Análisis de algoritmos
16 de febrero de 2010 (semipresencial)

Prueba individual

Nombre: **Grupo:**

Análisis de algoritmos

Dada una matriz M de tamaño $m \times n$ y un valor *num*, los dos siguientes algoritmos buscan algún valor de los que contiene la matriz que aparezca exactamente *num* veces en una fila o en una columna:

```
def busca_valor_1(M,num):
    m=len(M)
    n=len(M[0])
    for fil in xrange(m): #Para cada fila de la matriz...
        for col in xrange(n):
            candidato=M[fil][col] #obtiene los posibles candidatos y comprueba si alguno aparece num veces.
            cuenta=0
            for col2 in xrange(n):
                if M[fil][col2]==candidato:
                    cuenta+=1
            if cuenta==num:
                return candidato
    for col in xrange(n): #Para cada columna de la matriz...
        for fil in xrange(m):
            candidato=M[fil][col] #obtiene los posibles candidatos y comprueba si alguno aparece num veces.
            cuenta=0
            for fil2 in xrange(m):
                if M[fil2][col]==candidato:
                    cuenta+=1
            if cuenta==num:
                return candidato
    return None
```

```
def busca_valor_2(M,num):
    m=len(M)
    n=len(M[0])
    for fil in xrange(m):
        for col in xrange(n):
            candidato=M[fil][col] #Para cada posible candidato de la matriz...
            for fil2 in xrange(m):#comprueba si en alguna fila aparece exactamente num veces, y
                cuenta=0
                for col2 in xrange(n):
                    if M[fil2][col2]==candidato:
                        cuenta+=1
                if cuenta==num:
                    return candidato
            for col2 in xrange(n):#comprueba si en alguna columna aparece exactamente num veces.
                cuenta=0
                for fil2 in xrange(m):
                    if M[fil2][col2]==candidato:
                        cuenta+=1
                if cuenta==num:
                    return candidato
    return None
```

Indica y justifica cuál es el coste temporal y espacial asintótico de ambos, distinguiendo entre el mejor y el peor de los casos si los hubiera (señala ante qué entradas se comportarían en el mejor caso y cuando en el peor caso). Si tuvieras que elegir uno de los dos algoritmos para resolver el problema, ¿con cuál te quedarías? Justificalo.

Estructuras de datos
23 de febrero de 2010 (semipresencial)

Prueba colectiva

Grupo:
Asistentes:

Selección de estructuras de datos eficientes para algoritmos

Una empresa ha puntuado a cada uno de sus N empleados en función del trabajo realizado durante el año y ha decidido otorgar una bonificación a los n mejores empleados ($n << N$). No obstante, para no discriminar a ninguna de las m secciones en que se divide su producción, ha decidido asignar un mínimo de p bonificaciones los p mejores empleados de cada una de las secciones (hay que tener en cuenta que $n > pm$).

Con esto, el procedimiento seguido consiste en bonificar a los p mejores trabajadores de cada sección y otorgar las restantes bonificaciones ($n - pm$) a los trabajadores de la empresa que presentan mejor puntuación, independientemente de la sección en la que estén.

Debes seleccionar las estructuras de datos adecuadas y describir con la mayor precisión un algoritmo que solucione el problema de la manera más eficiente posible, indicando y justificando el coste temporal asintótico asociado al mismo.

Divide y vencerás
2 de marzo de 2010 (semipresencial)

Prueba individual

Nombre: **Grupo:**

1. Algoritmo mergesort — 30 puntos

El siguiente algoritmo es una versión aleatorizada de la primera variante presentada de *mergesort*:

```
1 from random import randrange
2
3 def rdm_mergesort(a):
4   if len(a) <= 1:
5     return a
6   else:
7     j = randrange(1,len(a))
8     return merge(rdm_mergesort(a[:j]), rdm_mergesort(a[j:]))
```

(La llamada *randrange*`(1, len(a))` —línea 7— devuelve un número aleatorio entre 1 y $len(a) - 1$.)

Responde a las siguientes preguntas, *justificando tu respuesta en todos los casos*:

a) ¿Cuál es el coste temporal del algoritmo en el mejor y en el peor de los casos (si los hay)?

b) ¿Cuál es el coste espacial del algoritmo en el mejor y en el peor de los casos (si los hay)?

c) ¿Se podría reducir alguno de los costes anteriormente citados si en lugar de generar dos subvectores se trabajara con índices sobre el vector?¿Cuáles serían los costes mejorados resultantes?

d) ¿Presenta el algoritmo recursión por cola? Si es así, elimínala e indica si ello afecta a alguno de los costes asintóticos previamente calculados y en que sentido.

2. Resolución de relaciones de recurrencia (Teorema maestro y corolarios) **30 puntos**

Encuentra (justificándola) una cota asintótica superior para cada una de las siguientes relaciones de recurrencia, donde $T(n)$ presenta un valor constante en $n = 1$ y cuyo término recursivo general (para $n > 1$) es:

a) $T(n) = 4T(n/2) + nc_1 + c_2$.

b) $T(n) = T(n/2) + \sqrt{n}c_1$.

c) $T(n) = 2T(n/3) + n^2c_1 + n \lg nc_2$.

d) $T(n) = 3T(n/3) + n \lg nc_1 + nc_2$.

(Nota: c_1 y c_2 son constantes positivas arbitrarias.)

Divide y vencerás y búsqueda con retroceso
16 de marzo de 2010 (semipresencial)

Prueba colectiva

Grupo:
Asistentes:

Divide y vencerás: problema **90 puntos**

Sea a un vector que contiene valores de tres tipos: negativos, positivos y `None`. Queremos saber *si el vector contiene más valores negativos que positivos*. Para resolver el problema debéis tener en cuenta que todos los valores numéricos de un mismo tipo se encuentran agrupados en uno de los extremos del vector en zonas contiguas, mientras que todos los valores `None` están agrupados entre los dos grupos de valores numéricos. He aquí algunos ejemplos de vectores que cumplen esos requisitos:

a	-3	-6	-5	-2	-1	-5	None	None	None	6	4	9	1	7

a	3	4	None	-1	-3	-2	-6	-4	-2

a	-2	None	None	None	None	None	None	None	9	9

en los dos primeros casos la respuesta sería `True`, mientras que en el tercero sería `False`. Observad que los valores negativos y los positivos pueden estar indistintamente tanto al principio como al final del vector.

a) Diseñad un algoritmo *empleando la técnica recursiva de divide y vencerás* que reciba un vector a con las características previamente descritas e indique si contiene más valores negativos que positivos. El algoritmo desarrollado debe ser *lo más eficiente posible*, tanto desde el punto de vista de la complejidad temporal como de la espacial.

   Fijaos que lo que se pide es que la *estrategia fundamental* con la que resolvais el problema sea la de divide y vencerás, no que haya un único algoritmo de divide y vencerás. Dependiendo del enfoque elegido, puede que os lleve a diseñar una o varias funciones (de divide y vencerás) que sean empleadas para poder devolver el valor solicitado. En cualquier caso, no olvidéis incluir todo el código necesario para la resolución completa del problema.

b) El algoritmo que habéis desarrollado ¿tiene mejor y peor caso o se comporta de manera uniforme? Describid con claridad el mejor y el peor caso si los hubiera y realizad los correspondientes análisis de complejidad temporal y espacial justificando los resultados obtenidos.

c) Indicad si el algoritmo desarrollado (o alguna de sus funciones) presenta recursividad por cola y si su transformación en un algoritmo iterativo daría lugar a la variación de alguno de los costes calculados en el anterior apartado.

Si os sobra tiempo, podríais intentar resolver también la siguiente **cuestión adicional:**

d) En el caso en que vuestro algoritmo (o alguna de sus funciones) presente recursividad por cola, elimínadla. (Si empleais más de una función que presenta recursividad por cola, basta con que la eliminéis en una de ellas.)

Divide y vencerás y búsqueda con retroceso
16 de marzo de 2010 (semipresencial)

Prueba individual

Nombre: **Grupo:**

Búsqueda con retroceso: traza del problema de las n reinas **50 puntos**

En el problema de las n reinas hay que disponer n reinas en un tablero de $n \times n$ escaques de modo que ninguna amenace a otra. El método `bactracking` de la clase `NQueensSolver1` (página 6-7) permite encontrar una solución factible para el problema aplicando la técnica de búsqueda con retroceso:

```
class NQueensSolver1:

    def is_complete(self, s):
        return len(s) == self.n
    def is_promising(self, s):
        return all(s[j]!=s[i] and j-i!=abs(s[j]-s[i]) for i in xrange(len(s)) \
                                                      for j in xrange(i+1, len(s)))

    def backtracking(self, s):
        if self.is_complete(s): return s
        for row in xrange(self.n):
            s' = s + [row]
            if self.is_promising(s'):
                found = self.backtracking(s')
                if found != None: return found
        return None
```

El método también permite encontrar una solución factible que tenga un prefijo determinado.

Realiza una traza de la ejecución del método para la llamada `NQueensSolver1(5).backtracking([3,1])` (es decir, el problema de ubicar 5 reinas sin que se amenacen en un tablero de 5×5 donde las dos primeras tienen que ocupar las posiciones 3 y 1 respectivamente).

Debes dibujar el árbol de estados incluyendo tan sólo aquellos *que se generan* durante el proceso de búsqueda. Incluye también los estados que se estudian pero se descartan por no superar la condición de ser prometedores, y márcalos de forma especial. Indica claramente cuál es el resultado devuelto por el algoritmo.

Búsqueda con retroceso
23 de marzo de 2010 (semipresencial)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

Dado un grafo $G = (V, E)$, diseña un algoritmo de *búsqueda con retroceso* que encuentre un camino que pase por todas las aristas del grafo *una sóla vez* (sin repetir ninguna arista). Para ello se pide que:

a) Describas cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que solucione el problema, indicando cuál sería la llamada inicial que harías para resolverlo.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

Si os sobra tiempo, podríais intentar también responder a las siguientes **cuestiones adicionales:**

1. ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

2. Una variante del problema anterior consistiría en, dado un grafo $G = (V, E)$ y dos vértice u y v, encontrar el camino que pase por todas las aristas del grafo sin repetir aristas y que empiece en un vértice u y acabe en un vértice v. ¿Serías capaz de adaptar tu algoritmo para que resolviera dicho problema?

Algoritmos voraces
30 de marzo de 2010 (semipresencial)

Prueba individual

Nombre: **Grupo:**

Desglose óptimo de una cantidad de dinero **20 puntos**

Dados los sistemas monetarios con monedas de valores:

a) $1, 2, 4, 16$.

b) $1, 2, 5, 10, 20, 25$.

c) $1, 3, 5$.

d) $1, 5, 10, 20$.

Indica y justifica, para cada uno de ellos, si el algoritmo voraz `greedy_change` proporciona siempre (ante cualquier entrada válida) la solución óptima (si tu respuesta es negativa deberás encontrar un contraejemplo que lo demuestre).

Algoritmos voraces
30 de marzo de 2010 (semipresencial)

Prueba colectiva

Grupo:
Asistentes:

El problema de la mochila con fraccionamiento — 25 puntos

El problema de la mochila con N objetos que se pueden fraccionar se resuelve eficientemente con una estrategia voraz con un coste temporal de $O(N \lg N)$. ¿Es posible efectuar alguna modificación en cada uno de estos supuestos para obtener algoritmos más eficientes? Si es así, indica cómo y acompaña la explicación con el coste de los algoritmos resultantes.

a) Todos los productos pesan lo mismo.

b) Todos los productos valen lo mismo.

c) Todos los productos pesan y valen lo mismo.

d) La mochila no tiene límite de capacidad.

e) De cada producto se puede cargar una cantidad arbitraria en la mochila (es decir, se pueden cargar tanto fracciones de objetos como varias unidades de los mismos).

Programación dinámica
10 de mayo de 2010 (presencial)

Prueba individual

Nombre: **Grupo:**

Problema **50 puntos**

Deseamos colocar una valla de M metros en línea recta para marcar el linde entre dos terrenos. La construcción de la valla la haremos ensamblando, una tras otra, piezas de valla prefabricadas que podemos adquirir en promoción en una gran superficie. Hay piezas disponibles de N longitudes distintas. De cada tipo de pieza i ($1 \leq i \leq N$) conocemos su longitud, $l(i)$, su precio, $p(i)$, y el número de unidades que quedan en stock, $s(i)$. La promoción indica que al hacer una compra de piezas estamos obligados a adquirir un mínimo de una pieza de cada tipo, por lo que ya sabemos que al menos deberemos emplear una de cada en la construcción de la valla.

Deseamos conocer qué piezas hemos de comprar para construir una valla de longitud M con el menor coste posible. La siguiente ecuación recursiva devuelve el coste mínimo asociado a la mejor combinación de piezas:

$$C(i,m) = \begin{cases} 0, & \text{si } i = 0 \text{ y } m = 0; \\ +\infty, & \text{si } i = 0 \text{ y } m > 0; \\ +\infty, & \text{si } i > 0 \text{ y } m < l(i); \\ \min\limits_{1 \leq x \leq \text{mín}(\lfloor m/l(i) \rfloor, s(i))} C(i-1, m - x \cdot l(i)) + x \cdot p(i), & \text{si } i > 0 \text{ y } m \geq l(i). \end{cases}$$

donde la llamada $C(N, M)$ proporciona la mínima inversión necesaria para construir una valla de M metros gastando al menos una pieza de cada uno de los N tipos disponibles. Se pide:

a) Representa el grafo de dependencias asociado a la ecuación recursiva para un problema con $M = 9$ metros de valla y $N = 3$ tipos de piezas con las siguientes longitudes $l = [1, 2, 3]$, precios $p = [2, 3, 4]$ y unidades en stock $s = [4, 3, 1]$.

b) Escribe un algoritmo iterativo asociado a la anterior ecuación que permita obtener el valor del coste más económico con el que podemos construir la valla.

c) Indica cuál es el coste temporal y espacial del algoritmo previo justificando brevemente tu respuesta.

Programación dinámica
10 de mayo de 2010 (presencial)

Prueba colectiva

Grupo:
Asistentes:

Problema **80 puntos**

Sea el anterior problema:

****

Deseamos colocar una valla de M metros en línea recta para marcar el linde entre dos terrenos. La construcción de la valla la haremos ensamblando, una tras otra, piezas de valla prefabricadas que podemos adquirir en promoción en una gran superficie. Hay piezas disponibles de N longitudes distintas. De cada tipo de pieza i ($1 \leq i \leq N$) conocemos su longitud, $l(i)$, su precio, $p(i)$, y el número de unidades que quedan en stock, $s(i)$. La promoción indica que al hacer una compra de piezas estamos obligados a adquirir un mínimo de una pieza de cada tipo, por lo que ya sabemos que al menos deberemos emplear una de cada en la construcción de la valla.

Deseamos conocer qué piezas hemos de comprar para construir una valla de longitud M con el menor coste posible. La siguiente ecuación recursiva devuelve el coste mínimo asociado a la mejor combinación de piezas:

$$C(i,m) = \begin{cases} 0, & \text{si } i = 0 \text{ y } m = 0; \\ +\infty, & \text{si } i = 0 \text{ y } m > 0; \\ +\infty, & \text{si } i > 0 \text{ y } m < l(i); \\ \min\limits_{1 \leq x \leq \min(\lfloor m/l(i) \rfloor, s(i))} C(i-1, m - x \cdot l(i)) + x \cdot p(i), & \text{si } i > 0 \text{ y } m \geq l(i). \end{cases}$$

donde la llamada $C(N, M)$ proporciona la mínima inversión necesaria para construir una valla de M metros gastando al menos una pieza de cada uno de los N tipos disponibles.

