

Ejercicios resueltos de programación 3

Tema 6. Algoritmos voraces.

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

1. Introducción teórica	3
2. Cuestiones de exámenes	5
3. Problemas de exámenes solucionados	36
4. Problemas de exámenes sin solución o planteados	51

Introducción teórica:

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

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

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

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

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

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

Las funciones empleadas más destacadas de este esquema son:

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

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

- Inicialmente, el conjunto de elementos seleccionados está vacío y el de solución también lo está.
- En cada paso, se considera añadir a este conjunto el mejor candidato sin considerar los restantes, estando guiada nuestra selección por la **función de selección**. Se nos darán estos casos:
 1. Si el conjunto ampliado de candidatos seleccionados ya no fuera factible (no podemos completar el conjunto de solución parcial dado por el momento), **rechazamos** el candidato considerado por el momento y no lo volvemos a considerar.
 2. Si el conjunto aumentado sigue siendo factible, entonces **añadimos** el candidato actual al conjunto de candidatos seleccionados. Cada vez que se amplía el conjunto de candidatos seleccionados comprobamos si este constituye una solución para nuestro problema. Se quedará en ese conjunto para siempre.

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

El **esquema voraz** es el siguiente:

```

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

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

1ª parte. Cuestiones de exámenes:

Febrero 2000-1ª (ejercicio 3)

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

Respuesta:

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

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

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

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

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

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

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

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

Septiembre 2000-reserva (ejercicio 2)

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

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

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

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

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

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

Febrero 2001-2ª (ejercicio 1)

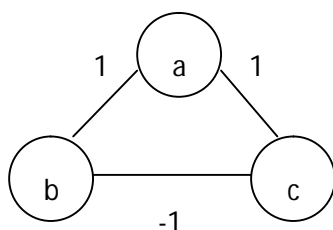
Enunciado: Mostrar mediante un ejemplo que el algoritmo de Dijkstra no siempre encuentra el camino mínimo si existe una distancia negativa.

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

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

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

1^{er} ejemplo: Se nos da este grafo:

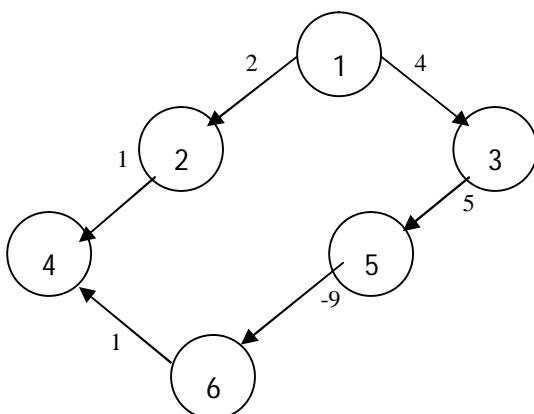


Siguiendo el algoritmo de Dijkstra empezando por el nodo a, tendremos:

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

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

2º ejemplo: Se nos da este grafo:



Siguiendo paso a paso el algoritmo de Dijkstra tendremos:

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

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

Septiembre 2001 (ejercicio 1)

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

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

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

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

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

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

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

Si se produce la eliminación de la raíz del montículo (el bucle se ejecuta $n - 2$ veces) y hay que flotar un máximo de a nodos (siendo a el número de aristas) obtenemos un tiempo total de $\theta((a + n) * \log(n))$.

Si el grafo es conexo, es decir, $a \geq n - 1$, el tiempo total es $\theta(n * \log(n))$.

Si el grafo es denso será preferible la implementación con matrices, si es disperso es mejor la implementación con montículos.

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

Septiembre 2001 (ejercicio 2)

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

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

Septiembre 2001 (ejercicio 3)

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

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

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

En resumen:

n : Número de tareas de tiempo unitario. Por ejemplo, una hora, días,...

$T = 1, 2, \dots, n$ En cada instante solo podemos realizar una tarea.

g_i : Beneficio asociado a la tarea i .

d_i : Plazo máximo de la tarea i .

El problema consiste en maximizar el beneficio total.

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

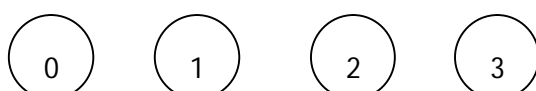
i	1	2	3	4
g_i	20	15	10	7
d_i	3	1	1	3

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

Los pasos son los siguientes:

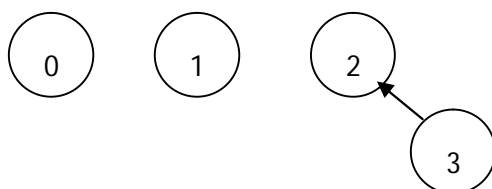
Inicialmente: $p = \min(n, \max(d_i)) = \min(4, 3) = 3$.

Por tanto, como máximo tendremos una planificación de 3 tareas:



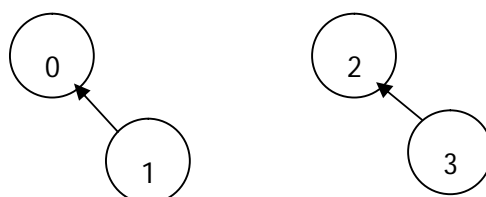
