Cuál es la interpretación más probable de la secuencia $_{625}$?

En principio hay 36 combinaciones de letras que pueden corresponderse con esta secuencia. Una de ellas es, por ejemplo, la palabra MAL. Para calcular la probabilidad de que hayamos querido escribir MAL, tenemos que dividir la probabilidad de esta palabra por el de las 36 restantes. Es decir:

$$p(_MAL_|_{625}) = \frac{p(M|_)p(A|M)p(L|A)p(_|L)}{p(_{625})}$$

Si nos conformamos con obtener la palabra más probable, podemos olvidarnos del factor de normalización $p(_{625})$ ya que este afecta a todas las palabras posibles.

Vamos a escribir como $f(n, x)$ la probabilidad (sin normalizar) de la interpretación más probable de los n primeros dígitos $d_1...d_n$ si el último dígito d_n se interpreta como la letra x . Trivialmente

$$f(1, x) = \begin{cases} 1 & \text{si } x = - \\ 0 & \text{en caso contrario} \end{cases}$$

En general, tenemos

$$f(n, y) = \begin{cases} \max_x f(n-1, x) p(y|x) & \text{si } y \in \rho(d_n) \\ 0 & \text{en caso contrario} \end{cases}$$

Esta fórmula recursiva puede resolverse utilizando programación dinámica.

Ejercicio 16

Ejercicio 17. Sea $f(n, m)$ a una función que vale 1 si el jugador que encuentra n fichas sobre la mesa tras haber retirado el rival m fichas dispone de estrategia ganadora y -1 en caso contrario. Es evidente que $f(0, m) = -1$, es decir, pierde aquél jugador que no encuentra fichas sobre la mesa. Otro caso en el que se pierde inmediatamente es si $n = 1$ y $m = 1$, ya que no es posible realizar ninguna jugada.

En general, el jugador que encuentra n fichas tiene una estrategia ganadora si hay algún movimiento permitido (esto es, puede retirar 1 o más fichas siempre y cuando el número de fichas retirado i sea distinto de m y no supere el máximo permitido M ni el número restante n) que conduce a la derrota del contrario, esto es, $f(n - i, i) = -1$.

En resumen,

$$f(n, m) = \begin{cases} -1 & \text{si } n = 0 \vee n = m = 1 \\ \max_{i \neq m: i \leq M \wedge i \leq n} -f(n - i, i) & \text{en otro caso} \end{cases}$$

La transformación de la fórmula recursiva anterior en un algoritmo de programación dinámica es sencilla.

Ejercicio 17

Ejercicio 18. Dado que no se puede aumentar la velocidad de transmisión enviando información de un nodo a sí mismo, las listas no deben contener elementos repetidos. Por tanto, el dominio de este problema puede definirse como

$$D = \{(x_0, \dots, x_n) \in Q^n : 0 < n < N \wedge x_0 = S \wedge x_n = F \wedge x_i \neq x_j \forall i, j\}$$

Con esta definición, todos los elementos de D son *soluciones* del problema, ya que no hay restricciones adicionales.

Por otro lado, el *objetivo* del problema es maximizar la velocidad de transmisión,

$$\phi(x_0, \dots, x_n) = \min\{v(S, x_1), v(x_1, x_2), \dots, v(x_{n-1}, F)\}$$

Llamaremos, por tanto, *solución óptima* a todo elemento de D que maximiza ϕ .

Árbol de estados. Una forma natural de separar las soluciones en D es considerar el siguiente nodo al que se envía la información. Dado que $x_0 = S$, podemos considerar los casos $x_1 = a, x_1 = b, \dots, x_1 = F$. En el primer caso, tenemos rutas del tipo (S, a, \dots) . En el último caso, el estado es una hoja, ya que no hay más elementos en D que comiencen por $x_0 = S$ y $x_1 = F$ que (S, F) . El proceso puede repetirse recursivamente. Una parte del árbol aparece dibujado en la figura 6.