****

Sobre el problema original se pide:

a) Escribid el algoritmo iterativo *que proporciona la secuencia de decisiones óptima, concretamente, debe proporcionar el número de piezas de cada tipo* que debemos comprar para efectuar el mínimo gasto.

A continuación, y debido a la finalización de la promoción en la superficie comercial, se producen las siguientes modificaciones en el enunciado del problema:

1. No hay obligación de comprar al menos una pieza de cada tipo.
2. El stock de piezas de cada tipo es ilimitado (pueden pedir las que necesitemos a la fábrica).

El resto de condiciones se mantienen.
Para este *nuevo problema* se pide:

a) Plantead la ecuación recursiva de programación dinámica que calcule, de la forma más eficiente posible, la mínima inversión que debemos realizar para construir la valla, indicando cuál debe ser la llamada inicial que se efectúe a dicha ecuación para obtener el valor óptimo.

b) Representad el grafo de dependencias entre las llamadas de la ecuación recursiva para los datos anteriores: $M = 9$ metros de valla y $N = 3$ tipos de piezas con las siguientes longitudes $l = [1, 2, 3]$ y precios $p = [2, 3, 4]$.

c) Indicad cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de vuestra ecuación recursiva que devolviera la *secuencia de decisiones óptima*. Justificad brevemente la respuesta.

d) ¿Sería posible reducir el coste espacial del apartado anterior si el algoritmo iterativo sólo tuviera que devolver *el valor* del máximo beneficio? Si es así, indicad y justificad cuál sería dicho coste.

Puntuación extra: Formalizad el nuevo problema en términos de optimización.

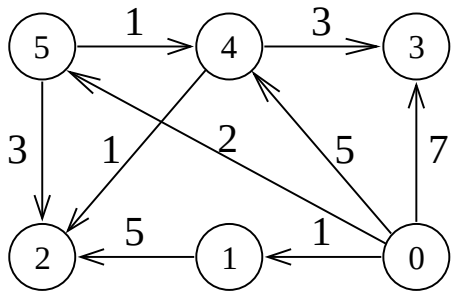
Algoritmos voraces
13 de abril de 2010 (semipresencial)

Prueba individual

Nombre: **Grupo:**

Algoritmo de Dijkstra: traza **35 puntos**

Dado el siguiente grafo dirigido y ponderado:



a) Haz una traza, iteración a iteración, de la ejecución de `all_shortest_paths_from_source2` (pág. 7-48, la primera versión del algoritmo de Dijkstra que utiliza punteros hacia atrás) sobre el grafo anterior para encontrar los caminos más cortos desde el vértice 0 a todos los demás. Para ello, rellena la siguiente tabla (haciendo uso de las filas que necesites):

	D						*backpointer*							
iter	0	1	2	3	4	5	0	1	2	3	4	5	*frontier*	*added*
inic.														
1														
2														
3														
4														
5														
6														
7														
8														
9														
10														
...														

b) Indica cuál es el *árbol* de caminos más cortos desde el vértice 0 a todos los demás que devuelve el algoritmo (puedes dibujarlo sobre el grafo).

Algoritmos voraces
13 de abril de 2010 (semipresencial)

Prueba colectiva

Grupo:
Asistentes:

Adaptación del algoritmo de Huffman — 30 puntos

El algoritmo de Huffman permite obtener una codificación binaria prefija para los n símbolos de un alfabeto de los que se conoce su frecuencia de aparición. Para ello construye un árbol binario. Dicho árbol cumple la propiedad de que, de entre todos los posibles, es aquel que minimiza el número esperado de bits en la codificación de los símbolos del alfabeto. El coste temporal del algoritmo es $O(n \lg n)$.

Estudia cada uno de los cuatro supuestos que se plantean a continuación. Para cada caso, indica si es posible modificar el algoritmo de Huffman o plantear alguna estrategia alternativa que iguale o mejore el coste de este. Si es así, indica cómo (señalando las estructuras de datos necesarias para la resolución) y acompaña la explicación con el coste temporal de los algoritmos resultantes:

a) Los símbolos del alfabeto se nos proporcionan inicialmente ordenados de menor a mayor frecuencia de aparición.

b) Los símbolos del alfabeto presentan todos la misma frecuencia de aparición.

c) Los símbolos del alfabeto cumplen que hay uno de ellos que se presenta la mitad de las veces (su frecuencia es 1/2), otro se presenta un cuarto de las veces (1/4), otro 1/8, otro 1/16 y así sucesivamente. Los símbolos se nos facilitan desordenados.

d) El mismo caso que el anterior, pero los símbolos se nos proporcionan ordenados de mayor a menor frecuencia.

En el caso de los supuestos b) y c), ¿podrías justificar cuál es el número máximo de bits que se utilizarán para representar un símbolo? ¿y el número mínimo?

Algoritmos voraces y programación dinámica
20 de abril de 2010 (semipresencial)

Prueba colectiva

Grupo:
Asistentes:

Análisis de la solución voraz de un problema — 65 puntos

Un profesor con m alumnos en su asignatura ha preparado nm trabajos. Cada alumno debe hacer n trabajos. A cada alumno se le ha pedido que puntue cada trabajo entre 0 y 10 según el interés que para él tendría realizar el trabajo. Por tanto, se dispone de una matriz P, donde cada celda $P[a][t]$ contiene la puntuación que el alumno a ($1 \leq a \leq m$) otorga al trabajo t ($1 \leq t \leq nm$).

Deseamos asignar trabajos a los alumnos de modo que la satisfacción global (suma de puntuaciones) sea máxima. Para resolver el problema hemos diseñado las siguientes *estrategias voraces*:

1. Se considera el primer alumno y se le asignan los n trabajos que prefiere; a continuación, se asignan al segundo alumno sus n trabajos preferidos de entre los no asignados al primero; y así sucesivamente.
2. Para cada alumno se calcula la suma de las puntuaciones de sus n trabajos preferidos y se ordena a los alumnos a partir de dicho valor de mayor a menor valor; considerando ese orden, se aplica el método del apartado anterior.
3. Se construye un máx-heap con todas la puntuaciones de la matriz (junto a las que se almacenan el alumno y trabajo asociados a cada puntuación); se van extrayendo los elementos del máx-heap y asignando los trabajos a los alumnos según el orden de extracción hasta que todos los alumnos tengan asignados sus n trabajos (si se extrae un trabajo que le corresponde a un alumno que ya tiene asignados sus n trabajos o se extrae un trabajo que ya ha sido asignado a otro alumno, dicho elemento se descarta).

Debes justificar *para cada estrategia* si es correcta (si está garantizado que ante una entrada válida *siempre* devuelve la solución óptima) o no (en el caso en que no lo sea, debes demostrarlo con un contraejemplo) y debes indicar, razonándolo, el coste temporal de la estrategia.

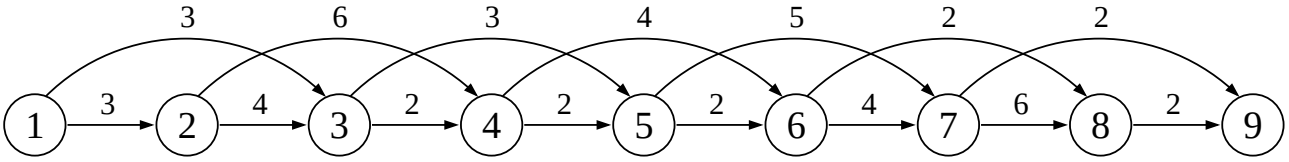
Algoritmos voraces y programación dinámica
20 de abril de 2010 (semipresencial)

Prueba individual

Nombre: **Grupo:**

Programación dinámica: traza del problema del trayecto más barato en el río Congo **50 puntos**

Debes encontrar cuál es la secuencia de embarcaderos óptima entre el 1 y el 9 para la siguiente instancia del problema del trayecto más barato en el río Congo:



Para ello, se pide:

a) Haz una traza del algoritmo `cheapest_price` (pág 7-10). Para ello, bien puedes utilizar como base el grafo anterior de forma similar a cómo se hace en el apartado (g) de la figura 7.8 (pág. 7-11), bien rellenar las estructuras C y B del algoritmo. No se pide una traza iteración a iteración, basta con que se muestre el contenido final de las estructuras citadas.

b) ¿Cuál es el valor de la solución óptima? ¿En qué estructura de las anteriores y en qué posición dentro de esta se almacena? ¿Qué resultado devuelve el algoritmo? ¿Qué significa este último resultado?

Programación dinámica
27 de abril de 2010 (semipresencial)

Prueba individual

Nombre: **Grupo:**

Problema de la mochila discreta **75 puntos**

Dada una mochila con capacidad para cargar $W = 4$ unidades de peso y $N = 5$ objetos que podemos cargar en ella, con pesos $w = [2, 1, 3, 2, 1]$ y valores $v = [10, 4, 13, 15, 7]$, se pide:

a) Dibuja, aprovechando la siguiente cuadrícula, el *grafo extendido de dependencias entre estados* asociado a dicha instancia del problema (utiliza las cajas que necesites, tachando el resto, y traza los arcos entre estados correspondientes al grafo para el problema propuesto):

(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)

b) Haz una traza del algoritmo `knapsack` (pág 7-22), el que permite recuperar la secuencia óptima de objetos. Para ello, escribe arriba de cada uno de los estados del grafo del apartado anterior el valor que el algoritmo almacena en el diccionario `V` y debajo del estado el valor que se almacena en el diccionario de punteros hacia atrás para el estado (alternativamente, puedes utilizar flechas tal y como se hacía en la figura 7-16).

c) ¿Cuál es el valor de la solución óptima? ¿En qué estado del grafo de dependencias se almacena? ¿Qué resultado devuelve el algoritmo? ¿Qué significa este último resultado?

Primer control parcial de Esquemas Algorítmicos (IG24)

27 de marzo de 2010

1. Divide y vencerás — 5 puntos

Sea a un vector que contiene valores numéricos que pueden aparecer repetidos. Los elementos se distribuyen en orden ascendente hasta una determinada posición, a partir de la cual aparecen en orden descendente. Además, todas las repeticiones de un mismo elemento están *agrupadas* ocupando posiciones contiguas dentro del vector. Deseamos encontrar la posición del elemento que tiene mayor valor. He aquí algunos ejemplos de vector, en los que se destacan en negrita las posiciones dentro del vector que se corresponden con el elemento de mayor valor (en los casos en que haya más de una, basta con indicar una cualquiera):

	0	1	2	3	**4**	**5**	6	7	8	9
a	1	3	3	3	11	11	6	6	6	4

	0	1	**2**	3	4	5	6	7	8	9	10	11	12	13
a	2	6	18	15	15	15	15	7	7	7	7	1	1	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	18	15	15	15	15	7	7	7	7	6	2	1	1	1

	0	1	2	3	4	5	6	7	8	9	10	11	12	**13**
a	1	1	1	2	6	7	7	7	7	15	15	15	15	18

Fíjate que en algunos casos puede aparecer más de una vez el elemento máximo (primer ejemplo), mientras que en otros (los dos últimos) el valor máximo puede estar en uno de los extremos del vector.

Debes resolver el problema de, dado un vector a tal y cómo se ha descrito, *averiguar cuál es la posición que ocupa el elemento con mayor valor*.

a) Diseña un algoritmo que reciba el vector a y resuelva el problema *mediante la técnica recursiva de divide y vencerás*. El algoritmo desarrollado debe ser *lo más eficiente posible*, tanto desde el punto de vista de la complejidad temporal como de la espacial.

   Fíjate que lo que se pide es que la *estrategia fundamental* con la que resuelvas el problema sea la de divide y vencerás, no que haya un único algoritmo de divide y vencerás. Dependiendo del enfoque elegido, puede que te lleve a diseñar una o varias funciones (de divide y vencerás) que sean empleadas para poder devolver el valor solicitado. En cualquier caso, no olvides incluir todo el código necesario para la resolución completa del problema.

   Pista: puede resultarte útil, dada la posición de un elemento en el vector, desarrollar y utilizar alguna función para determinar en que posición se encuentra la primera y/o la última aparición de dicho elemento en el vector.

b) El algoritmo que has desarrollado ¿tiene mejor y peor caso o se comporta de manera uniforme? Describe con claridad el mejor y el peor caso si los hubiera y realiza los correspondientes análisis de complejidad temporal y espacial justificando los resultados obtenidos.

c) Si el algoritmo desarrollado (o alguna de sus funciones) presenta recursividad por cola, elimínala e indica si alguno de los costes calculados en el anterior apartado varía. (Si has implementado más de una función que presenta recursividad por cola, basta con que la elimines en una de ellas, aunque deberás indicar qué otras funciones la tienen y señalar las posibles modificaciones de los costes asumiendo que hubieras transformado en iterativas dichas funciones.)

d) ¿Funcionaría tu algoritmo en el caso en el que los elementos repetidos no tuviesen que estar agrupados (es decir, que pudiesen aparecer indistintamente en el tramo de ascenso y en el de descenso)? Justifica brevemente tu respuesta.

2. Búsqueda con retroceso — 5 puntos

Disponemos de N objetos deformables, cada uno con un volumen $v_1, v_2, \ldots, v_N$, y de C cajas iguales, cada una con una capacidad volumétrica V, y nos gustaría encontrar, si existe, la forma de combinar los objetos para poder meterlos todos en las C cajas sin exceder su capacidad.

Queremos diseñar un algoritmo de *búsqueda con retroceso* capaz de resolver el problema. Se pide que:

a) Describas cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolver el problema.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

e) ¿Se te ocurre alguna manera de averiguar cuál sería el menor número de cajas necesarias para poder almacenar todos los objetos?

Segundo control parcial de Esquemas Algorítmicos (IG24)

15 de mayo de 2010

1. Algoritmos voraces — 3.25 puntos

En una asociación cultural con n socios, cada socio tiene la obligación de proponer un tema sobre el que puede impartir una conferencia para los miembros de la asociación que quieran asistir. Con el fin de reducir al mínimo el gasto, en la asociación han acordado *organizar el menor número posible de las conferencias propuestas*, con la condición de que se garantice que todos los socios participan al menos en una de ellas (sea como conferenciante o como público). Para ello se dispone de n listas: una por cada socio con las personas (incluido el propio socio conferenciante) a las que les gustaría asistir a la conferencia impartida por él. He aquí un ejemplo para $n = 8$ socios:

$$S1 = [S1, S2, S7], \quad S2 = [S1, S2, S6, S8], \quad S3 = [S3, S4, S6, S8], \quad S4 = [S3, S4, S5, S7], \quad S5 = [S4, S5],$$

$$S6 = [S2, S3, S6, S7, S8], \quad S7 = [S1, S4, S6, S7], \quad S8 = [S2, S3, S6, S8].$$

Una posible solución sería organizar las conferencias propuestas por los socios $S1$, $S2$, $S3$ y $S4$, ya que con esta combinación todos los socios imparten o asisten a alguna conferencia, aunque no sería la mejor solución, puesto que con los datos del ejemplo se podría encontrar una combinación con menor número de conferencias.

Para resolver el problema, y considerando inicialmente que los n socios son candidatos a dar su conferencia, se proponen estas dos Estrategías Voraces:

EV1: Seleccionar, de entre los candidatos, el socio cuya conferencia tiene un mayor número de solicitudes de asistencia. Esa conferencia se impartirá. Eliminar de los candidatos a los socios apuntados a su conferencia (entre los que se encuentra él mismo). Repetir el proceso hasta que no queden socios candidatos.