Primer intento: $d_1 = 3$. Se asigna la tarea 1 a la posición 3.

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



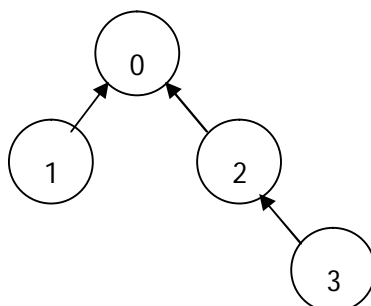
Segundo intento: $d_2 = 1$. Se asigna la tarea 2 a la posición 1.

$F(K) = 1, F(L) = F(K) - 1 = 0$. Fusionamos K con L



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

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



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

Febrero 2002-1ª (ejercicio 1)

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

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

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

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

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

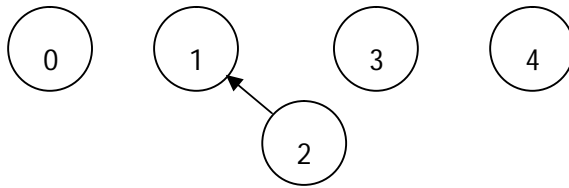
Emplearemos la técnica en la que resolveremos más rápidamente el problema. Para ello, usaremos estructura de partición y haremos lo siguiente:

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

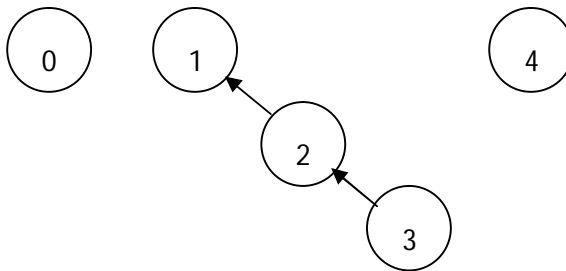
Por tanto, como máximo tendremos una planificación de 4 tareas:



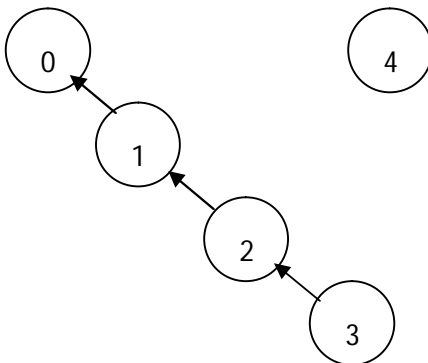
Primer intento: $d_1 = 2$. Se asigna la tarea 1 a la posición 2.
 $F(K) = 2, F(L) = F(K) - 1 = 1$. Fusionamos K con L



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



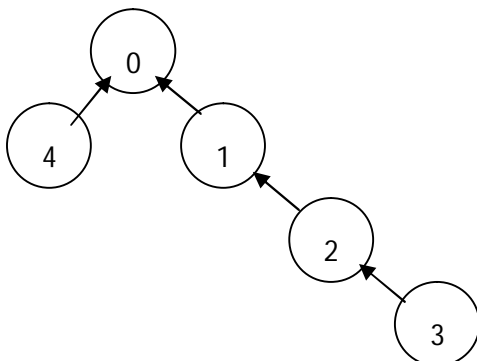
Tercer intento: $d_3 = 1$. Se asigna la tarea 3 a la posición 1.
 $F(K) = 1, F(L) = F(K) - 1 = 0$. Fusionamos K con L



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

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

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



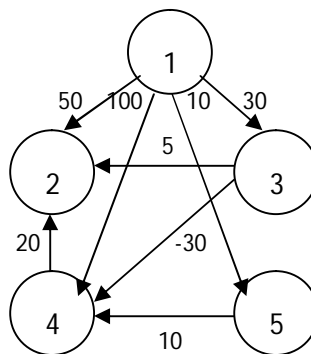
Septiembre 2002 (ejercicio 2)

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

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

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

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



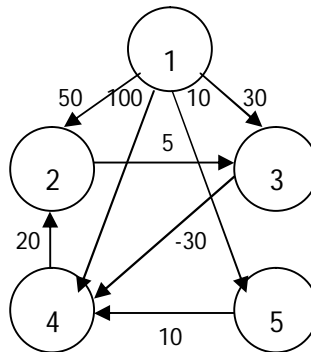
Realizaremos estos pasos para hallar los caminos mínimos, empleando para ello el algoritmo de Dijkstra:

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

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

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

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



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

Febrero 2003-1ª (ejercicio 1)

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

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

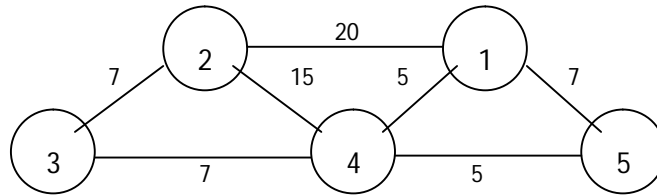
Emplearemos **montículos invertidos o de mínimos** (la raíz es el menor elemento) para ordenar las aristas de menor coste. El coste de esta implementación usando montículos será:

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

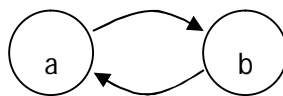
Por tanto, tendremos que el coste del algoritmo será más ventajoso usando montículos cuando el grafo es disperso, es decir, cuando a tiende a n , siendo el coste $\theta(n * \log(n))$. En tales casos, el algoritmo original desperdicia el tiempo ordenando estas aristas inútiles, aunque asintóticamente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

Febrero 2003-1ª (ejercicio 2)

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



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



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

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

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

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

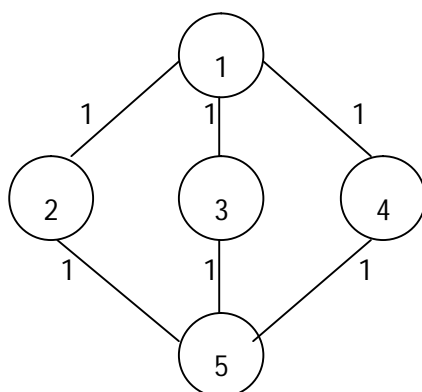
Enunciado: ¿Puede un grafo tener dos árboles de recubrimiento mínimo diferentes? En caso afirmativo, poner un ejemplo. En caso negativo, justificar la respuesta.

Respuesta:

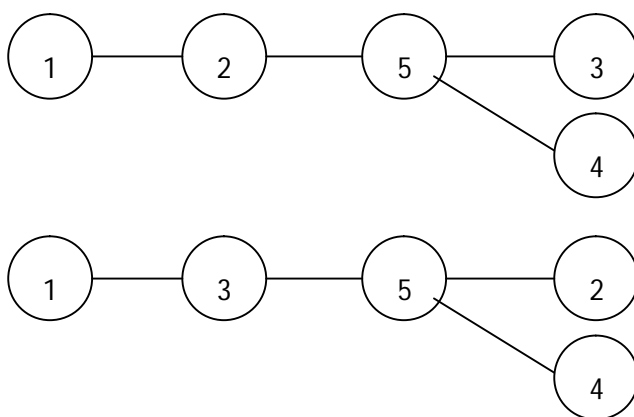
Podemos decir que un grafo sí que puede tener dos árboles de recubrimiento mínimo diferentes.

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

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



Tendremos dos árboles de recubrimiento mínimo:



Febrero 2003-2ª (ejercicio 3)

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

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

Respuesta:

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

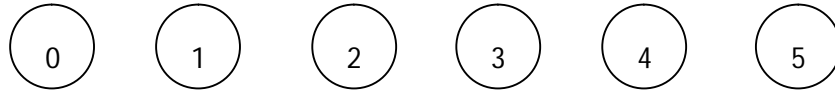
i	1	2	3	4	5	6	7	8
a_i	8	5	1	4	6	2	3	7
g_i	30	25	20	15	15	10	7	5
d_i	2	2	4	1	3	5	1	1

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

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

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

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



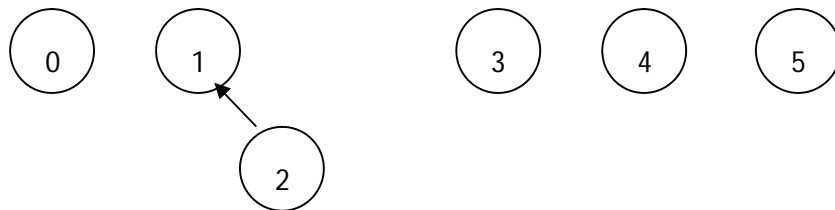
Como añadido pondremos la matriz j , donde reflejaríamos la **planificación parcial** paso a paso de las tareas con respecto a los plazos, que sería:

	1	2	3	4	5
j	0	0	0	0	0

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

Primer intento: $d_1 = 2$. Se asigna la tarea 1 a la posición 2.

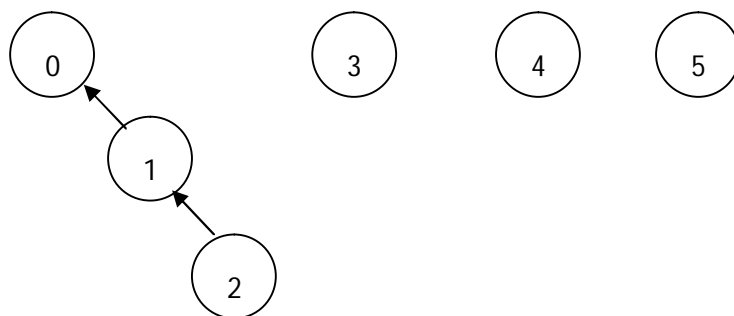
$F(K) = 2, F(L) = F(K) - 1 = 1$. Fusionamos K con L



	1	2	3	4	5
j	0	1	0	0	0

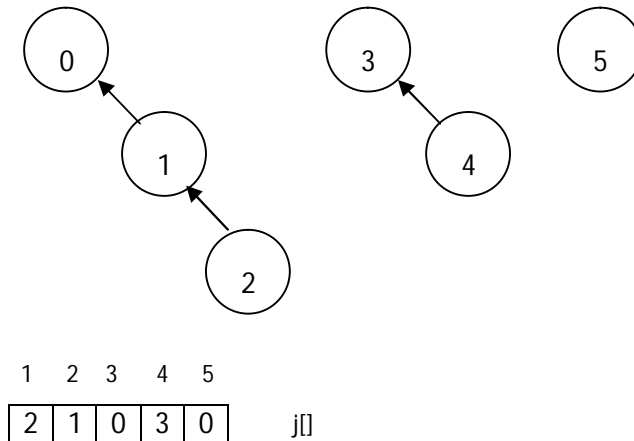
Segundo intento: $d_2 = 2$. Se asigna la tarea 2 a la posición 2.

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



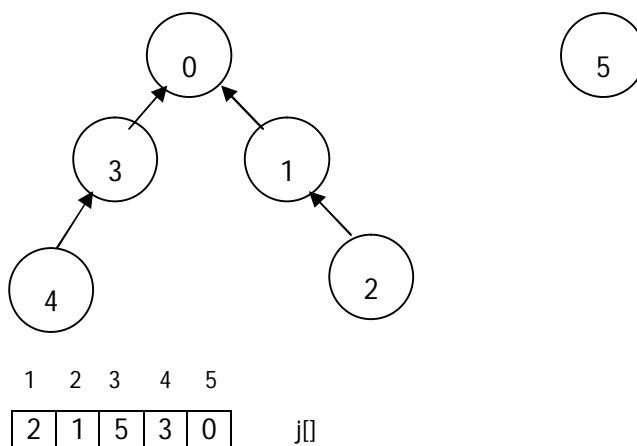
	1	2	3	4	5
j	2	1	0	0	0

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



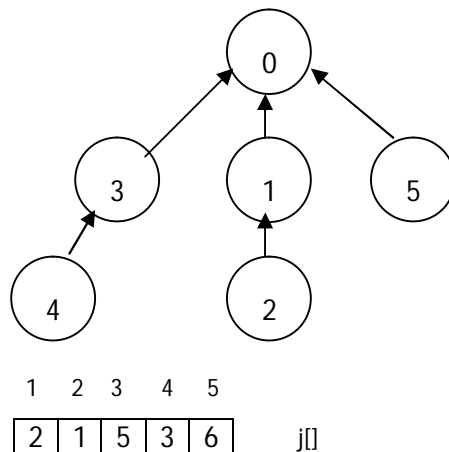
Cuarto intento: $d_4 = 1$. Se rechaza la tarea, porque no hay posiciones libres disponibles.

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



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

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