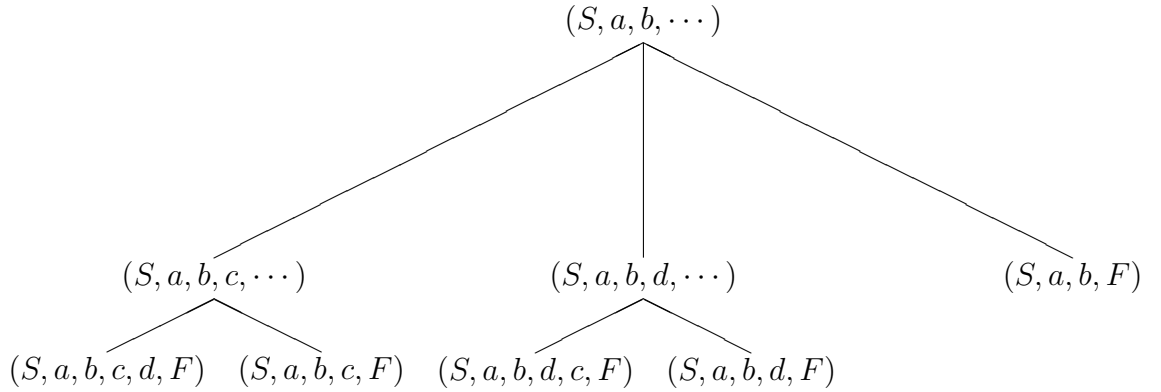


Figura 6: Parte del árbol de llamadas recursivas para un problema de interconexión.

Función de cota optimista. Supongamos que estamos analizando el estado (S, a, b, \dots) . Es evidente que, sea cual sea el resto de la ruta, no podremos alcanzar una velocidad mayor que la que nos permite llegar a b por el camino elegido, esto es $\min\{v(S, a), v(a, b)\}$. Por tanto una cota optimista de la velocidad para el estado $X = (x_0, x_1, \dots, x_n, \dots)$ viene dado por

$$g_1(X) = \min\{v(x_0, x_1), \dots, v(x_{n-1}, x_n)\}$$

que en nuestro ejemplo es $g((S, a, b, \dots)) = 3$. Nótese que si el estado es una hoja, esto es, (S, a, b, \dots, f) entonces g_1 coincide con ϕ .

Sin embargo, es posible afinar un poco más esta cota para los estados internos, ya que no podremos superar la velocidad de la conexión más rápida con x_n aún no utilizada (que en este caso es 2). Es decir, si definimos

$$g_2(X) = \max\{v(x_n, q) : q \neq x_0, \dots, x_n\}$$

podemos tomar $g(X) = \min\{g_1(X), g_2(X)\}$ si X es un estado interno y $g(X) = g_1(X) = \phi(X)$ para las hojas.

Función de cota pesimista Las cotas pesimistas garantizan un valor para la solución óptima. En nuestro ejemplo, las soluciones del tipo (S, a, b, \dots) incluyen como caso particular a (S, a, b, F) cuya velocidad es, en nuestro ejemplo, 2. Por tanto, podemos tomar, siempre que $x_n \neq F$

$$h(X) = \min\{v(x_0, x_1), \dots, v(x_{n-1}, x_n), v(x_n, F)\}$$

En el caso particular de las soluciones del tipo (S, a, b, \dots, F) podemos asumir que $v(F, F) = \infty$ y, con esta convención, aplicar también la fórmula anterior.

Es posible también afinar más la cota h para los estados internos evaluando más soluciones y eligiendo la mejor entre ellas. Se puede, por ejemplo, explorar todos los caminos de longitud 2 además del único c (de longitud 1) que se explora en la h definida anteriormente. También se pueden usar técnicas voraces para buscar una ruta rápida (por ejemplo, enviando la información sucesivamente al ordenador no utilizado que está mejor conectado con el actual hasta que por azar se llega a F). Ejercicio 18