EV2: Buscar el socio candidato cuya conferencia tenga *menos* peticiones. De los socios que quieren asistir a ella, seleccionamos a aquel cuya conferencia tenga *más* solicitudes de asistencia y que aun sea candidato (si el socio inicial fuera el único que es candidato, él es el seleccionado). La conferencia del socio seleccionado se impartirá. Eliminar de los candidatos a los socios apuntados a su conferencia (entre los que se encuentra él mismo). Repetir el proceso hasta que no queden socios candidatos.

Se pide:

a) Como se trata de un problema de optimización, describe qué características debería tener una solución para ser factible, cuál sería la función objetivo y qué debe cumplir una solución factible para ser la óptima.

b) Aplica cada estrategia propuesta al problema del ejemplo e indica el resultado que proporcionaría, detallando en cada iteración el socio seleccionado y los descartados.

c) Para cada estrategia propuesta, justifica si encuentra siempre una solución factible al problema y, en caso de hacerlo, si es siempre la óptima (debes demostrar con un contraejemplo los casos en que no esté garantizada la obtención de una solución factible y/u óptima).

d) Para cada estrategia propuesta, señala el coste temporal que supone aplicarla (indica para ello las estructuras de datos que utilizarías en la implementación de un algoritmo lo más eficiente posible que siga la estrategia, describiendo los detalles del mismo que den lugar al coste señalado).

2. Programación dinámica — 3.75 puntos

Tenemos una caja en la que guardamos sellos de S valores distintos, donde $v(s)$ es el valor del sello del tipo s ($1 \leq s \leq S$) y $P(s)$ es la cantidad de sellos de ese tipo que tenemos guardados. Queremos enviar una carta cuyo valor de franqueo es F. Como hemos acumulado demasiados sellos nos gustaria emplear la mayor cantidad posible posible, gastando al menos $p(s)$ sellos de cada tipo ($1 \leq s \leq S$).

Deseamos saber, aplicando la estratégia de *programación dinámica*, cuántos sellos de cada tipo emplearemos para completar el franqueo exacto F de la carta de forma que el número total de sellos utilizados sea el máximo posible con las restricciones previamente indicadas.

1. Bajo las condiciones anteriormente expuestas:

   a) Formaliza el problema en términos de optimización.

   b) Plantea la ecuación recursiva asociada que calcula el máximo número de sellos necesarios para realizar el franqueo. Indica cuál debe ser la llamada que se efectue a dicha función para obtener el valor óptimo.

   c) Representa el grafo de dependencias entre llamadas de la ecuación recursiva para un franqueo de $F = 8$, si tenemos $S = 4$ tipos de sellos, con valores $v = [1, 2, 3, 5]$, número de sellos de cada tipo disponible $P = [4, 4, 5, 2]$ y debiendo emplear al menos $p = [2, 1, 1, 0]$. Indica y razona los costes espacial y temporal con los que un algoritmo iterativo podría efectuar los cálculos para resolver el problema (indica explícitamente, justificando tu respuesta, si es posible efectuar una reducción de la complejidad espacial si nos interesara obtener tan sólo el número de sellos máximo).

d) ¿Podría haber instancias del problema para las que no existiera una solución factible? Razona la respuesta y, si es así, indica las condiciones que se deberían cumplir para que pueda haber una solución factible.

2. Una variante del problema sería la siguiente: no tengo necesidad de gastar un mínimo número de sellos de cada valor, por contra me gustaría quedarme con al menos $p'(s)$ sellos de cada valor para futuros envíos.

   ¿Cómo modificarías la formulación del problema para que recogiera este nuevo planteamiento? ¿Cuál sería ahora la ecuación recursiva que resuelve el problema y la llamada que harías a la misma? ¿Con qué coste temporal y espacial se podría resolver?

3. Programación dinámica **3 puntos**

Queremos obtener el máximo beneficio posible alquilando una sala para la realización de una serie de actividades. Tenemos la opción de realizar A posibles actividades (numeradas entre 1 y A). De cada actividad a ($1 \leq a \leq A$) conocemos la hora de comienzo $s(a)$, la de terminación $t(a)$ y el número de particpantes $p(a)$. En la sala no puede realizarse más de una actividad al mismo tiempo. Todas las actividades pagan un precio fijo por cada participante en la misma, por lo que el máximo beneficio lo obtendremos con la combinación de actividades que mayor número total de asistentes atraigan a la sala. El número total de horas disponible de la sala es H (por simplicidad, todos los datos relacionados con las horas de comienzo y terminación de las actividades, así como de las horas en que está disponible la sala se han normalizado a valores enteros entre 0 y H.)

Debes averiguar, empleando la técnica de programación dinámica, qué actividades debemos seleccionar para obtener el mayor número de asistentes a la sala. Como ayuda, la siguiente ecuación recursiva de programación dinámica permite resolver el problema de averiguar el número de asistentes máximo que podrá ocupar la sala:

$$P(h) = \begin{cases} 0, & \text{si } h = 0; \\ \max(P(h-1), \max\limits_{\substack{1 \leq a \leq A: \\ t(a)=h}} P(s(a)) + p(a)), & \text{si } h > 0. \end{cases}$$

La llamada $P(H)$ proporciona dicho valor.
Se pide:

a) Representa el grafo de dependencias para el problema en el que el número de actividades es $A = 6$, la hora de cierre de la sala es $H = 9$ y las horas de comienzo, terminación y número de participantes en cada actividad son s_t_p = [(0,3,10), (2,4,8), (1,5,20), (5,7,9), (3,8,12), (4,8,15)] (cada triplete de la estructura s_t_p es una actividad, el primer componente es la hora de comienzo s, el segundo la hora de finalización t y el tercero el número de participantes p).

b) Diseña un algoritmo iterativo lo más eficiente posible que devuelva *las actividades* que se deben realizar para obtener el mayor número de participantes.

c) Indica cuál es el coste temporal y espacial del algoritmo diseñado justificando brevemente tu respuesta. ¿Crees que sería posible reducir el coste espacial si únicamente nos interesará obtener el valor del número de participantes máximo? Razona tu respuesta y, si es positiva, índica cuál sería dicho coste y justifícalo.

Examen convocatoria ordinaria de Esquemas Algorítmicos (IG24)

11 de junio de 2010

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo):
Parte 1 (Preg. 1 y 2) ☐ **Parte 2 (Preg. 3 y 4)** ☐ **Partes 1 y 2** ☐ **Renuncio a la evaluación alternativa** ☐
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Divide y vencerás — 2,5 puntos

Dados dos vectores que contienen cada uno n valores enteros positivos no repetidos, a ordenado de menor a mayor y b ordenado de mayor a menor, decimos que sus elementos están "hermanados crecientemente" si al elemento i-ésimo de a le asociamos el elemento $(n-1)-i$ de b, para $0 \leq i < n$. A los elementos $a[i]$ y $b[n-1-i]$ les llamamos "hermanos". Aquí tienes un ejemplo de dos vectores hermanados crecientemente con $n = 18$ elementos:

	0	1	2	3	4	5	6	7	8	9	10	11	12	**13**	14	15	16	17
a	2	**4**	8	10	12	15	23	*32*	40	42	44	58	60	**72**	84	89	93	102
b	113	100	97	96	**85**	73	71	62	54	53	*42*	30	22	14	8	7	5	1
	0	1	2	3	**4**	5	6	7	8	9	10	11	12	13	14	15	16	17

para el que, por ejemplo, el hermano del 32 ($a[7]$) es el 42 ($b[(18-1)-7] = b[10]$) y el hermano del 102 ($a[17]$) es el 113 ($b[(18-1)-17] = b[0]$).

Debes diseñar un algoritmo de *divide y vencerás* que, dados dos vectores a y b hermanados crecientemente y un valor v, devuelva *el par de elementos hermanos de cada vector cuya suma sea más cercana a* v. Para los anteriores vectores y el valor $v = 155$, el algoritmo debería devolver $(72, 85)$ cuya suma es 157, la más cercana a 155 entre hermanos y que se corresponde con los hermanos que ocupan las posiciones 13 en a y 4 en b (fíjate que $b[4]$ es el hermano de $a[13]$ porque (18-1)-13=4). Se pide:

a) Diseña un algoritmo recursivo *lo más eficiente posible* mediante la técnica de divide y vencerás que permita resolver este problema.

b) Analiza y justifica la complejidad temporal y espacial del algoritmo desarrollado en el apartado anterior, indicando si presenta un comportamiento uniforme o existe un mejor y un peor caso.

2. Búsqueda con retroceso — 2,5 puntos

En una organización mafiosa las rencillas internas han puesto en peligro tanto la integridad fisica de sus M miembros como la supervivencia de la organización. Tras una reunión, deciden dividir el territorio que controlan en N zonas independientes, a las que asignarán a los distintos miembros de la organización (un mafioso sólo podrá estar asignado a una zona). Queremos saber qué mafiosos se instalan en cada zona, teniendo en cuenta que dos mafiosos no pueden estar en la misma zona si previamente han tenido alguna disputa.

Para resolver el problema puedes asumir que los mafiosos aparecen numerados del 1 al M, las zonas de la 1 a la N y que dispones de una estructura de datos consistente en un vector `mafiosos_odiados` en el que en cada posicion, `mafiosos_odiados`[i] (para $1 \leq i \leq M$), se almacena un conjunto con los mafiosos con los que tiene cuentas pendientes el mafioso i y, por tanto, con los que no puede estar en la misma zona.

Debes desarrollar un algoritmo de *búsqueda con retroceso* que, dados M, N y la estructura `mafiosos_odiados`, devuelva el reparto de los mafiosos entre las zonas sin que estén en peligro (ellos y la organización). Se pide:

a) Describe cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribe un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que realizarías.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

3. Algoritmos voraces — 1,5 puntos

La programación diaria de una cadena de televisión que compite por tener una buena audiencia está compuesta por P programas (numerados entre el 1 y el P según el orden de emisión).

La cadena está obligada a clasificar los programas en F franjas o bloques de emisión consecutivos sabiendo que no es posible alterar el orden de emisión prefijado de los programas y que no puede haber más de M programas por franja. Es decir, a la primera franja irán los programas desde el 1 hasta el i (donde i es un valor variable, $1 \leq i \leq M$), en la segunda franja los programas desde el $i+1$ hasta el j ($i+1 \leq j \leq i+M$) y así sucesivamente.

Se dispone de estimaciones de la audiencia prevista para cada programa, $a_1, a_2, \ldots, a_P$, la audiencia de una franja es la audiencia del programa con mayor audiencia incluido en la franja y la audiencia global de la cadena se calcula sumando las audiencias de todas las franjas (de sus programas más vistos).

Teniendo en cuenta todo lo anterior, se desea encontrar la distribución de los P programas entre las F franjas de emisión que máximiza la audiencia de la cadena. Para ello, se propone la siguiente *estrategia voraz*, que se puede aplicar en tres etapas consecutivas:

1. Averiguamos cuáles son los F programas con mayor audiencia y los almacenamos por orden de emisión en un vector (de tipo offset) `mejores`.
2. Asignamos los programas a las franjas horarias de la siguiente manera: los programas comprendidos entre el 1 y `mejores[1]` van a la primera franja, los que están entre `mejores[1]+1` y `mejores[2]` van a la segunda franja, y así sucesivamente excepto para la última franja, en la que metemos los programas que están entre `mejores[F-1]+1` y el programa P.
3. Comprobamos que no haya ninguna franja horaria que contenga más de M programas, si fuera así desplazaríamos los programas por exceso a la(s) franja(s) anterior(es) y/o posterior(es) hasta que todas las franjas contengan cómo máximo M programas.

Se pide:

a) Ya que se trata de un problema de optimización, debes indicar las características que posee una *solución factible*, cuál es la función bjetivo que se pretende optimizar y qué debe cumplir una solución factible para ser considerada óptima.

b) Detalla cómo implementarías cada una de las etapas de la estrategia anterior para que el algoritmo resultante sea lo más eficiente posible: para cada etapa debes describir las características más importantes de su implementación y citar las estructuras de datos más significativas que utilizarías, así como el coste temporal de la misma. Finalmente, señala el coste temporal global del algoritmo.

c) Justifica si la estrategia propuesta encuentra siempre una solución factible al problema y, en caso de hacerlo, si es siempre la óptima (debes demostrar con un contraejemplo el caso en que no esté garantizada la obtención de la solución factible y/u óptima).

4. Programación dinámica — 3,5 puntos

Sea el problema presentado en el apartado de algoritmos voraces de encontrar la distribución de P programas de televisión entre F franjas de emisión que máximiza la audiencia de una cadena de televisión, con *todas las restricciones y condiciones allí fijadas* y con un par de modificaciones adicionales:

- Como mínimo hay que ubicar m programas en cada franja (además de no superar el máximo M).
- La audiencia de una franja no es la del programa con mayor audiencia de la franja, sino la media entre el programa de mayor audiencia y el de menor audiencia de la franja.

Deseamos diseñar un algoritmo mediante la estrategia de *programación dinámica* que determine qué programas ubicamos en cada franja para obtener la máxima audiencia. Se pide:

a) Formaliza el problema en términos de optimización.

b) Diseña una ecuación recursiva que calcule el valor de la máxima audiencia asociada a cualquier distribución de los programas entre las franjas, indicando cuál debe ser la llamada que se haga a dicha ecuación para resolver el problema.

c) Diseña un algoritmo iterativo que resuelva el problema, devolviendo el valor de la máxima audiencia empleando la menor ocupación espacial (en términos asintóticos) posible. Indica y justifica sus costes espacial y temporal.

Examen segunda convocatoria de Esquemas Algorítmicos (IG24)

17 de septiembre de 2010

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo):
Parte 1 (Preg. 1 y 2) ☐ **Parte 2 (Preg. 3 y 4)** ☐ **Partes 1 y 2** ☐ **Renuncio a la evaluación alternativa** ☐
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Divide y vencerás — 2,5 puntos

Dado un vector a que contiene una secuencia con un número indeterminado de valores `None` en sus primeras posiciones, otra secuencia de `None`'s en sus últimas posiciones y una serie de valores enteros positivos ordenados crecientemente en las posiciones que hay entre las dos secuencias de `None`'s, queremos averiguar *las posiciones de a en que comienza y acaba la secuencia numérica*. He aquí algunos ejemplos de vectores como el descrito, en los que hemos oscurecido los valores de las posiciones buscadas:

	0	1	2	**3**	4	5	6	**7**	8	9	10	11	12	13
a	None	None	None	1	2	4	7	12	None	None	None	None	None	None

	0	1	**2**	3	4	5	6	**7**	8
a	None	None	1	3	4	5	14	22	None

	0	1	2	3	4	5	6	**7**	**8**	9
a	None	None	None	None	None	None	None	13	29	None

Debes diseñar un algoritmo de *divide y vencerás* que, dado un vector a que cumple las condiciones señaladas, resuelva el problema de encotrar las posiciones inicial y final de la secuencia numérica. Se pìde:

a) Diseña un algoritmo recursivo *lo más eficiente posible* mediante la técnica de divide y vencerás que de solución a este problema.

b) Analiza y justifica la complejidad temporal y espacial del algoritmo desarrollado en el apartado anterior, indicando si presenta un comportamiento uniforme o existe un mejor y un peor caso.

2. Búsqueda con retroceso — 2,5 puntos

Disponemos de N objetos deformables, cada uno con un volumen $v_1, v_2, \ldots, v_N$, y de C cajas, cada una con una capacidad volumétrica $V_1, V_2, \ldots, V_C$, y nos gustaría encontrar, si existe, la forma de combinar los objetos para poder meterlos todos en las C cajas sin exceder su capacidad y siempre y cuando el número de objetos contenidos en cada caja no sea mayor de 10.