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

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

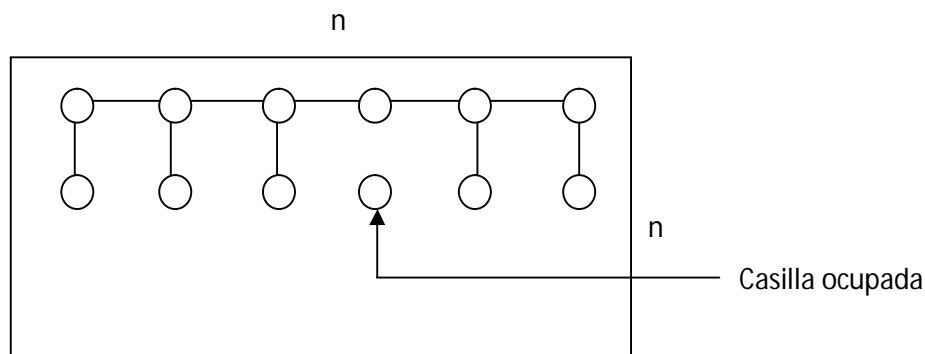
Queda decir que las tareas que salen en la solución son las ordenadas, con lo que habría que hacer el cambio con respecto a a_i , que serían las tareas reales. Según dicha solución el orden de ejecución de las tareas sería: a_5, a_8, a_6, a_1, a_2 , que realmente serían el orden de las tareas tras hacer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

Septiembre 2003 (ejercicio 3)

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

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

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



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

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

Septiembre 2003-reserva (ejercicio 2)

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

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

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

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

Febrero 2004-1ª (ejercicio 3)

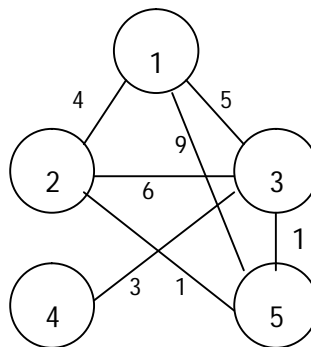
Enunciado: Resolver mediante el algoritmo de Kruskal el árbol de expansión mínimo del grafo definido por la siguiente matriz de adyacencia:

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

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

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

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



Tendremos estos **pasos**:

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

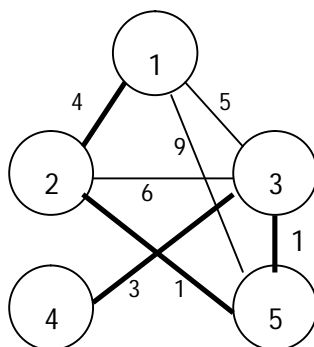
Nodo	Coste
{3,5}	1
{2,5}	1
{3,4}	3
{1,2}	4
{1,3}	5
{2,3}	6
{1,5}	9

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

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

Proceso terminado, ya no queda ninguna componente no conexas.

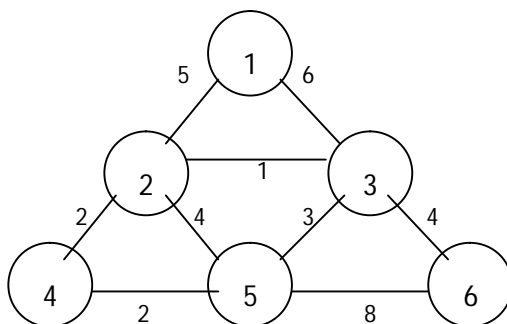
Para que se vea de modo mejor resaltaremos las aristas que forman el **árbol de recubrimiento mínimo** (será el primer y último ejercicio en el que lo hagamos):



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

Febrero 2004-2ª (ejercicio 2)

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



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

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

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

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

Proceso terminado. No queda ninguna componente no conexa.

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

Septiembre 2004 (ejercicio 3)

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

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

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

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

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

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

Septiembre 2004-reserva (ejercicio 1)

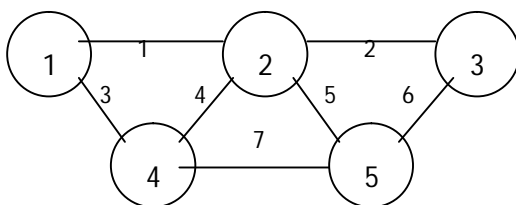
Enunciado: Defina que es un árbol de recubrimiento mínimo, explica alguno de los algoritmos propuestos en la asignatura para hallarlos, explicando paso a paso un ejemplo.

Respuesta: La solución está hecha completamente por un alumno.

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

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

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



Ordenamos las aristas de menor a mayor coste:

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

Los pasos nos quedarían así:

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

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

Septiembre 2004-reserva (ejercicio 3)

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

Respuesta:

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

Se nos da estos productos con $PMAX = 100$

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

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

$$w = 51.$$

$$v = 150.$$

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

$$w = 100.$$

$$v = 200.$$

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

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

Febrero 2005-1ª (ejercicio 1)

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

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

Respuesta:

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

Para la **demostración**, tendríamos lo siguiente:

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

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

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

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

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

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

- b) El algoritmo voraz no obtendría la solución óptima en todos los casos.

Contraejemplo: Supongamos la siguiente lista de ficheros con tamaños asociados:

Fichero	F_1	F_2	F_3
Tamaño	40	30	15

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

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

Febrero 2005-1ª (ejercicio 2)

Enunciado: Exponga y explique el algoritmo más eficiente que conozca para realizar una planificación de tareas de plazo fijo maximizando el beneficio.

Dada la tabla adjunta de tareas con sus beneficios (g_i) y caducidades (d_i), aplique paso a paso el algoritmo propuesto, suponiendo que se desea realizar una planificación con tiempo $t = 5$

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

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

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