Ejercicio 19. Este problema no puede ser resuelto de forma eficiente mediante programación dinámica, así que diseñaremos un algoritmo de ramificación y poda. Supongamos que los nodos se representan mediante vectores del tipo $(x_1, x_2, \dots, x_m, *, \dots, *)$ donde x_i es 0 si no se ubica una cámara en la intersección i y 1 en caso contrario. Una cota pesimista viene dada por cualquier solución particular contenida en el estado. Es importante darse cuenta de que en algunos casos no existirán soluciones con las opciones x_1, \dots, x_m tomadas. En ese caso, ningún número finito ω garantiza que existe una solución mejor o igual que ω en X y la cota pesimista debe ser, por tanto, $h(X) = \infty$. En caso contrario, nos conviene que $h(X)$ sea lo más baja posible y una forma que permite encontrar resultados con un coste computacional moderado es una estrategia voraz:

1. Sea $u = \sum_{i=1}^m x_i$.
2. Hágase $x_i = 0$ para $i = m + 1, \dots, M$.
3. Sea C el conjunto de calles no vigiladas, esto es, tales que en sus dos extremos $x_i = 0$ o $x_i = *$.
4. Tómesese $n = |C|$.
5. Constrúyase una lista L con las intersecciones $m + 1, \dots, M$ ordenadas de mayor a menor número de calles de C convergentes y hágase $k = 1$.
6. Selecciónese la intersección k en la lista anterior (supongamos que $L_k = i$), hágase $u = u + 1$ y recalcúlese n teniendo en cuenta que $x_i = 1$.
7. Si $n > 0$ y $k < |L|$ vuélvase al paso anterior con $k = k + 1$.
8. Si $n > 0$ devuélvase ∞ . En caso contrario devuélvase u .

Las cotas optimistas deben garantizar que ninguna solución es mejor que el valor g dado. Una opción sencilla consiste en usar el algoritmo anterior, pero modificando el paso 6 de la siguiente forma:

6. Selecciónese la intersección k en la lista anterior (supongamos que $L_k = i$), hágase $u = u + 1$ y $n = n - C_i$.

de manera que n se calcula restando el número de calles que confluyen en i sin tener en cuenta repeticiones.

Ejercicio 19

Ejercicio 20. Supondremos que inicialmente ($t = 0$) todos los vehículos son nuevos y que los vehículos deben mantenerse pasado del mes T . La decisión de renovar o no un vehículo debe tener en cuenta dos efectos contrapuestos:

1. por un lado conviene alargar al máximo la antigüedad del vehículo;
2. por otro, conviene aprovechar los momentos de precios bajos.

Es evidente que

$$g(x_1, \dots, x_t, *, \dots, *) = \sum_{k=1}^t w_k x_k$$

es una cota optimista del gasto que hay que realizar. Sin embargo, dado que es preciso realizar algunas renovaciones en el tiempo restante se puede proceder de la siguiente manera:

1. Tómesese $g = \sum_{k=1}^t w_k x_k$.
2. Sea i el último índice tal que $x_i = 1$ (última renovación).
3. Calcúlese el número mínimo de renovaciones necesarias en el tiempo restante: $\mu = \lfloor \frac{T-i}{N} \rfloor$.
4. Calcúlese los μ valores menores entre w_{t+1}, \dots, w_T (el coste de esta búsqueda es lineal).
5. Súmense dichos valores a g .

Una función de cota pesimista se puede obtener tomando cualquier decisión de las posibles. Una que no conducirá a resultados muy malos es prolongar al máximo la vida de los vehículos, esto es,

1. Hágase $h = \sum_{k=1}^t w_k x_k$.
2. Tómesese i , el último índice tal que $x_i = 1$ (última renovación).
3. Mientras $i + M \leq T$ hágase $i = i + M$ y súmese w_i a h .

En ambas funciones de cota hemos supuesto que x_1, \dots, x_t es una solución para los primeros t meses. En caso contrario, tanto g como h deben retornar un valor infinito.