Queremos diseñar un algoritmo de *búsqueda con retroceso* capaz de resolver el problema. Se pide:

a) Describe cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribe un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolver el problema.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

3. Voraces — 1,5 puntos

Dos equipos de n miembros, a y b, van a enfrentarse en un concurso de inteligencia. En el concurso, cada miembro de un equipo participa una única vez en un enfrentamiento cara a cara directo con un miembro del otro equipo. El ganador del enfrentamiento consigue un punto para su equipo. El objetivo de cada equipo es obtener la máxima cantidad de puntos.

Con antelación al comienzo del concurso, cada equipo dispone de los cocientes intelectuales de todos sus miembros (pero no de los cocientes de los miembros del otro equipo). Sin embargo, el mecanismo del concurso puede favorecer al equipo b, ya que antes de cada enfrentamiento individual el equipo a debe indicar qué miembro de su equipo va participar y su cociente y, una vez sabido esto, el equipo b elige al miembro de su equipo que se enfrentará al del equipo a. Asumiendo que no hay dos concursantes con el mismo cociente y que cuando se enfrentan dos personas lo normal es que gane la de mayor cociente, diseña una *estrategia voraz* que permita al equipo b ganar la mayor cantidad posible de puntos.

Se pide:

a) Describe con la mayor precisión una estrategia voraz lo más eficiente posible que proporcione la máxima cantidad de puntos al equipo b. La estrategia deberá ser especialmente eficiente en el momento en que el equipo a facilite los datos de un concursante y el equipo b tenga que determinar qué miembro de su equipo concursará.

b) Indica cuál es el coste temporal de la estrategia que has diseñado, justificándolo adecuadamente. Señala las estructuras de datos empleadas.

4. Programación dinámica **3,5 puntos**

En un juego en el que nos enfrentamos varios amigos, en un instante dado dispongo de una baraja compuesta por C cartas, en la que cada carta c, $1 \leq c \leq C$, tiene puntos de ataque $p(c)$ y de defensa $p'(c)$. Para contrarrestar la jugada de un contrincante tengo que responder con la combinación de cartas que emplee el menor número de ellas y que proporcione una puntuación exacta de ataque de valor A y una de defensa de valor D.

Se pide:

a) Formaliza el problema en términos de optimización,

b) plantea la ecuación recursiva de *programación dinámica* que calcula el menor número de cartas necesario para alcanzar las puntuaciones indicadas, señalando cuál debe ser la llamada recursiva que se efectue a dicha ecuación para resolver el problema,

c) diseña un algoritmo recursivo con memorización que devuelva el valor del número mínimo de cartas que necesito emplear, indicando y justificando sus costes espacial y temporal.

Análisis de algoritmos y Estructuras de datos
25 de febrero de 2009 (Mañana)

Prueba individual

Nombre: **Grupo:**

Análisis de algoritmos y estructuras de datos

La siguiente función, dados dos vectores de números sin repetir, a y b, devuelve una lista con los numeros comunes a los dos vectores siempre que haya k o más elementos iguales; en caso contrario devuelve dos listas con los elementos distintos de cada vector:

```
1  def iguales_o_distintos(a,b,k):
2      iguales=[]
3      distintos_a=[]
4      for x in a:
5          if x not in b:
6              distintos_a.append(x)
7          else:
8              iguales.append(x)
9      if len(iguales)>=k:
10         return iguales
11     else:
12         distintos_b=[]
13         for y in b:
14             if y not in a:
15                 distintos_b.append(y)
16     return (distintos_a,distintos_b)
```

a) Indica y justifica cuál es el coste temporal y espacial asintótico del algoritmo, distinguiendo entre el mejor y el peor de los casos si los hubiera (señala ante qué entradas se comportaría en el mejor caso y cuando en el peor caso).

b) ¿Podrías utilizar alguna(s) estructura(s) de datos auxiliar(es) que te permitiera(n) reducir el coste temporal del algoritmo propuesto? Si es así, indica cuál(es), cómo la(s) emplearías y el coste temporal resultante.

Prueba colectiva

Grupo:
Asistentes:

Estructuras de datos y costes

Nos plantean un problema en el que vamos a almacenar valores numéricos en una cola de prioridad implementada mediante un heap, pero debes modificarlo para que, *sin afectar al coste de las operaciones básicas sobre heaps* (inserción de un elemento, obtención del mínimo y extracción de este), sea posible, *con el menor coste temporal posible*:

a) conocer en cualquier momento el valor de la suma de todos los elementos del heap, y

b) que no se inserte en la cola un valor que en ese momento ya exista en el heap.

Para cada caso debes describir brevemente (y con claridad) cómo incorporarías la modificación efectuada, detallando cómo afecta a las operaciones básicas sobre heaps, asi como el coste temporal que implica realizar la operación asociada a la modificación introducida y si afecta al coste espacial del heap.

Análisis de algoritmos y Estructuras de datos
25 de febrero de 2009 (Tarde)

Prueba individual

Nombre: **Grupo:**

Análisis de algoritmos y estructuras de datos

La siguiente función comprueba si en un vector de números a existen al menos k elementos mayores o iguales que uno dado x. Si es así, devuelve los k elementos del vector mayores y más cercanos a x:

```
1  def k_mayores_mas_cercanos(a,x,k):
2      b=[a[i] for i in xrange(len(a)) if a[i]>=x]
3      if len(b)<k:
4          return None
5      elif len(b)==k:
6          return b
7      else:
8          c=[None]*k
9          for i in xrange(k):
10             c[i]=min(b)
11             b.remove(min(b))
12         return c
```

a) Indica y justifica cuál es el coste temporal y espacial asintótico del algoritmo, distinguiendo entre el mejor y el peor de los casos si los hubiera (señala ante qué entradas se comportaría en el mejor caso y cuando en el peor caso).

b) ¿Podrías utilizar alguna estructura de datos auxiliar que te permitiera reducir el coste temporal del algoritmo propuesto? Si es así, indica cuál, cómo la emplearías y el coste temporal resultante.

Prueba colectiva

Grupo:
Asistentes:

Estructuras de datos y costes

Nos plantean un problema en el que tenemos necesidad de utilizar una estructura de datos para trabajar con multiconjuntos (conjuntos en los que un mismo elemento puede aparecer más de una vez). Concretamente, necesitaremos utilizar las siguientes operaciones, que queremos que se ejecuten de la forma *más eficiente posible*:

a) añadir un elemento a la estructura,

b) preguntar si un elemento está en la estructura,

c) eliminar un elemento de la estructura (si hubiera más de una aparición de dicho elemento, eliminar una cualquiera de ellas), y

d) eliminar todas las apariciones de un elemento en la estructura.

Debes señalar brevemente (y con claridad) cómo implementarías esta estructura (puedes tomar como base otra u otras estructuras que conozcas), detallando cómo diseñarías cada operación y cuál sería el coste temporal de cada una de ellas. Señala también el coste espacial asociado a la estructura.

Divide y vencerás
9 de marzo de 2009 (Mañana)

Prueba individual

Nombre: **Grupo:**

1. Algoritmo *min_max* (traza de la recursión)

Dado el vector:

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

dibuja el árbol de recursión que generaría la llamada al algoritmo de divide y vencerás *min_max*(*a*) (página 5-28). Para cada llamada debes señalar tanto los valores de los parámetros i y k en la función recursiva, como el valor devuelto por dicha llamada. ¿Cúal es el valor finalmente devuelto por el algoritmo?

2. Planteamiento y análisis de un problema

Dado un vector a que contiene valores enteros diferentes ordenados de menor a mayor, queremos averiguar cuantos de dichos valores cumplen que $i = a[i]$. Un ejemplo de vector sería el siguiente:

	0	1	2	3	**4**	**5**	**6**	7	8
a	-3	-2	0	2	**4**	**5**	**6**	9	11

en el que hay 3 valores (el 4, el 5 y el 6) que cumplen el criterio indicado (fijate en que, tanto en este caso como en cualquier otro, si hay varios valores que cumplen esa condición necesariamente deben aparecer en posiciones consecutivas del vector).

Describe con la mayor claridad posible cómo aplicarías la estrategia de divide y vencerás para abordar y resolver el problema de la forma más eficiente posible. Indica asimismo cuál crees que sería el coste temporal asociado a la estrategía planteada, dando una breve justificación del mismo.

Divide y vencerás
9 de marzo de 2009 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema

Dado un vector a que contiene valores enteros diferentes ordenados de menor a mayor, queremos averiguar cuantos de dichos valores cumplen que $i = a[i]$. Un ejemplo de vector sería el siguiente:

	0	1	2	3	**4**	**5**	**6**	7	8
a	-3	-2	0	2	**4**	**5**	**6**	9	11

en el que hay 3 valores (el 4, el 5 y el 6) que cumplen el criterio indicado (fijate en que, tanto en este caso como en cualquier otro, si hay varios valores que cumplen esa condición necesariamente deben aparecer en posiciones consecutivas del vector).

a) Diseñad un algoritmo recursivo *mediante la técnica de divide y vencerás* que devuelva el número total de valores que cumplen la condición descrita. El algoritmo desarrollado debe ser lo más eficiente posible, tanto desde el punto de vista de la complejidad temporal como de la espacial.

b) El algoritmo que habéis desarrollado ¿tiene mejor y peor caso o se comporta de manera uniforme? Describid con claridad el mejor y el peor caso si los hubiera y realizad los correspondientes análisis de complejidad temporal y espacial justificando los resultados obtenidos.

c) Modificad vuestro algoritmo para que reciba un valor u que representa un umbral. Cuando se efectue una llamada al algoritmo en la que el tamaño del vector que se vaya a analizar sea menor o igual que dicho umbral, el algoritmo deberá sustituir las llamadas recursivas por una búsqueda secuencial sobre el correspondiente fragmento de vector en la que se resuelva directamente el problema que se solucionaba recursivamente.

Divide y vencerás
9 de marzo de 2009 (Tarde)

Prueba individual

Nombre: **Grupo:**

1. Algoritmo *partition* (traza)

Haz la traza del comportamiento del algoritmo de partición de un vector empleado en *quicksort* (pág. 5-36) para la llamada: *partition*`([13, 4, 14, 9, 8, 12, 9, 14, 22, 16, 15, 18, 25, 22, 17], 0, 7)`.

Sobre la siguiente tabla debes detallar el valor de las variables i y j y el estado del vector justo antes de entrar en el bucle, después de cada una de las iteraciones del bucle y al final de la ejecución del algoritmo (si lo prefieres, puedes rellenar únicamente los valores de las casillas que se modifican entre una iteración y la siguiente). Marca también de forma especial la posición del pivote antes y después de la ejecución del algoritmo:

		vector														
j	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		13	4	14	9	8	12	9	14	22	16	15	18	25	22	17

2. Planteamiento y análisis de un problema

Dados dos vectores con n componentes, a, con valores ordenados de menor a mayor, y b, con valores ordenados de mayor a menor, queremos averiguar, de todas las i posiciones del vector ($0 \leq i < n$), cuál es la posición en la que los valores de $a[i]$ y $b[i]$ están más cercanos. En el siguiente ejemplo,

	0	1	2	3	**4**	5	6	7	8
a	-3	-2	0	2	**4**	8	10	12	15
b	13	10	7	6	**5**	3	1	-2	-4

la posición 4 del vector es en la que los valores están más cerca.

Describe con la mayor claridad posible cómo aplicarías la estrategia de divide y vencerás para abordar y resolver el problema de la forma más eficiente posible. Indica asimismo cuál crees que sería el coste temporal asociado a la estrategía planteada, dando una breve justificación del mismo.

Divide y vencerás
9 de marzo de 2009 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema

Dados dos vectores con n componentes, a, con valores ordenados de menor a mayor, y b, con valores ordenados de mayor a menor, queremos averiguar, de todas las i posiciones del vector ($0 \leq i < n$), cuál es la posición en la que los valores de $a[i]$ y $b[i]$ están más cercanos. En el siguiente ejemplo,

	0	1	2	3	**4**	5	6	7	8
a	-3	-2	0	2	**4**	8	10	12	15
b	13	10	7	6	**5**	3	1	-2	-4

la posición 4 del vector es en la que los valores están más cerca.

a) Diseñad un algoritmo recursivo *mediante la técnica de divide y vencerás* que devuelva el valor i en el que $a[i]$ y $b[i]$ son más cercanos. El algoritmo desarrollado debe ser lo más eficiente posible, tanto desde el punto de vista de la complejidad temporal como de la espacial.

b) El algoritmo que habéis desarrollado ¿tiene mejor y peor caso o se comporta de manera uniforme? Describid con claridad el mejor y el peor caso si los hubiera y realizad los correspondientes análisis de complejidad temporal y espacial justificando los resultados obtenidos.

c) ¿Presenta vuestro algoritmo recursión por cola? Si es así, eliminadla e indicad si ello afecta a alguno de los costes asintóticos previamente calculados y en que sentido.

Búsqueda con retroceso
25 de mayo de 2009 (Mañana)

Prueba individual

Nombre: **Grupo:**

Problema de las n reinas: variante **50 puntos**

En la variante del problema de las n reinas conocida como el problema de las n reinas *cojas*, la definición del problema es similar a la del de las n reinas, excepto en que las reinas (cojas) también amenazan a cualquier otra reina que esté en la misma fila o columna, pero no a las de las diagonales; en lugar de eso, amenazan a cualquiera que se ubique en una casilla vecina a ella (cada casilla tiene un máximo de 8 casillas vecinas).

Indica cómo modificarías el algoritmo que resuelve el problema de las n reinas (no cojas) para que resuelva el de las n reinas cojas. Puedes partir de cualquiera de los algoritmos (o de las funciones que estos utilizan) que hemos visto en clase siempre que señales qué versión estás usando e indiques claramente y con detalle los cambios que realizas en ella.

Búsqueda con retroceso
25 de mayo de 2009 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

El juego del dominó dispone de 28 piezas rectangulares de igual tamaño divididas en dos partes. En cada parte de la pieza aparece un número de puntos (desde 0 puntos hasta 6 puntos). No hay dos piezas con la misma combinación de puntos en sus dos partes, y entre todas engloban toda posible combinación de puntos (desde el (0,0) hasta el (6,6)).

Podemos juntar dos o más piezas de dominó uniendo dos de sus partes libres por un extremo sólo si el valor de las partes unidas es el mismo (la pieza (0,1) no se podría juntar con la (2,3), pero sí con la (2,1), de esta manera, (0,1)(1,2); a partir de ahí, podríamos ahora unir, por ejemplo, la pieza (2,4) o la (0,4) a las dos anteriores, pero no la (1,4)). Quisieramos, partiendo de una pieza cualquiera, juntar todas las piezas del juego de forma correcta.

Para ello debes diseñar un algoritmo de *búsqueda con retroceso* que devuelva como resultado una secuencia de piezas que cumpla las condiciones comentadas.

Se pide que:

a) Describas cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolverlo.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

Si os sobra tiempo, podríais intentar responder también a las siguientes **cuestiones adicionales:**

1. ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

2. Una variante del problema anterior consistiría en que la última pieza de la secuencia cumpla que pueda unirse a la primera. ¿Serías capaz de adaptar tu algoritmo para que resolviera dicho problema?

Búsqueda con retroceso
25 de mayo de 2009 (Tarde)

Prueba individual

Nombre: **Grupo:**

Problema de la suma del subconjunto: variante **50 puntos**

En una variante del problema de la suma del subconjunto, además del peso se considera también el volumen: dados N objetos con pesos $w_1, w_2, \ldots, w_N$ y volumen $v_1, v_2, \ldots, v_N$ y una mochila con capacidad para soportar una carga W y un volumen V, deseamos cargar una selección arbitraria de objetos cuyo peso sea exactamente W sin sobrepasar el volumen máximo admitido por la mochila.