```

funcion secuencia2 ( $d[1..n]$ ):  $k$ , matriz  $[1..k]$ 
    matriz  $j$ ,  $F[0..n]$ 
    { Iniciación }
     $p = \min(n, \max\{d[i] | 1 \leq i \leq n\})$ ;
    para  $i \leftarrow 0$  hasta  $p$  hacer  $j[i] \leftarrow 0$ 
                                 $F[i] \leftarrow i$ 
                                Iniciar el conjunto  $\{i\}$ 

    { Bucle voraz }
    para  $i \leftarrow 1$  hasta  $n$  hacer    { Orden decreciente de  $g$  }
         $k \leftarrow \text{buscar}(\min(p, d[i]))$ 
         $m \leftarrow F[k]$ 
        si  $m \neq 0$  entonces
             $j[m] \leftarrow i$ ;
             $l \leftarrow \text{buscar}(m - 1)$ 
             $F[k] \leftarrow F[l]$ 
            { El conjunto resultante tiene la etiqueta  $k$  o  $l$  }
            fusionar ( $k, l$ )
    { Sólo queda comprimir la solución }
     $k \leftarrow 0$ 
    para  $i \leftarrow 1$  hasta  $p$  hacer
        si  $j[i] > 0$  entonces  $k \leftarrow k + 1$ 
                                 $j[k] \leftarrow j[i]$ 