Ejercicio 20

Ejercicio 21. Una cota optimista es $g(Z) = \max(g_1(Z), g_2(Z))$ donde

$$g_1(Z) = \max_{k=1}^m z_k$$

es la longitud de la línea más larga hasta el momento y

$$g_2(Z) = \frac{L - L_m}{M - m}$$

con $L = \sum_{k=1}^N s_k$ (longitud total del texto) y $L_m = \sum_{k=1}^m z_m$ (longitud del texto ya ubicado) es la longitud mínima en caso de un reparto óptimo (difícilmente alcanzable, porque no está permitido romper palabras).

Por otro lado, una cota pesimista puede obtenerse mediante el siguiente procedimiento:

1. Calcúlese $h = g_1(Z)$ y $\mu = g_2(Z)$ y hágase $i = m$.
2. Mientras $i < M$, hágase $i = i + 1$ y elíjase el menor $z_i \geq \mu$ posible tal que $\sum_{k=1}^i z_k = \sum_{k=1}^n s_k$ para algún n (es decir, coincida con algún límite de palabra).
3. Devuélvase $\max_m \{z_m\}$.

Ejercicio 21

Ejercicio 22. No es una función de cota pesimista porque no garantiza que exista una solución con ese valor: las soluciones deben satisfacer $\sum x_i w_i = M$ y no basta con que $\sum x_i w_i \leq M$.

Tampoco es una cota optimista, ya que ésta se consigue considerando los objetos (o sus fracciones) de mayor valor específico (v_i dividido por w_i) y no simplemente de mayor valor.

Ejercicio 22

Ejercicio 23. Una función de cota pesimista para $(x_1, x_2, \dots, x_n, *, \dots, *)$ consiste en calcular la función objetivo

$$\Phi(z) = \max\left(\sum_{i:z_i=1} t_{i1}, \sum_{i:z_i=2} t_{i2}\right)$$

para una solución (z_1, \dots, z_N) en la que $z_i = x_i$ si $i \leq n$ y las demás componentes se eligen con una estrategia que permita obtener soluciones razonablemente satisfactorias. Por ejemplo:

Mientras i sea menor que N , hágase $i = i + 1$ y $z_i = 1$ si $\sum_{i < k: z_i=1} t_{i1} < \sum_{i < k: z_i=2} t_{i2}$ y $z_i = 2$ en caso contrario.

Esta estrategia permite asignar cada trabajo a la máquina menos ocupada, pero no garantiza que la solución obtenida sea óptima.

En cambio, una cota optimista debe proporcionar un valor que ninguna solución pueda mejorar. Por ejemplo, como el trabajo i consumirá un tiempo que será, al menos, $\min(t_{i1}, t_{i2})$ es imposible que se realicen todos los trabajos en un tiempo menor que si se reparten estos tiempos entre ambas máquinas para acabar simultáneamente:

$$\frac{1}{2} \left(\sum_{i=1}^n t_{iz_i} + \sum_{i=n+1}^N \min(t_{i1}, t_{i2}) \right)$$

Ejercicio 23

Ejercicio 24. Una forma de calcularlo es la siguiente:

```
void simula ( int N ) {  
    double S, S2, t;  
    S = S2 = 0;  
    for ( int n = 1; n <= N; ++n ) {  
        t = coste ();  
        S += t;  
        S2 += t * t;  
    }  
    // Your code  
    std::cout << S / N << " +- "  
               << std::sqrt ( S2 - (S*S) / N ) / N;  
}
```

Ejercicio 24

Ejercicio 25. Si dividimos el círculo en dos coronas: los puntos situados a menos de $r/2$ del centro y el resto, este procedimiento distribuye la mitad de los puntos en cada corona, cuando es fácil comprobar que el área de la segunda es tres veces la de la primera. Por tanto, el procedimiento no es correcto. Una forma de conseguir una distribución homogénea es aplicar la transformación $r = R\sqrt{\zeta_1}$.

Ejercicio 25

Ejercicio 26. Dado que para generar la distribución $f(x)$ usamos

$$x = F^{-1}(\zeta) = \sqrt{1 - \sqrt{1 - \zeta}}$$

entonces,

$$F(x) = \zeta = 1 - (1 - x^2)^2$$

Y, por definición,

$$f(x) = F'(x) = 4x(1 - x^2)$$

Ejercicio 26