Indica cómo modificarías el algoritmo que resuelve el problema de la suma del subconjunto para que resuelva esta nueva versión. Puedes partir de cualquiera de los algoritmos (o de las funciones que estos utilizan) que hemos visto en clase siempre que señales qué versión estás usando e indiques claramente y con detalle los cambios que realizas en ella.

Búsqueda con retroceso
25 de mayo de 2009 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

En cada una de las seis caras de los dados empleados en los juegos infantiles aparece un número distinto de puntos (entre 1 y 6 puntos). Dados n dados y un valor M, quisieramos seleccionar una cara de cada uno de los dados de forma que el producto de los puntos que aparecen en las caras seleccionadas de todos los dados sea exactamente M.

Para ello debes diseñar un algoritmo de *búsqueda con retroceso* que devuelva como resultado una secuencia con los valores de la cara seleccionada de cada dado de forma que su producto sea M.

Se pide que:

a) Describas cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolverlo.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

Si os sobra tiempo, podríais intentar responder también a las siguientes **cuestiones adicionales:**

1. ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

2. Una variante del problema anterior no permite que en la solución pueda aparecer el mismo valor de un dado más de V veces. ¿Serías capaz de adaptar tu algoritmo para que resolviera dicho problema?

Algoritmos voraces
8 de abril de 2009 (Mañana)

Prueba individual

Nombre: **Grupo:**

Kruskal **40 puntos**

El algoritmo de Kruskal encuentra el árbol de recubrimiento de coste mínimo en un grafo no dirigido y ponderado. No obstante, resulta interesante considerar algunos casos especiales, pues podría ser que encontrar el árbol de recubrimiento de coste mínimo para ellos fuera más eficiente si se modificase el algoritmo de Kruskal. Dados los siguientes supuestos:

1. Todas las aristas del grafo pesan 1.
2. Todas las aristas del grafo tienen como peso uno de dos posibles valores.
3. El grafo es un grafo conexo en el que cada vértice tiene exactamente dos aristas y todas las aristas forman un ciclo hamiltoniano.
4. Las aristas del grafo tienen pesos enteros en el rango $[0..|V|]$.

Indica qué modificaciones del algoritmo de Kruskal harías *en cada supuesto* para solucionar estos problemas con la mayor eficiencia computacional. Señala también, justificándolo, el coste temporal de las propuestas.

Algoritmos voraces
8 de abril de 2009 (Mañana)

Prueba colectiva

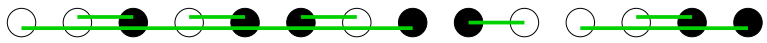
Grupo:
Asistentes:

Análisis de la solución voraz de un problema — 60 puntos

En una línea recta hay n puntos blancos y n puntos negros distribuidos equiespaciadamente: la distancia entre puntos consecutivos siempre es de un metro.

Deseamos conectar con cable cada punto negro con un punto blanco, *gastando la menor cantidad posible de cable*. La cantidad de cable necesario para empalmar dos puntos es igual a la distancia que los separa. Se nos ha ocurrido un algoritmo voraz para resolver el problema al que denominamos "Primero los más cercanos": Se divide el conjunto de puntos en "vivos" y "muertos". Inicialmente todos los puntos están vivos. Se ordena el conjunto de todas las posibles conexiones por valor no decreciente de distancia. Seleccionamos la primera posible conexión (la que consume menor cantidad de cable) y nos aseguramos de que une puntos vivos. Si es así, conectamos ambos puntos y los declaramos muertos. Iteramos el procedimiento hasta que todos los puntos estén muertos.

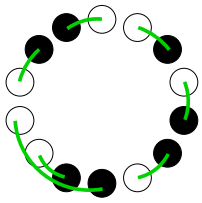
La figura muestra una conexión de puntos blancos con negros que se obtiene de aplicar el método propuesto sobre la figura anterior:



Las conexiones utilizan $1 + 1 + 1 + 1 + 1 + 7 + 3 = 15$ metros de cable.

a) Justificad si el algoritmo propuesto es correcto (*proporciona siempre la solución óptima*) y señalar, razonándolo, el coste temporal del mismo.

b) Suponiendo que los puntos se distribuyeran en un círculo tal y como se muestra en la siguiente figura (y manteniendo el resto de condiciones anteriores), un resultado posible de la aplicación del anterior algoritmo sería el siguiente:



¿Sería ahora correcto el método descrito? Justificadlo

Algoritmos voraces
8 de abril de 2009 (Tarde)

Prueba individual

Nombre: **Grupo:**

Búsqueda del camino más corto entre dos vértices **40 puntos**

Dado un grafo dirigido $G = (V, E)$ y ponderado por una función $d : E \to \mathbb{R}^{\geq 0}$, deseamos calcular de *la manera más eficiente posible* el camino más corto entre un vértice s y otro t. Sabemos que dicho camino se puede calcular haciendo uso de un conocido algoritmo (que presenta algunas variantes). Ante determinados supuestos puede que fuera más eficiente utilizar una estrategia alternativa o emplear una variante concreta del algoritmo. Dados los siguientes supuestos:

1. El grafo consta de una serie de vértices numerados de forma consecutiva (entre 0 y $|V|-1$), las únicas aristas que hay unen vértices consecutivos, $E = \{(i, i+1) \mid 0 \leq i < |V|\}$, y los vértices origen y destino del camino cumplen $s < t$.

2. El grafo es un árbol dirigido implementado mediante la representacion con referencias a padres (recordatorio: mediante un diccionario con una entrada por vértice en la que se almacena como valor el padre de dicho vértice en el árbol) y el vértice origen del camino s cumple que es un ascendente del vértice destino t.

3. El grafo tiene la particularidad de que de cada vértice salen aristas a todos los demás.

4. El grafo tiene grado de salida 3.

Debes describir brevemente qué harías *en cada supuesto* para obtener de la forma más eficiente posible el camino más corto (estrategia que aplicarías, algoritmo o variante que escogerías, modificación que harías sobre algún algoritmo...), así como justificar el coste temporal en función de $|V|$ y $|E|$ que conlleva la aplicación de la estrategia elegida.

Algoritmos voraces
8 de abril de 2009 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Análisis de la solución voraz de un problema **60 puntos**

Hemos montado una agencia de contactos personales (heterosexuales). Tenemos N clientes masculinos y M clientes femeninas a los que hemos pedido que cuantifiquen su interés con un número entre 0 (ningún interés) y 10 (mucho interés) por fijar una cita con cada una de las personas del otro sexo. El valor $c_{a,b}$ es el interés manifestado por la persona a en fijar una cita con la persona b.

Te presentamos un ejemplo suponiendo que hay $N = 3$ hombres y $M = 2$ mujeres. La matriz de la izquierda muestra el interés que cada hombre ha manifestado por mantener una cita con cada mujer y la matriz de la derecha muestra el interés que cada mujer ha manifestado por cada hombre:

	María	Nerea
Antonio	7	10
Benito	0	1
Carlos	3	0

	Antonio	Benito	Carlos
María	10	0	7
Nerea	6	8	1

El índice de compatibilidad de una pareja (a, b) se define como el producto de $c_{a,b}$ por $c_{b,a}$ y es, evidentemente, un valor numérico entre 0 y 100. En el ejemplo, el índice de compatibilidad entre Antonio y María es 70 ($= 7 \cdot 10$) y el índice de compatibilidad entre Benito y Nerea es 8 ($= 1 \cdot 8$).

La empresa cobra en función de cada cita organizada. El importe facturado por cita es una cantidad de euros igual al índice de compatibilidad de las personas convocadas. Acuciados por las deudas, hemos decidido organizar K citas en el mismo día, por lo que no podemos convocar a una misma persona a dos citas. Para maximizar el beneficio decidimos aplicar el siguiente procedimiento voraz y aplicarlo K veces: citar al par de personas de mayor compatibilidad y eliminar a ambas de la lista de personas a las que podemos citar. En el ejemplo, con $K = 2$, si citamos a Antonio y María por una parte ($7 \cdot 10 = 70$ euros) y, por otra, a Benito y Nerea ($1 \cdot 8 = 8$ euros), ganamos un total de 78 euros.

a) ¿Qué coste temporal presenta ese algoritmo?

b) ¿Encuentra siempre ese algoritmo la solución óptima, es decir, encuentra las K citas (sin repetir persona) que mayor beneficio nos reportan?

Cuando ya hemos saneado la situación financiera, trabajamos con algo más de calma y organizamos una nueva serie de K citas. Como ya no tenemos la restricción de que todas sean en el mismo día, planificamos estas nuevas K citas con la misma estrategia voraz (escoger cada vez la pareja con mayor compatibilidad), pero con la posibilidad de que una misma persona participe en más de una cita. En el ejemplo, también con $K = 2$, podemos citar a Antonio y María por una parte ($7 \cdot 10 = 70$ euros) y a Antonio y Nerea por otra ($10 \times 6 = 60$ euros), con un beneficio total de 130 euros.

c) ¿Qué coste temporal presenta ese algoritmo?

d) ¿Encuentra siempre ese algoritmo la solución óptima, es decir, encuentra las K citas (con la posibilidad de repetir persona, pero no pareja) que mayor beneficio nos reportan?

Debéis razonar con claridad y concisión las respuestas.

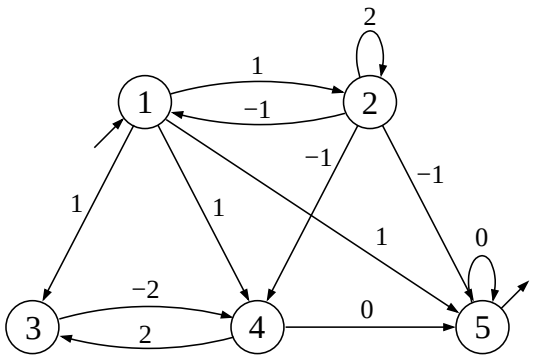
Programación dinámica
4 de mayo de 2009 (Mañana)

Prueba individual

Nombre: **Grupo:**

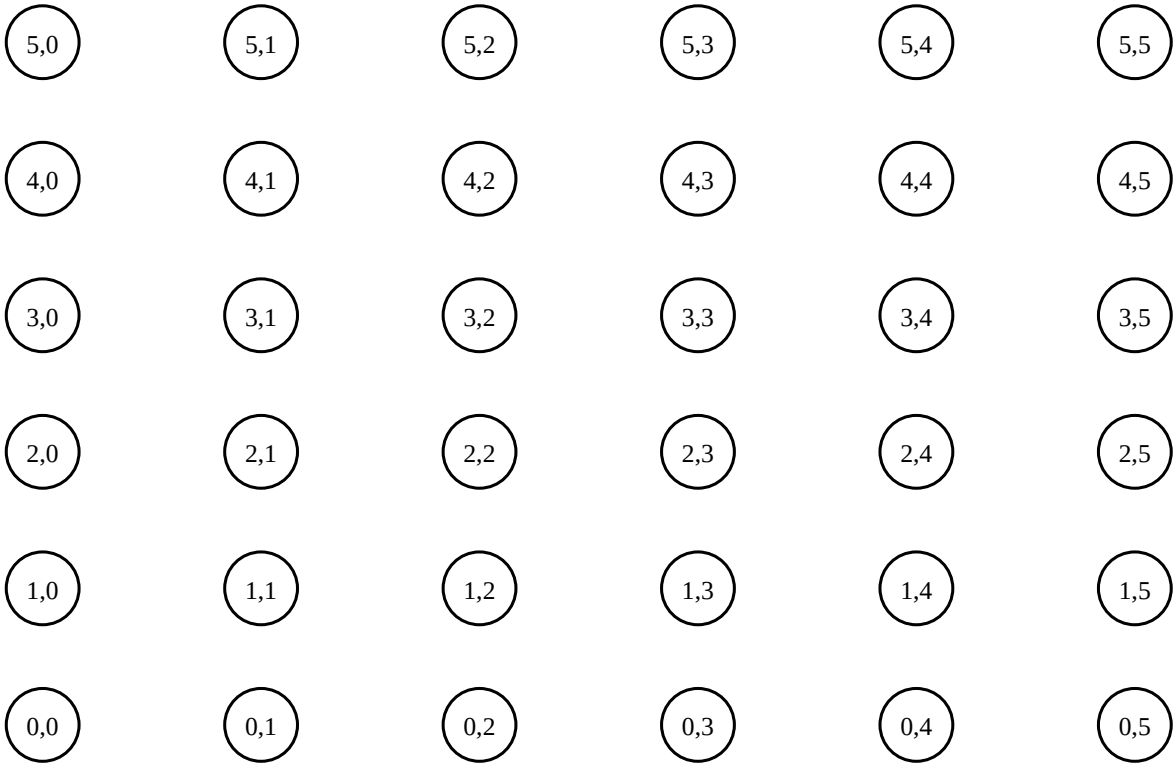
Problema del camino más corto formado por k aristas en un grafo **30 puntos**

Debes encontrar cuál es el camino más corto que empieza en el vértice $s = 1$ y acaba en el vértice $t = 5$ formado por exactamente $k = 3$ aristas en el siguiente grafo:



Para ello, se pide:

a) Dibuja, aprovechando la siguiente figura, el *grafo multietapa de dependencias* asociado al grafo anterior y a la ecuación recursiva (7.8) (pág. 7-37) que resolvía el problema. (Utiliza los nodos que necesites, tachando el resto, y traza los arcos entre estados correspondientes al problema propuesto):



b) Haz una traza del algoritmo `k_edges_shortest_distance` (pág 7-38). Para ello, escribe arriba de cada uno de los estados del grafo del apartado anterior el valor que el algoritmo almacena en el diccionario `D`.

c) ¿Cuál es el valor de la solución óptima? ¿En qué estado se encuentra? ¿Cuál sería la secuencia de aristas del grafo original que devolvería un algoritmo que calculara la solución óptima del problema (si hay más de una secuencia óptima, indica una cualquiera)?

Programación dinámica
4 de mayo de 2009 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema de la mochila con exactamente M objetos — 40 puntos

La siguiente ecuación recursiva permite plantear la resolución del problema de la mochila discreta cuando deseamos obtener el máximo beneficio (sin exceder la capacidad de carga W) cargando exactamente M objetos de los N disponibles:

$$V(i,c,m) = \begin{cases} 0, & \text{si } m = 0; \\ -\infty, & \text{si } m > 0 \text{ e } (i = 0 \text{ o } c = 0); \\ \max\left(V(i-1,c,m), V(i-1,c-w_i,m-1)+v_i\right), & \text{si } m > 0,\ i > 0,\ c > 0 \text{ y } w_i \leq c; \\ V(i-1,c,m), & \text{si } m > 0,\ i > 0,\ c > 0 \text{ y } w_i > c. \end{cases}$$

donde la llamada $V(N,W,M)$ proporciona el valor de la solución óptima.

Debéis escribir el algoritmo recursivo con memorización asociado a la anterior ecuación, justificando su coste temporal y espacial.