    devolver  $k, j[1..k]$ 

```

El **algoritmo** será el siguiente:

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

$$p = \min(n, \max(d_i)).$$

→ Mayor de los plazos

→ Número de tareas

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

- ii. Adición de una tarea con plazo d : se busca un conjunto que contenga a d ; sea K este conjunto. Si $F(K) = 0$ se rechaza la tarea, en caso contrario:
- Se asigna la nueva tarea a la posición $F(K)$.
 - Se busca el conjunto que contenga $F(K) - 1$. Llamemos L a este conjunto (no puede ser igual a K).
 - Se fusionan K y L . El valor de F para este nuevo conjunto es el valor viejo de $F(L)$.

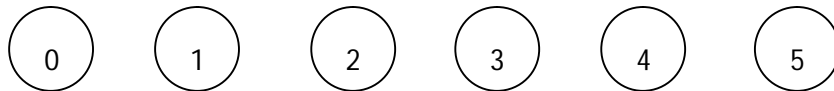
b) Como en ejercicios anteriores seguiremos estos pasos:

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

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

Veremos paso a paso el algoritmo como sigue:

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



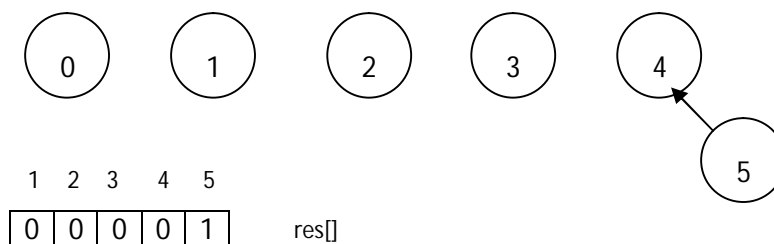
Ponemos de nuevo la matriz de resultados, donde reflejaríamos la planificación parcial de las tareas con respecto a los plazos (las tareas reales), que sería:

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

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

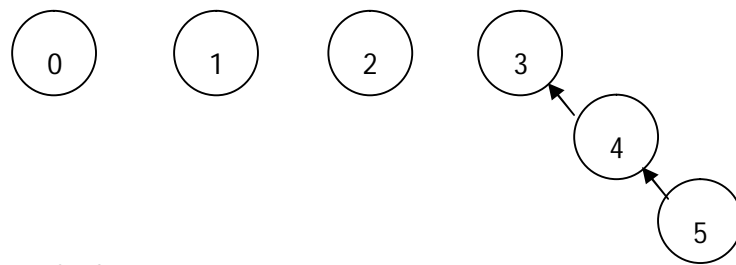
Vamos escogiendo cada una de las tareas por el orden antes puesto e incluyéndolos en su correspondiente de partición. Esta operación implica fusionar las estructuras en la cual se ha incluido la tarea con la estructura de partición inmediatamente inferior. Escogeremos, en este caso (como en anteriores),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

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



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

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

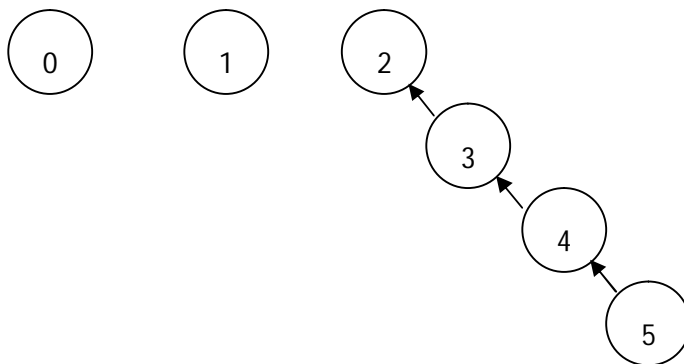


1	2	3	4	5
0	0	0	2	1

res[]

Tercer intento: $d_3 = 4$. Se asigna la tarea 3 a la posición 4.

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

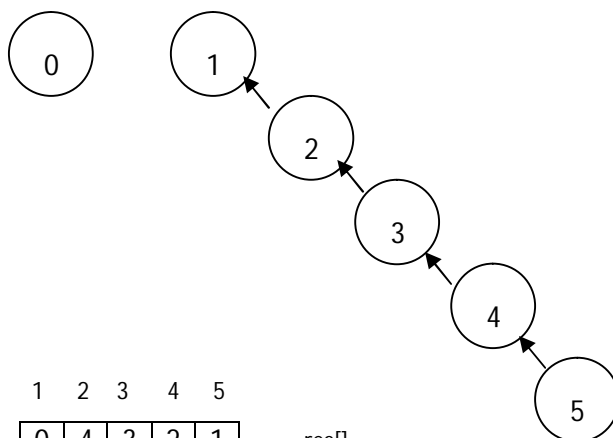


1	2	3	4	5
0	0	3	2	1

res[]

Cuarto intento: $d_4 = 2$. Se asigna la tarea 4 a la posición 2.

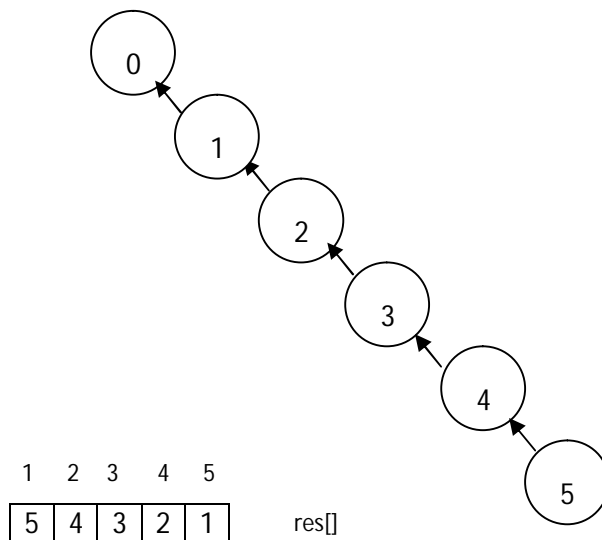
$F(K) = 2, F(L) = F(K) - 1 = 1$. Fusionamos K con L



1	2	3	4	5
0	4	3	2	1

res[]

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



El algoritmo termina cuando ya no queda ninguna estructura de partición libre para asignar tareas.

Por tanto, haciendo el cambio de i a a_i quedaría el resultado total así:

1	2	3	4	5
5	4	1	9	8

res[]

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

Febrero 2005-2ª (ejercicio 3)

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

Respuesta:

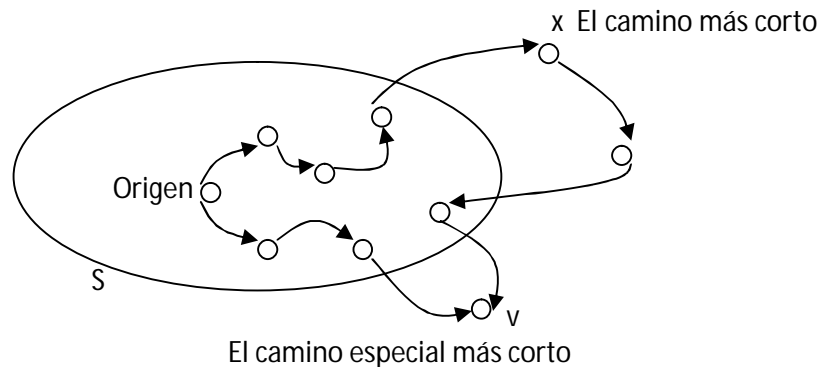
Esta solución es totalmente dada en el libro, y hecho en el resumen del tema. Demostraremos por inducción matemática que:

- Si un nodo $i \neq 1$ está en S , entonces $D[i]$ da la longitud del camino más corto desde el origen hasta i , y
 - Si un nodo i no está en S , entonces $D[i]$ da la longitud del camino *especial* más corto desde el origen hasta i .
- Base:** Inicialmente, solo el nodo 1, que es el origen, se encuentra en S , así que la situación a) es cierta sin más demostración. Para los demás nodos, el único camino especial desde el origen es el camino directo y D recibe valores iniciales en consecuencia. Por tanto, la situación b) es también cierta cuando comienza el algoritmo.
 - Hipótesis de inducción:** La hipótesis de inducción es que tanto la situación a) como la b) son válidas inmediatamente antes de añadir un nodo v a S (conjunto de nodos seleccionados). Detallamos los pasos de inducción por separado para ambas situaciones.

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

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

Gráficamente, sería:



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

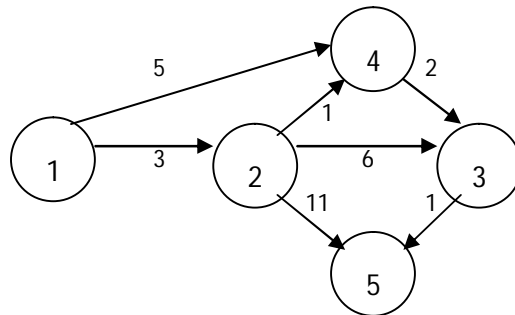
En el **segundo caso**, sea x el último nodo de S visitado antes de llegar a w . La longitud de este camino es $D[x] + L[x, w]$. Parece a primera vista que para calcular el nuevo valor de $D[w]$ deberíamos comparar el valor anterior de $D[w]$ con $D[x] + L[x, w]$ para todo nodo x de S (incluyendo a v).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

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

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

Febrero 2006-1ª (ejercicio 3)

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



Respuesta:

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

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

Febrero 2006-2ª (ejercicio 2)

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

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

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

Respuesta:

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

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

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

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

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

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

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

Respuesta:

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

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

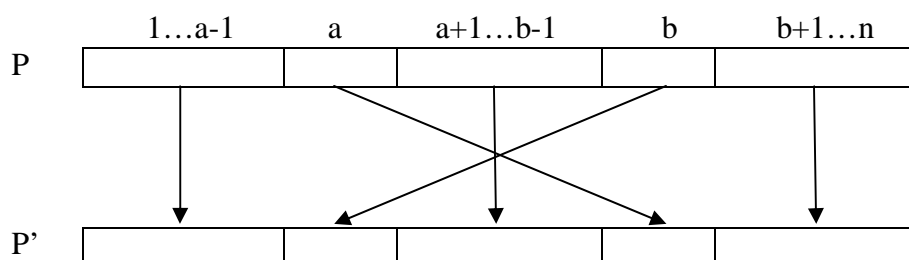
Sea $P = p_1 p_2 \dots p_n$ cualquier permutación de enteros del 1 al n y sea $s_i = t_{p_i}$. Si se sirven clientes en el orden P , entonces el tiempo requerido por el j -ésimo cliente que haya que servir será s_j y el tiempo transcurrido en el sistema por todos los clientes es:

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

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

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

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



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

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

La nueva planificación es preferible a la vieja, porque:

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

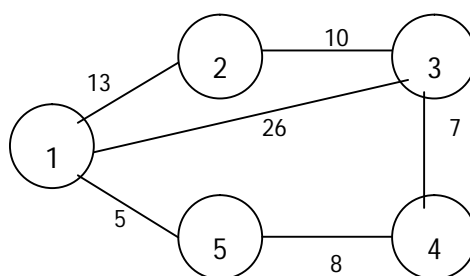
Se observa tras el intercambio que los clientes salen en su posición adecuada, ya que $s_b < s_a$ por nuestra suposición inicial, estando el resto ordenados. Por tanto, P' es mejor que P en conjunto.

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

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

Diciembre 2006 (ejercicio 3)

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



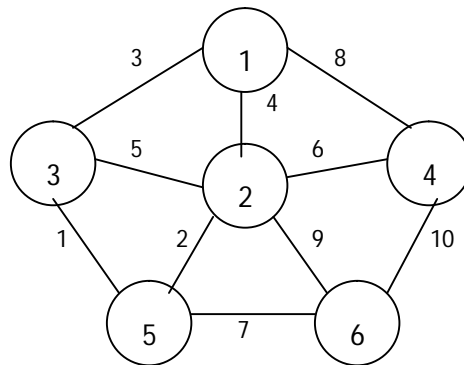
Respuesta: Estos ejercicios se hacen exactamente igual a los que hemos visto previamente, así que pondremos el cuadro con los datos que nos solicitan:

Paso	Nodos seleccionados	Nodos no seleccionados	Vector de distancias	Vector de predecesores
Inicialización	{1}	{2,3,4,5}	^{2 3 4 5} [13,26,∞,5]	[1,1,—,1]
1	{1,5}	{2,3,4}	[13,26,13,5]	[1,1,5,1]
2	{1,2,5}	{3,4}	[13,23,13,5]	[1,2,5,1]
3	{1,2,4,5}	{3}	[13,20,13,5]	[1,4,5,1]

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

Febrero 2007-2ª (ejercicio 3)

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



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

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

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

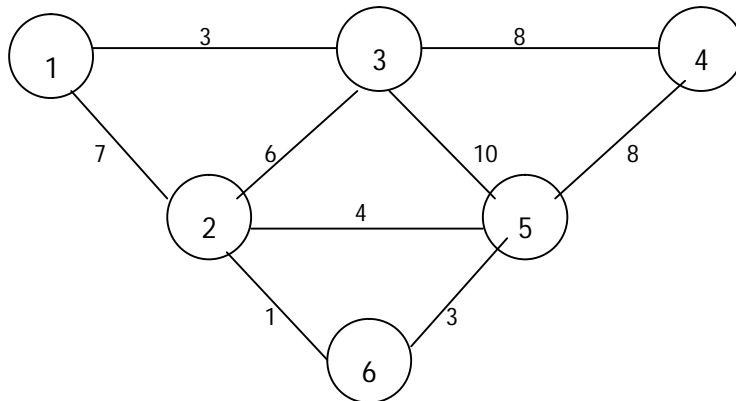
Siguiendo el algoritmo será:

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

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

Septiembre 2008-reserva (ejercicio 3)

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



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

2ª parte. Problemas de exámenes solucionados:

Tendremos en cuenta los pasos a seguir para resolver problemas:

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

Antes de ver los ejercicios decir que el esquema nos basaremos en los del libro de teoría, de Brassard, que como diferencia más importante es que existe una función que es *factible*, mientras que en el libro de práctica es *completable*. Ambas hacen la misma función, pero nos seguiremos basando en el libro de teoría.

Hay ejercicios que hemos modificado, sobre todo el código del algoritmo completo, ya que no cuadran la solución dada, por lo que lo especificaremos cuando llegue ese caso en el ejercicio.

Al ser problemas con voraces, tendremos que realizar una demostración de optimalidad y si existe lo pondremos en la elección el esquema, por ser más lógico ponerlo así (lo estimo yo).

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

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

Respuesta:

1. Elección razonada del esquema

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

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

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

2. Esquema general e identificación con el problema

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

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

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