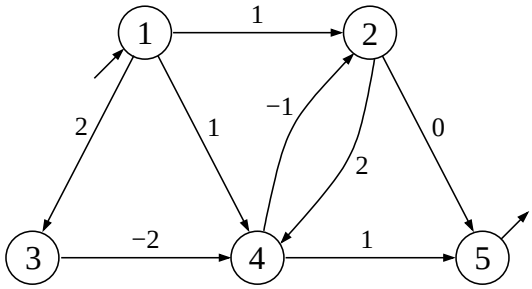
Programación dinámica
4 de mayo de 2009 (Tarde)

Prueba individual

Nombre: **Grupo:**

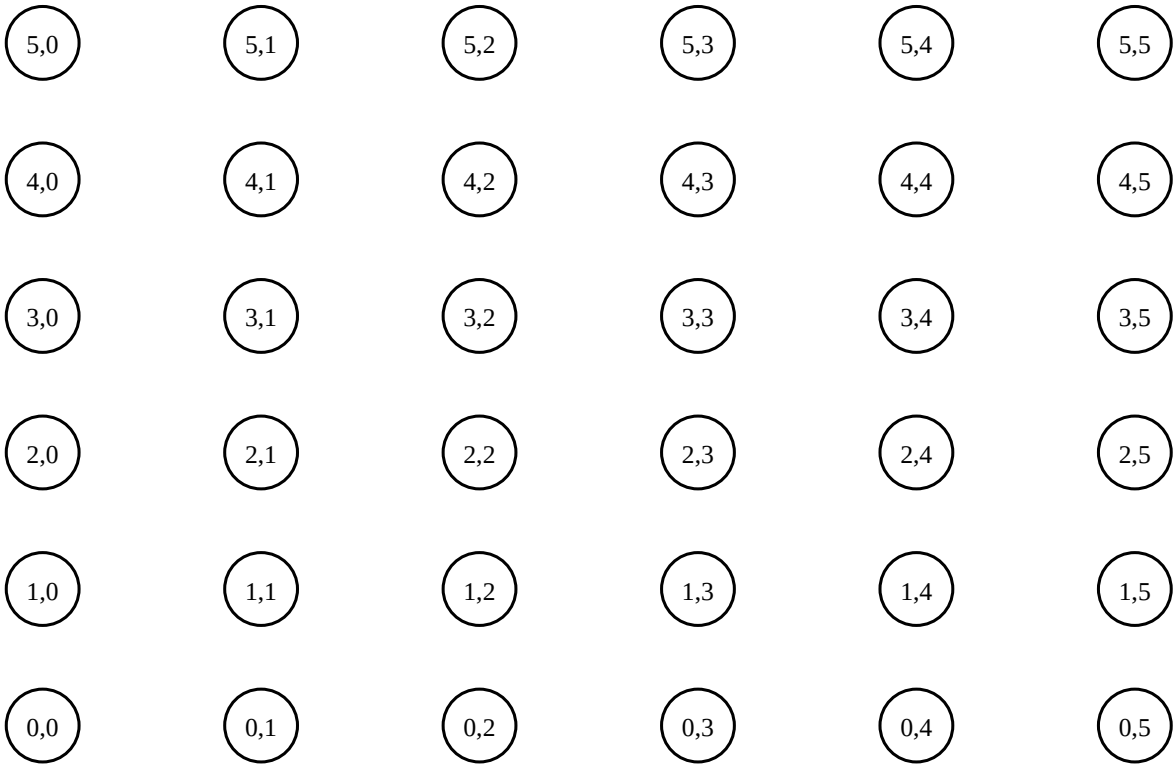
Problema del camino más corto en un grafo sin ciclos negativos: Bellman-Ford **30 puntos**

Debes encontrar cuál es el camino más corto que empieza en el vértice $s = 1$ y acaba en el vértice $t = 5$ en el siguiente grafo que contiene ciclos no negativos:



Para ello, se pide:

a) Dibuja, aprovechando la siguiente figura, el *grafo multietapa de dependencias* asociado a la resolución del problema que se propone en la pág. 7-39 para el ejemplo anterior. (Utiliza los nodos que necesites, tachando el resto, y traza los arcos entre estados correspondientes al problema propuesto):



b) Haz una traza del algoritmo de Bellman-Ford `shortest_distance` (pág 7-40). Para ello, escribe arriba de cada uno de los estados del grafo del apartado anterior el valor que el algoritmo almacena en el diccionario `D`.

c) ¿Cuál es el valor de la solución óptima? ¿En qué estado se encuentra? ¿Cuál sería la secuencia de aristas del grafo original que devolvería un algoritmo que calculara la solución óptima del problema (si hay más de una secuencia óptima, indica una cualquiera)?

Programación dinámica
4 de mayo de 2009 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema de la mochila con un número ilimitado de objetos de cada valor **40 puntos**

La siguiente ecuación recursiva permite plantear una resolución del problema de la mochila discreta cuando deseamos obtener el máximo beneficio (sin exceder la capacidad de carga W a partir de N tipos de objetos diferentes) cuando disponemos de un número ilimitado de objetos de cada valor v_i y peso w_i ($1 \leq i \leq N$):

$$V(i,c) = \begin{cases} 0, & \text{si } i = 0 \text{ o } c = 0; \\ \max\limits_{0 \leq x \leq \lfloor c/w_i \rfloor} \left(V(i-1, c - w_i * x) + v_i * x \right), & \text{si } i > 0 \text{ y } c > 0. \end{cases}$$

donde la llamada $V(N,W)$ proporciona el valor de la solución óptima.

a) Dibujad, aprovechando la siguiente cuadrícula, el *grafo multietapa de dependencias* asociado a la anterior ecuación recursiva para los datos $W = 6$, $N = 5$, $v = [90, 75, 60, 20, 10]$ y $w = [4,3,3,2,2]$:

(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)

b) Escribid el algoritmo iterativo que permite conocer el valor de la solución óptima para la ecuación recursiva propuesta, justificando su coste temporal y espacial.

Programación dinámica
13 de mayo de 2009 (Mañana)

Prueba individual

Nombre: **Grupo:**

Problema **50 puntos**

Un vendedor ambulante de productos de artesanía realiza una ruta a través de una serie de P pueblos dispuestos a lo largo de una carretera. Parte de la ciudad en la que habita (situada al comienzo de la carretera) y debe llegar a una ciudad que se encuentra en el otro extremo de la carretera sin retroceder nunca. Su sistema de ventas es siempre el mismo: un día llega a un pueblo por la mañana, vende toda la mercancía, compra la artesanía típica del pueblo y continúa hasta otro pueblo, en el que, al día siguiente, repetirá la operación. Al principio del viaje partirá con productos típicos de su ciudad y al acabar deberá vender la mercancía que le quede en la ciudad final de trayecto. El comerciante, una vez ha parado en un pueblo, no vuelve a detenerse hasta haber atravesado un mínimo de dos pueblos más (al ser pueblos vecinos no podría vender a buen precio los objetos).

El comerciante tiene una estimación del beneficio que puede obtener en un determinado pueblo en función de la mercancía que venda en él. Es decir, si numeramos los pueblos entre 1 y P según el orden en que aparecen en la carretera, el beneficio que obtiene al vender la mercancia de un pueblo i en otro j sería, $b(i, j)$, para $1 \leq i < j \leq P$.

Se pretende averiguar en que pueblos debería parar para obtener el máximo beneficio posible.

La siguiente ecuación recursiva devuelve el valor del máximo beneficio que puede obtener el comerciante en una ruta entre el pueblo 1 y el pueblo p:

$$B(p) = \begin{cases} 0, & \text{si } p = 1; \\ -\infty, & \text{si } p = 2 \text{ o } p = 3; \\ \max\limits_{1 \leq p' \leq p-3} B(p') + b(p', p), & \text{si } p > 3. \end{cases}$$

donde la llamada $B(P)$ proporciona el máximo beneficio para la ruta completa entre 1 y P. Se pide:

a) Escribe el algoritmo recursivo con memorización asociado a la anterior ecuación.

b) Representa el grafo de dependencias asociado a la ecuación recursiva para un problema con $P = 10$ pueblos.

c) Indica cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de la ecuación presentada. Justifica brevemente tu respuesta.

Programación dinámica
13 de mayo de 2009 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

Sea el anterior problema:

****

Un vendedor ambulante de productos de artesanía realiza una ruta a través de una serie de P pueblos dispuestos a lo largo de una carretera. Parte de la ciudad en la que habita (situada al comienzo de la carretera) y debe llegar a una ciudad que se encuentra en el otro extremo de la carretera sin retroceder nunca. Su sistema de ventas es siempre el mismo: un día llega a un pueblo por la mañana, vende toda la mercancía, compra la artesanía típica del pueblo y continúa hasta otro pueblo, en el que, al día siguiente, repetirá la operación. Al principio del viaje partirá con productos típicos de su ciudad y al acabar deberá vender la mercancía que le quede en la ciudad final de trayecto. El comerciante, una vez ha parado en un pueblo, no vuelve a detenerse hasta haber atravesado un mínimo de dos pueblos más (al ser pueblos vecinos no podría vender a buen precio los objetos).

El comerciante tiene una estimación del beneficio que puede obtener en un determinado pueblo en función de la mercancía que venda en él. Es decir, si numeramos los pueblos entre 1 y P según el orden en que aparecen en la carretera, el beneficio que obtiene al vender la mercancia de un pueblo i en otro j sería, $b(i, j)$, para $1 \leq i < j \leq P$.

Se pretende averiguar en que pueblos debería parar para obtener el máximo beneficio posible.

La siguiente ecuación recursiva devuelve el valor del máximo beneficio que puede obtener el comerciante en una ruta entre el pueblo 1 y el pueblo p:

$$B(p) = \begin{cases} 0, & \text{si } p = 1; \\ -\infty, & \text{si } p = 2 \text{ o } p = 3; \\ \max\limits_{1 \leq p' \leq p-3} B(p') + b(p', p), & \text{si } p > 3. \end{cases}$$

donde la llamada $B(P)$ proporciona el máximo beneficio para la ruta completa entre 1 y P.

****

Sobre el problema original se pide:

a) Escribid el algoritmo iterativo *que proporciona la secuencia de decisiones óptima, concretamente, debe proporcionar los pueblos del recorrido* que permite obtener el máximo beneficio al comerciante.

A continuación, se proponen las siguientes modificaciones en el enunciado del problema del vendedor ambulante:

1. El vendedor debe hacer la ruta en D días (es decir, deberá pasar exactamente por D pueblos, sin contar el pueblo de origen).
2. Que la mercancia de un pueblo no se pueda vender en los dos siguientes porque no da beneficio es un criterio muy general. El vendedor lo ha mejorado y ahora, para cada pueblo p, sabe cuál es el pueblo anterior más cercano, $v[p]$, cuya venta de mercancia en p ya proporciona beneficio. Por ejemplo, $v[7] = 5$ implica que en el pueblo 7 podemos vender con beneficio la mercancia de pueblo 5 y de todos los anteriores al 5. Si $v[p] = 0$ en el pueblo p no podemos vender la mercancia de ningún pueblo anterior.

El resto de condiciones se mantienen.

Para este *nuevo problema* se pide:

a) Plantead la ecuación recursiva de programación dinámica que calcule el máximo beneficio que se puede obtener con cualquier secuencia válida de pueblos, indicando cuál debe ser la llamada inicial que se efectúe a dicha función para obtener el valor óptimo.

b) Representad el grafo de dependencias entre las llamadas de la ecuación recursiva para $P = 6$ pueblos, $D = 3$ días y $v[1] = 0$, $v[2] = 0$, $v[3] = 1$, $v[4] = 3$, $v[5] = 3$ y $v[6] = 4$.

c) Indicad cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de vuestra ecuación recursiva que devolviera la *secuencia de decisiones óptima*. Justificad brevemente la respuesta.

d) ¿Sería posible reducir el coste espacial del apartado anterior si el algoritmo iterativo sólo tuviera que devolver *el valor* del máximo beneficio? Si es así, indicad y justificad cuál sería dicho coste.

Puntuación extra: Formalizad el nuevo problema en términos de optimización.

Programación dinámica
13 de mayo de 2009 (Tarde)

Prueba individual

Nombre: **Grupo:**

Problema **50 puntos**

Dados los números naturales N y K, una descomposición de N en K factores positivos es una secuencia n_1, n_2, $\ldots$, n_K, tal que

$$N = \prod_{1 \leq k \leq K} n_k \,.$$

Por ejemplo, si $N = 12$ y $K = 3$ tenemos las siguientes cuatro factorizaciones posibles: $12 \cdot 1 \cdot 1$, $6 \cdot 2 \cdot 1$, $3 \cdot 2 \cdot 2$ y $3 \cdot 4 \cdot 1$.

Deseamos obtener la descomposición de N en K factores de modo que la suma de los factores,

$$\sum_{1 \leq k \leq K} n_k,$$

sea la *mínima* posible. Siguiendo con el ejemplo anterior, la suma de los factores es, respectivamente, $12+1+1 = 14$, $6 + 2 + 1 = 9$, $3 + 2 + 2 = 7$ y $3 + 4 + 1 = 8$, con lo que la descomposición óptima es $3, 2, 2$.

La siguiente ecuación recursiva devuelve el valor de la mínima suma de factores que se puede obtener de la descomposición de n en k factores:

$$S(n,k) = \begin{cases} n, & \text{si } k = 1; \\ k, & \text{si } k > 1 \text{ y } n = 1; \\ \min\limits_{\substack{1 \leq m \leq n: \\ (n \text{ mód } m) = 0}} S(n/m, k-1) + m, & \text{si } k > 1 \text{ y } n > 1, \end{cases}$$

donde la llamada $S(N, K)$ proporciona dicho valor mínimo para la descomposición del número N en K factores. Se pide:

a) Escribe el algoritmo recursivo con memorización asociado a la anterior ecuación.

b) Representa el grafo de dependencias asociado a la ecuación recursiva para $N = 12$ y $K = 3$.

c) Indica cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de la ecuación presentada. Justifica brevemente tu respuesta.

Programación dinámica
13 de mayo de 2009 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

Sea el anterior problema:

****

Dados los números naturales N y K, una descomposición de N en K factores positivos es una secuencia $n_1, n_2, \ldots, n_K$, tal que

$$N = \prod_{1 \leq k \leq K} n_k \,.$$

Por ejemplo, si $N = 12$ y $K = 3$ tenemos las siguientes cuatro factorizaciones posibles: $12 \cdot 1 \cdot 1$, $6 \cdot 2 \cdot 1$, $3 \cdot 2 \cdot 2$ y $3 \cdot 4 \cdot 1$.

Deseamos obtener la descomposición de N en K factores de modo que la suma de los factores,

$$\sum_{1 \leq k \leq K} n_k,$$

sea la *mínima* posible. Siguiendo con el ejemplo anterior, la suma de los factores es, respectivamente, $12 + 1 + 1 = 14$, $6 + 2 + 1 = 9$, $3 + 2 + 2 = 7$ y $3 + 4 + 1 = 8$, con lo que la descomposición óptima es $3, 2, 2$.

La siguiente ecuación recursiva devuelve el valor de la mínima suma de factores que se puede obtener de la descomposición de n en k factores:

$$S(n,k) = \begin{cases} n, & \text{si } k = 1; \\ k, & \text{si } k > 1 \text{ y } n = 1; \\ \min\limits_{\substack{1 \leq m \leq n: \\ (n \bmod m) = 0}} S(n/m, k-1) + m, & \text{si } k > 1 \text{ y } n > 1, \end{cases}$$

donde la llamada $S(N, K)$ proporciona dicho valor mínimo para la descomposición del número N en K factores.

****

Sobre el problema original se pide:

a) Escribid el algoritmo iterativo *con reducción de la complejidad espacial* que permite obtener el valor de la suma de la descomposición óptima.

A continuación, se proponen las siguientes modificaciones en el enunciado del problema de la minimización de la suma de factores resultantes de la descomposición de un número:

1. La descomposición no tiene por qué tener un número fijo K de componentes.
2. Los factores tienen que ser números mayores que 1.