```
funcion voraz (C: Conjunto): conjunto
{ C es el conjunto de candidatos }
 $S \leftarrow \emptyset$       { Construimos la solución en el conjunto S }
mientras  $C \neq \emptyset$  y  $\neg$ solución (S) hacer
     $x \leftarrow \text{seleccionar}(C)$ 
     $C \leftarrow C \setminus \{x\}$ ;
    si factible ( $S \cup \{x\}$ ) entonces  $S \leftarrow S \cup \{x\}$ 
si solución (S) entonces devolver S
si no devolver “no hay solución”
```

El esquema general para el libro de problemas es:

```
fun voraz (C: Conjunto) dev (S:conjunto)
 $S \leftarrow \emptyset$ 
mientras  $\neg$ solución (S) y  $C \neq \emptyset$  hacer
     $x \leftarrow \text{elemento que maximiza objetivo}(x)$ 
     $C \leftarrow C \setminus \{x\}$ ;
    si completable ( $S \cup \{x\}$ ) entonces
         $S \leftarrow S \cup \{x\}$ ;
    fsi
fmientras
dev S
ffun
```

donde:

C: Conjunto de vértices del grafo.

S: Recubrimiento del grafo.

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

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

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

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

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

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

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

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

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

3. Estructuras de datos

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

grafo = vértice [1..N] de vértice

vertice = tupla

 índice: entero

 // Posición en el vector

 grado: entero

 // Grado del vértice

 adyacentes: apuntador a nodo_adyacente

nodo_adyacente = tupla

 adyacente: entero

 siguiente: apuntador a nodo_adyacente

4. Algoritmo completo a partir del refinamiento del esquema

Adaptamos el algoritmo general a lo dicho anteriormente:

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

Las funciones son las siguientes:

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

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

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

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

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

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

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

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

5. Análisis del coste

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

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

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

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

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

Repuesta:

1. Elección razonada del esquema algorítmico

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

CUIDADO: No hay que confundir este problema (de "*caminos mínimos*") con el problema de dar un *árbol de expansión mínimo*, que resuelven algoritmos como el de Prim o Kruskal. En este caso, un árbol de expansión mínimo 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.

2. Descripción del esquema usado e identificación con el problema

Tendremos el esquema general voraz:

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

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

El algoritmo de Dijkstra opera como sigue:

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

El algoritmo es el siguiente:

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

3. Estructuras de datos

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

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

4. Algoritmo completo a partir del refinamiento del esquema general

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

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

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

5. Estudio del coste

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

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

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

Respuesta:

1. Elección razonada del esquema algorítmico

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

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

2. Descripción del esquema usado e identificación del problema

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

En notación algorítmica puede describirse así:

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

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

```

fun prim ( $G = \langle N, A \rangle$ : grafo, longitud:  $A \rightarrow R^+$ ): conj. aristas
   $S \leftarrow \emptyset$ ;
   $B \leftarrow$  un elemento arbitrario de  $N$ 
  mientras  $B \neq N$  hacer
    buscar  $e = \{u, v\}$  de longitud mínima tal que  $u \in B$  y  $v \in N/B$ 
     $S \leftarrow S \cup \{e\}$ ;
     $B \leftarrow B \cup \{v\}$ ;
  fmientras
  devolver  $T$ 
ffun

```

3. Estructuras de datos

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

4. Algoritmo completo

El algoritmo de Prim no necesita ninguna modificación posterior.

5. Estudio del coste

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

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

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

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

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

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

Respuesta:

1. Elección razonada del esquema algorítmico

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

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

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

Demostración de optimalidad:

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

2. Descripción del esquema usado e identificación con el problema

El esquema voraz es:

```

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

```

Hemos de particularizar las siguientes funciones:

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

3. Estructura de datos

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

4. Algoritmo completo a partir del refinamiento del esquema general

Retocando el **esquema general** tenemos:

```

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

```

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

La única función (de selección) por desarrollar es 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      // Escoge la de menor tamaño
    fsi
    fpara
    dev c[min]
ffun

```

5. Estudio del coste

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

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

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

Respuesta:

1. Elección razonada del esquema algorítmico

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

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

donde la constante k no afecta al resultado.