El resto de condiciones se mantienen.
Con las nuevas condiciones y el ejemplo $N = 12$, las siguientes descomposiciones serían válidas: $3 \cdot 2 \cdot 2$, $6 \cdot 2$, $3 \cdot 4$, mientras que la descomposición $12 \cdot 1 \cdot 1$ no lo sería.
Para este *nuevo problema* se pide:

a) Plantead la ecuación recursiva de programación dinámica que calcule la suma de los factores asociados a la descomposición óptima, indicando cuál debe ser la llamada que se efectue a dicha función para obtener el valor óptimo.

b) Representad el grafo de dependencias entre las llamadas de la ecuación recursiva para $N = 12$.

c) Indicad cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de vuestra ecuación recursiva que devolviera *la descomposición óptima*. Justificad brevemente la respuesta.

d) ¿Sería posible reducir el coste espacial indicado en el apartado anterior si hubiera que diseñar un algoritmo iterativo que sólo encontrara *el valor* de la suma de la descomposición óptima? Si es así, indicad cuál sería dicho coste y justificadlo.

Puntuación extra: Formalizad el nuevo problema en términos de optimización.

Primer control parcial de Esquemas Algorítmicos (IG24)

28 de marzo de 2009

1. Divide y vencerás — 5 puntos

Sean un vector a, que contiene n elementos sin repetir ordenados de forma creciente, y un vector b que contiene m elementos (muchos de ellos repetidos) también ordenados de forma creciente. El número de elementos de b es mucho mayor que el de a ($m >>> n$). Se desea conocer, para cada elemento contenido en a, cuantas veces aparece en b. El siguiente ejemplo ilustra una instancia del problema: dados $a = [3, 5, 8, 10]$ y $b = [0, 0, 0, 1, 1, 3, 3, 3, 3, 3, 5, 7, 7, 7, 7, 7, 10, 10, 10, 10]$, la solución sería que el 3 aparece cinco veces, el 5 aparece una vez, el 8 ninguna y el 10 cuatro. Se pide:

a) Diseña un algoritmo que, dados a y b, resuelva el problema mediante la técnica recursiva de *divide y vencerás*. El algoritmo debe ser lo más eficiente posible y, en cualquier caso, el coste temporal del mismo debe ser inferior a $O(nm)$.

   Fijate que lo que se pide es que la *estrategia fundamental* con la que resuelvas el problema sea la de divide y vencerás, no que el algoritmo completo tenga que ser exclusivamente de divide y vencerás. Dependiendo del enfoque elegido, puede que te lleve a diseñar una o varias funciones (de divide y vencerás) que sean empleadas desde el algoritmo principal para devolver los valores pedidos. En cualquier caso, no olvides incluir todo el código necesario para la resolución completa del problema.

b) Indica si el algoritmo presenta un comportamiento uniforme o existe un mejor y un peor caso. Si el comportamiento es uniforme, trata de modificarlo para que ante determinadas instancias pueda finalizar antes su ejecución. Señala en qué condiciones (ante que tipos de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor.

c) Realiza un análisis de la complejidad temporal y espacial del algoritmo presentado, justificando los costes en el mejor y en el peor de los casos.

d) Si el algoritmo propuesto (o alguna de sus funciones) presenta recursividad por cola, elimínala e indica si alguno de los costes calculados en el apartado anterior varía.

2. Búsqueda con retroceso — 5 puntos

Estamos en un país exotico y queremos enviar una postal a un amigo. Para ello, hemos adquirido un lote de sellos de n valores diferentes y que contiene 3 sellos de cada valor. En la oficina de correos nos han indicado que enviar una postal a nuestro amigo tiene un coste C. Además, nos informan de que en las postales del país es obligatorio colocar un número exacto de sellos M, que depende del tipo de postal.

Dados los valores de los sellos del lote, $(v_1, v_2, \ldots, v_n)$, C y M, queremos encontrar una combinación de exactamente M sellos de coste total C, sabiendo que no podemos usar más de tres sellos del mismo valor.

Por ejemplo, si los valores fuesen $v = [1, 2, 3, 5, 8, 10]$, el coste del envío $C = 16$ y tuviese que emplear $M = 5$ sellos, una posible solución podría ser usar un sello de 10, dos de 2 y dos de 1. No serían soluciones válidas usar un sello de 10, otro de 5 y otro de 1 (no son 5 sellos), ni usar uno de 8 y cuatro de 2 (se usan más de tres sellos de un valor).

Queremos diseñar un algoritmo de *búsqueda con retroceso* que resuelva el problema de encontrar una combinación de sellos que cumpla los requisitos indicados.

Se pide que:

a) Describas cómo vas a representar los estados s en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolver el problema.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

Segundo control parcial de Esquemas Algorítmicos (IG24)

16 de mayo de 2009

1. Algoritmos voraces — 3.25 puntos

En una gran fábrica de vehículos las diferentes secciones de la fábrica se hayan distribuidas en distintos edificios. Cuando una sección necesita envíar material a alguna otra (piezas, herramientas,...), utiliza a algún empleado que debe dejar su trabajo para hacer la entrega, con los consiguientes retrasos en la fabricación.

Para evitarlo, la fábrica ha decidido poner en marcha un sistema automatizado de transporte de material, empleando una vagoneta que se traslada por raíles entre las distintas secciones. Quisieran gastarse la menor cantidad de dinero posible en el sistema, cuyo coste depende, básicamente, de la cantidad de raíles que haya que instalar. Como conectar cada sección con todas los demás sería muy costoso, han pensado que cada sección se pueda utilizar como punto intermedio para el traslado de material, de manera que si, por ejemplo, la sección a tiene que enviar material a la sección b no es necesario que haya una vía directa entre las dos, sino que si existe una vía que une a la sección a con la c, una que une la c con la d y otra entre la d y la b, se podría seguir ese trayecto para enviar el material desde a hasta b.

La fábrica está estudiando algunas propuestas de *estrategias voraces* que permiten determinar, a partir del coste que supone colocar raíles entre todos los pares de secciones, $r(a, b)$, *qué vías son las que habría que instalar de forma que el coste de la instalación sea el mínimo* posible garantizando que se van a poder hacer envíos entre cualquier par de secciones. He aquí tres de ellas:

1. Para cada sección estudiamos cuál es la sección con la que le cuesta menos construir un raíl y construimos esos raíles.
2. Partiendo de una sección i al azar, estudiamos con cuál podemos poner una vía lo más barata posible j, y construimos el raíl. A continuación nos desplazamos a j y realizamos el mismo estudio, excluyendo a i, seleccionamos la sección k a la que podemos llegar con menor coste y construimos el raíl entre j y k. Hacemos lo mismo con k (excluyendo los raíles que puedan ir hasta i y hasta j) y así sucesivamente hasta haber unido todas las secciones.
3. Como lo que nos interesa es construir las uniones entre secciones más baratas, lo que haremos es construir en primer lugar la unión más barata de todas, luego la segunda más barata, y continuaremos de esta manera hasta que veamos que no queda ninguna sección sin estar unida, al menos, a otra sección.

Se pide:

a) Como se trata de un problema de optimización, describe qué características debería tener una solución para ser factible, cuál sería la función objetivo y qué debe cumplir una solución factible para ser la óptima.

b) Para cada estrategia propuesta, justifica si encuentra siempre una solución factible al problema y, en caso de hacerlo, si es siempre la óptima (debes demostrar con un contraejemplo los casos en que no esté garantizada la obtención de una solución factible y/u óptima).

c) Para cada estrategia propuesta, señala el coste temporal que supone aplicarla (indica para ello las estructuras de datos que utilizarías en la implementación de un algoritmo lo más eficiente posible que siga la estrategia, describiendo los detalles del mismo que den lugar al coste señalado).

d) Si de entre las estrategias propuestas has detectado más de una que garantice la obtención de la solución óptima, di cuál emplearías. Si no es así y crees que puede haber alguna estrategia correcta alternativa a las propuestas, coméntala, señalándo también su coste (en el caso en que hubiera más de una, cita únicamente la mejor).

2. Programación dinámica — 4 puntos

Una empresa que tiene que promocionar un nuevo producto ha decidido, como parte de la campaña publicitaria, desplazar una furgoneta de la compañía a distintas zonas de la ciudad con muestras para regalar. La campaña durará D días. La furgoneta debe ir cada día a una única zona de la ciudad (la ciudad se ha dividido para la campaña en Z zonas, numeradas, por comodidad, entre 1 y Z). La furgoneta no tiene porqué visitar todas las zonas y puede ir más de una vez a una zona; sin embargo, si un día visita una zona, al día siguiente no puede volver a esa ni a las zonas vecinas (para cada zona z ($1 \leq z \leq Z$), $v(z)$ nos indica cuáles son sus zonas vecinas). Eso sí, la campaña ha de comenzar y acabar en la zona 1.

La promoción del producto que obtiene la compañía en una visita de la furgoneta a la zona z el día d de la campaña ha sido cuantificada en $p(z, d)$. La promoción total conseguida en la campaña es la suma de los valores de las promociones diarias.

Deseamos saber, aplicando la estratégia de *programación dinámica*, qué zonas debe visitar cada día la furgoneta para máximizar la promoción conseguida con la campaña.

1. Con las condiciones anteriormente expuestas:
   a) Formaliza el problema en términos de optimización.
   b) Plantea la ecuación recursiva asociada al hecho de visitar durante D días distintas zonas de la ciudad con las condiciones previamente indicadas para conseguir la máxima promoción del producto. Indica cuál debe ser la llamada que se efectue a dicha función para obtener el valor óptimo.

c) Representa el grafo de dependencias entre llamadas de la ecuación recursiva para el ejemplo $D = 4$, $Z = 5$ y $v[1] = \{2, 4\}$, $v[2] = \{1, 3\}$, $v[3] = \{2, 4\}$, $v[4] = \{1, 3, 5\}$, $v[5] = \{4\}$ (donde, por ejemplo, $v[1] = \{2, 4\}$ indica que la zona 1 tiene como vecinas a las zonas 2 y 4). Indica y razona los costes espacial y temporal con los que un algoritmo iterativo podría efectuar los cálculos para resolver el problema (indica explícitamente, justificando tu respuesta, si es posible efectuar una reducción de la complejidad espacial si sólo nos interesara obtener el valor de la mejor promoción).

2. Una variante del problema sería la siguiente: la empresa se ha dado cuenta que no importa repetir la visita a una zona (o a zonas vecinas) de un día al siguiente, ya que eso permitiría reforzar la campaña en dicha zona. Por otra parte, tampoco es obligatorio empezar y acabar la campaña en una zona concreta.

   ¿Cómo modificarías la formulación del problema para que recogiera este nuevo planteamiento? ¿Cuál sería ahora la ecuación recursiva que resuelve el problema y la llamada que harías a la misma? ¿Con qué coste temporal y espacial se podría resolver?

3. Programación dinámica — 2.75 puntos

Un joven periodista que acaba de ser contratado como becario en una cadena de televisión se ha trazado como objetivo llegar a ser el presentador estrella de la cadena en el menor tiempo posible. Sabe que los periodistas de la cadena se dividen en C categorías, numeradas jerárquicamente, de manera que a los becarios les corresponde la categoría 1 y al mejor presentador la C.

El sistema de promoción interna en la empresa es muy exigente, sólo se puede ascender a la categoría c como mucho desde la categoría $c - p(c)$, donde $p(c)$ devuelve, para cada categoría $2 \leq c \leq C$, el número máximo de categorías anteriores desde las que se puede ascender (si, por ejemplo, $c = 6$ y $p(6) = 2$, eso implica que a la categoría 6 se puede ascender desde la 5 y la $4 (= 6 - 2)$ pero no desde categorías previas). Por otra parte, para conseguir el ascenso a una categoría c desde una categoría anterior válida c', hay que lograr una serie de méritos que cuesta un tiempo $t(c', c)$ reunir. Finalmente, la empresa exige, para llegar a la máxima categoría, haber pasado en total por al menos M categorías (incluyendo la C).

Debes averiguar, empleando la técnica de programación dinámica, por qué categorías debe pasar el periodista, partiendo de becario, para llegar a ser la estrella de la cadena en el menor tiempo posible con las restricciones anteriores. Como ayuda, la siguiente ecuación recursiva de programación dinámica permite resolver el problema de obtener el tiempo mínimo necesario para conseguirlo:

$$T(c, m) = \begin{cases} 0, & \text{si } c = 1 \text{ y } m = 1; \\ +\infty, & \text{si } (c = 1 \text{ y } m > 1) \text{ o } (c > 1 \text{ y } m = 1); \\ \min\limits_{c - p(c) \leq c' < c} T(c', m - 1) + t(c', c), & \text{si } c > 1 \text{ y } m > 1. \end{cases}$$

La llamada $\min\limits_{M \leq m \leq C} T(C, m)$ proporciona dicho valor.

Se pide:

a) Representa el grafo de dependencias para el problema en el que $C = 6$, $M = 4$ y $p = [None, 1, 2, 1, 2, 3]$ (recuerda que $p[1]$ es $None$ porque la categoría de becario es la de entrada en la empresa). Indica qué representa el estado $(5, 3)$ y cómo interpretarías el significado de un arco entre los estados $(3, 2)$ y $(5, 3)$ dentro del grafo.

b) Diseña un algoritmo iterativo que devuelva *las categorías por las que debe pasar el periodista* hasta llegar a estrella empleando el menor tiempo posible.

c) Indica cuál es el coste temporal y espacial del algoritmo diseñado justificando brevemente tu respuesta. ¿Podrías acotar algo más alguno de los anteriores costes si nos dijeran que el valor máximo almacenado en p es 3? Justifícalo.

d) ¿Crees que sería posible reducir el coste espacial si únicamente nos interesará obtener el valor del mínimo tiempo necesario? Si es así, índica cuál sería dicho coste y justifícalo.

Examen convocatoria ordinaria de Esquemas Algorítmicos (IG24)

12 de junio de 2009

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo):
Parte 1 (Preg. 1 y 2) ☐ **Parte 2 (Preg. 3 y 4)** ☐ **Partes 1 y 2** ☐ **Renuncio a la evaluación alternativa** ☐
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Divide y vencerás — 2,5 puntos

Sea un vector a que contiene n elementos, muchos de los cuales están repetidos. Los elementos repetidos se ubican en posiciones contiguas del vector. El número de elementos distintos que contiene a es m ($m <<< n$). Un ejemplo de vector podría ser el siguiente: a = [7, 7, 1, 1, 1, 6, 6, 6, 6, 6, 6, 6, 6, 12, 12, 12, 8, 9, 9, 9, 9, 9].

Has de diseñar un algoritmo de *divide y vencerás* que, dado un vector como el descrito, devuelva los elementos distintos contenidos en a (para el ejemplo anterior, el algoritmo devolvería {7, 1, 6, 12, 8, 9}). Se pìde:

a) Diseña un algoritmo recursivo *lo más eficiente posible* mediante la técnica de divide y vencerás que permita resolver este problema.

b) Analiza y justifica la complejidad temporal y espacial del algoritmo desarrollado en el apartado anterior, indicando si presenta un comportamiento uniforme o existe un mejor y un peor caso.

2. Búsqueda con retroceso — 2,5 puntos

Quisieramos resolver un pasatiempo de un periodico consistente en construir una frase de uso común (un refrán, una máxima, una frase célebre...) formada por M palabras que debemos elegir (y combinar) de entre un conjunto de N palabras posibles ($M < N$).

Con la idea de ayudarnos a resolver el problema con el ordenador nos proporcionan el número M de palabras que forman la frase, una lista l que contiene las N palabras que podemos utilizar y dos funciones, v_1 que nos permite saber si una secuencia (una lista) de palabras puede llegar a formar una frase con sentido, y v_2 que nos permite comprobar si una lista de palabras forma una frase válida (el coste de ejecutar v_1 y v_2 es lineal con el número de palabras que se le pasan).