El tiempo medio de acceso es:

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

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

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

Demostración de optimalidad

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

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

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

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

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

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

2. Descripción del esquema usado e identificación con el problema

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

En notación algorítmica:

funcion voraz (C: Conjunto): conjunto

{ C es el conjunto de candidatos }

$S \leftarrow \emptyset$ { Construimos la solución en el conjunto S }

mientras $C \neq \emptyset$ y \neg solución (S) hacer

$x \leftarrow seleccionar(C)$

$C \leftarrow C \setminus \{x\};$

si factible ($S \cup \{x\}$) entonces $S \leftarrow S \cup \{x\}$

si solución (S) entonces devolver S

si no devolver “no hay solución”

En este caso:

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

3. Estructuras de datos

Hay dos posibles representaciones:

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

tipo programa = tupla
 identificador: entero
 longitud: entero

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

4. Algoritmo completo a partir del refinamiento del esquema general

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

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

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

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

5. Estudio del coste

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

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

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

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

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

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

Septiembre 2003 (problema)

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

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

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

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

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

1. Elección razonada del esquema algorítmico

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

2. Descripción del esquema usado e identificación con el problema

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

```
funcion prim ( $G = \langle N, A \rangle$ : grafo, longitud:  $A \rightarrow R^+$ ): conj. aristas
{ Iniciación }
 $T \leftarrow \emptyset$ ;
 $B \leftarrow \{ \text{un miembro arbitrario de } N \}$ 
mientras  $B \neq N$  hacer
    buscar  $e = \{u, v\}$  de longitud mínima tal que  $u \in B$  y  $v \in N/B$ 
     $T \leftarrow T \cup \{e\}$ ;
     $B \leftarrow B \cup \{v\}$ ;
devolver  $T$ 
```


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

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

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

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

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

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

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

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

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

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

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

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

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

Coste del árbol de expansión mínimo: 4

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

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

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

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

3. Estructuras de datos

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

4. Algoritmo completo a partir del esquema general

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

```
funcion prim (L[1..n,1..n]): conj. aristas
{ Iniciación: solo el nodo 1 se encuentra en B }
 $T \leftarrow \emptyset$ ; { Contendrá las aristas del árbol de recubrim. mínimo }
para  $i \leftarrow 2$  hasta  $n$  hacer
    mas próximo  $[i] \leftarrow 1$ 
     $distmin[i] \leftarrow L[i, 1]$ 
{ Bucle voraz }
repetir  $n - 1$  veces
     $min \leftarrow \infty$ 
    para  $j \leftarrow 2$  hasta  $n$  hacer
        si  $0 \leq distmin[j] < min$  entonces  $min \leftarrow distmin[j]$ 
                                          $k \leftarrow j$ 
     $T \leftarrow T \cup \{ \text{más próximo } [k], k \}$ 
     $distmin[k] \leftarrow -1$ 
    para  $j \leftarrow 2$  hasta  $n$  hacer
        si  $L[j, k] \leq distmin[j]$  entonces  $distmin[k] \leftarrow L[j, k]$ 
                                         más próximo  $[j] \leftarrow k$ 
devolver T
```

5. Coste del algoritmo

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

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

Febrero 1997-2ª (problema 1)

Enunciado: Queremos grabar n canciones de duraciones t_1, t_2, \dots, t_n en una cinta de audio de duración $T < \sum_{i=1}^n t_i$.

Diseñar un algoritmo que permita almacenar el máximo número de canciones en el espacio disponible.

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

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

Febrero 1998-1ª (problema 1)

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

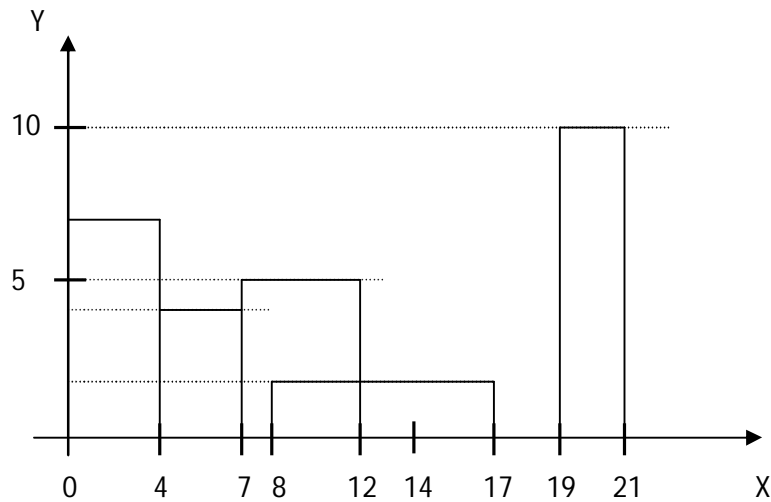
Respuesta: Este ejercicio nos está hablando del metro de Japón, considerando este como un grafo, en el que hay que seleccionar las aristas más cortas. Por ello y sin ánimo de equivocarme estimo que corresponde con el algoritmo de Dijkstra, en el que se calcula la distancia menor entre cada nodo, seleccionando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

Febrero 1998-2ª (problema 1)

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

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

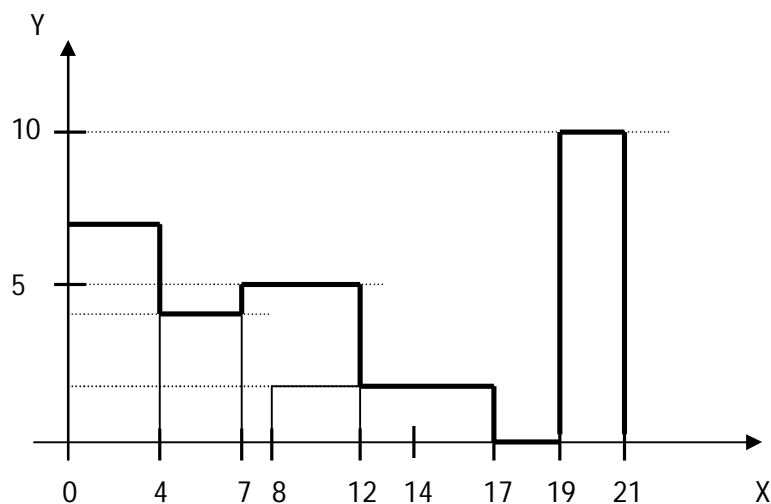
Respuesta: No sé exactamente el tipo de problema al que pertenece el mismo. Lo he preguntado y al final hemos llegado a la conclusión que puede ser un voraz. Realmente como digo no lo tengo nada claro, solo sé que los candidatos son los edificios y se trata de un problema de maximización (muy disimulado),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



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

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

Resaltaremos con **negrita** el contorno que nos piden que hagamos.



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

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

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

Septiembre 1998 (problema 1)

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

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

Septiembre 1998 (problema 2)

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

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

Febrero 2000-2ª (problema)

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

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

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

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

Febrero 2001-2ª (problema)

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

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

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

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

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

Septiembre 2004-reserva (problema)

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

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

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

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

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

siendo

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

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

Septiembre 2006-reserva (problema)

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

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

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

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