Por ejemplo, dados $M = 5$ y la lista de palabras l = ['madrugar', 'hombres', 'mucho', 'no', 'todos', 'más', 'iguales', 'temprano', 'por', 'los', 'son', 'amanece'], nuestras funciones se comportarían de la siguiente manera:
v_1(['los', 'temprano','no']) = `False`,
v_1(['por', 'mucho', 'madrugar']) = `True`,
(en el primer caso la construcción no tiene sentido, mientras que en el segundo la construcción sintáctica es correcta de momento, podríamos llegar a tener una frase completa con sentido).
v_2(['por', 'mucho','madrugar']) = `False`,
v_2(['no', 'por', 'mucho', 'madrugar', 'amanece', 'más', 'temprano']) = `True`,
v_2(['todos', 'los', 'hombres', 'son', 'iguales']) = `True`, y
(en el primer caso la frase completa no tiene sentido, en el segundo y el tercero son válidas, pero sólo sería una solución la última, ya que es la única que contiene 5 palabras).

Debes desarrollar un algoritmo de *búsqueda con retroceso* que, dados M, l, v_1 y v_2, devuelva una frase válida de M carácteres. Se pide:

a) Describe cómo vas a representar los estados en la resolución del problema planteado.

b) Escribe un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que realizarías.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

3. Voraces — 1,5 puntos

En un congreso organizado por la universidad tenemos que planificar la distribución en aulas de n conferencias (cada una de las cuales tiene una hora de comienzo y una hora de finalización) de forma que no haya más de una conferencia en la misma aula de forma simultanea. Queremos desarrollar un *algoritmo voraz* que realice una distribución de las conferencias entre aulas *haciendo uso del menor número posible de aulas*. Se pide:

a) Ya que se trata de un problema de optimización, debes indicar las características que posee una *solución factible*, cuál es la función bjetivo que se pretende optimizar y qué debe cumplir una solución factible para ser considerada óptima.

b) Propón una estrategia voraz (descríbela con la mayor precisión posible) que devuelva la solución óptima para el problema ante cualquier entrada.

c) Indica y justifica el coste temporal y espacial de la estrategía (y del algoritmo) propuestos.

4. Programación dinámica **3,5 puntos**

Deseamos colocar una valla de M metros en línea recta para marcar el linde entre dos terrenos. La construcción de la valla la haremos ensamblando, una tras otra, piezas de valla prefabricadas que podemos adquirir en liquidación en una gran superficie. Hay piezas disponibles de n longitudes distintas. De cada tipo de pieza i ($1 \leq i \leq n$) conocemos su longitud, $l(i)$, su precio, $p(i)$, y el número de unidades que quedan en stock, $s(i)$.

Deseamos diseñar un algoritmo mediante la estrategia de *programación dinámica* que determine qué piezas hemos de emplear para costruir una valla con la longitud indicada con el menor coste posible. Se pide:

a) Formaliza el problema en términos de optimización.

b) Diseña una ecuación recursiva que calcule el mínimo coste necesario para construir los M metros de valla, indicando cuál debe ser la llamada que se haga a dicha ecuación para resolver el problema.

c) Diseña un algoritmo iterativo que resuelva el problema, devolviendo el valor del mínimo coste empleando la menor ocupación espacial (en términos asintóticos) posible. Indica y justifica sus costes espacial y temporal.

Examen segunda convocatoria de Esquemas Algorítmicos (IG24)

1 de septiembre de 2009

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo):
Parte 1 (Preg. 1 y 2) ☐ **Parte 2 (Preg. 3 y 4)** ☐ **Partes 1 y 2** ☐ **Renuncio a la evaluación alternativa** ☐
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Divide y vencerás — 2,5 puntos

Hay un juego infantil de azar en el que se van eliminando participantes en sucesivas etapas hasta que queda sólo uno (que es el ganador) y que funciona siguiendo una estrategia de divide y vencerás: Inicialmente, se situan los n participantes en una fila circular de manera que el primero pueda llegar a tocar al segundo, el segundo al tercero y así sucesivamente hasta el n-ésimo participante, que debe poder tocar al primero. En cada etapa se sigue el siguiente procedimiento: se elige al azar a un participante; dicho participante le da una patada en el culo al participante que va tras el, que queda eliminado; a continuación, el siguiente partipante en la fila (el que está detras del que acaban de eliminar) le da una patada al que va tras el, eliminándolo, y así hasta llegar al participante anterior al elegido en el sorteo que, en caso de no haber recibido una patada, se la da al primero, eliminándolo; entonces, sin modificar el orden de los participantes en la fila, se estrecha el círculo para que puedan alcanzarse los participantes y se inicia una nueva etapa (sorteo, patada y eliminación, estrechamiento del círculo), hasta que sólo queda el ganador.

Suponiendo que recibimos un vector a con los nombres de los n participantes iniciales en el orden en que se colocarán en la fila, se pide:

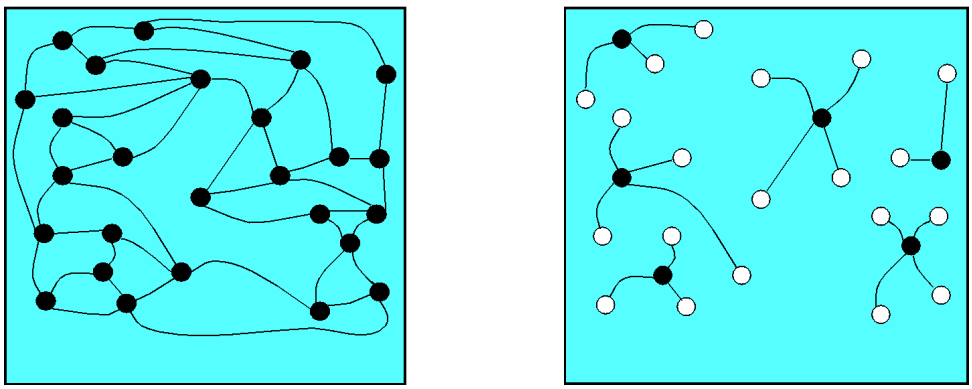
a) Implementa un algoritmo recursivo *mediante la técnica de divide y vencerás* que siga la estratégia anteriormente descrita y que devuelva el nombre del ganador. (Para efectuar los sorteos tienes disponibles la función de python `randrange(m)` (devuelve un valor entero aleatorio entre 0 y m-1) o la función `sample(lista, 1)` (devuelve un vector con un elemento seleccionado al azar de `lista`).)

b) Analiza y justifica la complejidad temporal y espacial del algoritmo desarrollado en el apartado anterior, indicando si presenta un comportamiento uniforme o existe mejor y peor caso.

c) Si el algoritmo propuesto presenta recursividad por cola, elimínala e indica si alguno de los costes calculados en el apartado anterior varía.

2. Búsqueda con retroceso — 2,5 puntos

Un *conjunto dominante* de m vértices en un grafo no dirigido es un subconjunto formado por m vértices del grafo que cumplen que, considerándolos a ellos y a todos los vértices que son adyacentes a ellos, se tienen todos los vértices del grafo. La siguiente figura ilustra este concepto:



(a) (b)

donde (a) es un ejemplo de grafo no dirigido y (b) un conjunto dominante de 6 vértices sobre dicho grafo en el que los vértices sombreados son los que forman el conjunto dominante y los no sombreados son los adyacentes a estos (como puede observarse, entre los sombreados y los no sombreados están considerados todos los vértices del grafo original).

Dado un grafo no dirigido $G = (V, E)$ y un valor m, diseña un algoritmo de *búsqueda con retroceso* que encuentre un conjunto dominante compuesto por exactamente m vértices. Para ello se pide que:

a) Describas cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial qué harías para resolverlo.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Crees que es posible mejorar la eficiencia del algoritmo que has desarrollado introduciendo alguna variable o estructura de datos adicional o haciendo algún tipo de preproceso? Si es así, describe con claridad la mejora o mejoras que introducirías en el algoritmo e indica si ello afectaría a los costes previamente calculados.

3. Voraces **1,5 puntos**

Tenemos que recoger la cosecha en n campos, cada uno con un cultivo diferente. La recolección de un campo exige un día completo de trabajo. Para cada campo i ($1 \leq i \leq n$) conocemos el beneficio $b[i]$ que nos reportará la venta de su cosecha, así como el número de días máximo $d[i]$ que, a partir de hoy, tardaría en estropearse el cultivo si no lo cosechamos.

Queremos desarrollar un *algoritmo voraz* que indique qué campos y en qué días debemos recolectar cada uno de ellos para maximizar el beneficio total obtenido. Se pide:

a) Ya que se trata de un problema de optimización, debes indicar las características que posee una *solución factible*, cuál es la función objetivo que se pretende optimizar y qué debe cumplir una solución factible para ser considerada óptima.

b) Propón una estrategia voraz (descríbela con precisión) que devuelva la solución óptima para el problema ante cualquier entrada.

c) Indica y justifica el coste temporal y espacial de la estrategía propuesta.

d) ¿La estrategia que has desarrollado permite resolver también el problema si la recolección de cada campo requiriera un número variable de días, dependiendo de cada uno de los campos? Justifícalo adecuadamente.

4. Programación dinámica **3,5 puntos**

Disponemos de una serie de R recursos, $r_1, r_2, \ldots, r_R$, que podemos asignar a $d_1, d_2, \ldots, d_D$ actividades, con $R \geq D$. Podemos asignar una serie de recursos *contiguos* a una misma actividad. Si asignamos los recursos $r_i, r_{i+1}, \ldots, r_j$ a la actividad d_l, asignaremos los recursos $r_{j+1}, r_{j+2}, \ldots, r_k$ a la actividad d_{l+1}, donde $i \leq j < k$. No podemos dejar recurso alguno sin asignar a alguna actividad y cada actividad ha de recibir al menos uno de los recursos. La asignación de $r_i, r_{i+1}, \ldots, r_j$ a la actividad d_l ofrece un beneficio $v(i, j, l)$. Deseamos encontrar la asignación de recursos a actividades que proporciona el máximo beneficio.

Se pide:

a) Formaliza el problema en términos de optimización,

b) plantea la ecuación recursiva de *programación dinámica* que calcula el mayor beneficio resultante de cualquier combinación válida de asignaciones, indicando cuál debe ser la llamada recursiva que se efectue a dicha ecuación para resolver el problema,

c) representa el grafo de dependencias entre las llamadas recursivas asociadas a la anterior ecuación para la instancia $R = 6$ y $D = 3$, y

d) diseña un algoritmo iterativo de *programación dinámica* lo más eficiente posible que recorra el anterior grafo y devuelva el beneficio máximo que puede obtenerse y las asignaciones que dan lugar a ese beneficio, indicando y justificando sus costes espacial y temporal.

Análisis de algoritmos y Estructuras de datos
21 de febrero de 2008 (Mañana)

Prueba individual

Nombre: **Grupo:**

Análisis de algoritmos

Dado un vector de enteros positivos a y un valor x, los dos siguientes algoritmos informan de si existen en el vector al menos dos números cuya suma es menor o igual que dicho valor:

```
def almenosdossuman_1(a,x):
  for i in xrange(len(a)):
    if a[i]<x:
      for j in xrange(i+1,len(a)):
        if a[i]+a[j]<=x:
          return True
  return False
```

```
def almenosdossuman_2(a,x):
  b=sorted(a)
  if b[0]+b[1]<=x:
    return True
  return False
```

a) Indica y justifica cuál es el coste temporal y espacial asintótico de ambos, distinguiendo entre el mejor y el peor de los casos si los hubiera. Si tuvieras que elegir uno de los dos algoritmos para resolver el problema, ¿con cuál te quedarías? Justifica tu respuesta.

b) ¿Se te ocurre algún método alternativo que garantizara un tiempo máximo menor que el de cualquiera de los dos algoritmos anteriores? Si es así, descríbelo brevemente y justifica su coste temporal.

Estructuras de datos (min-heap)

a) De los siguientes vectores, ¿cuáles podrían corresponderse con un min-heap y cuáles no? Justífica tu respuesta.

- $[1, 30, 7, 40, 50, 10]$
- $[1, 7, 9, 3, 5, 11]$
- $[1, 4, 6, 7, 5, 3, 2]$

b) Dado el vector $[20, 33, x, 100, 40, 39]$, indica en que rango de valores debería estar el elemento x para que el vector representara un min-heap.

c) El vector $a = [10, 22, 18, 30, 25, 40, 60, 50]$ representa un min-heap. Indica cuál sería su estado tras ejecutar la operación $a.extract_min()$: dibuja el árbol antes y después de la operación.

Prueba colectiva

Grupo:
Asistentes:

Estructuras de datos y costes

Deseamos incorporar a la estructura de datos grafo (representado mediante conjuntos de adyacencia con inversa) una función adicional para combinar grafos, *difsim*(G_1, G_2), que reciba dos grafos $G_1 = (V_1, E_1)$ y $G_2 = (V_2, E_2)$, con $V_1 = V_2$, y devuelva un nuevo grafo $G = (V, E)$, donde $V = V_1 = V_2$ y el conjunto de aristas E debe contener las aristas que están en E_1 pero no en E_2 y las que están en E_2 pero no en E_1.

Debes describir brevemente (y con claridad) como implementarías esta nueva operación, teniendo en cuenta que puedes utilizar los métodos ya disponibles en la interfaz de la estructura grafo, y de forma que sea lo más eficiente posible. Debes indicar, justificándolos, los costes temporal y espacial asociados a la función.

Análisis de algoritmos y Estructuras de datos
21 de febrero de 2008 (Tarde)

Prueba individual

Nombre: **Grupo:**

Análisis de algoritmos

Dado un vector de enteros positivos a y un valor x, los dos siguientes algoritmos calculan cuál es el máximo número de elementos del vector que, sumados, dan un resultado menor o igual que x:

```
def cuantossuman_1(a,x):
    suma=0
    b=a[:]
    for k in xrange(len(a)):
        suma+=min(b)
        if suma>x:
            return k
        else:
            b.remove(min(b))
    return len(a)
```

```
def cuantossuman_2(a,x):
    suma=0
    b=sorted(a)
    for k in xrange(len(a)):
        suma+=b[k]
        if suma>x:
            return k
    return len(a)
```

Indica y justifica cuál es el coste temporal y espacial asintótico de ambos, distinguiendo entre el mejor y el peor de los casos si los hubiera (señala ante qué entradas se comportarían en el mejor caso y cuando en el peor caso). Si tuvieras que elegir uno de los dos algoritmos para resolver el problema, ¿con cuál te quedarías? Justifica tu respuesta.

Estructuras de datos (MFSET)

Dado el siguiente MFSET:

	0	1	2	3	4	5	6	7	8	9
S	0	0	0	2	2	5	5	6	7	5

a) Dibuja cada uno de los árboles que representan los subconjuntos disjuntos que contiene S.

b) Indica sobre la siguiente tabla el contenido de la estructura después de realizar de forma consecutiva cada una de las operaciones que se indican a continuación, teniendo en cuenta que se emplean los heurísticos de *unión por rango* y *comprensión de caminos*. En la unión, en caso de que la altura de los dos árboles sea idéntica, el árbol del segundo parámetro será hijo del del primero. (Si lo prefieres, puedes rellenar únicamente los valores de las casillas que se modifican desde la anterior operación):

	0	1	2	3	4	5	6	7	8	9
S	0	0	0	2	2	5	5	6	7	5
$S.find(3)$										
$S.merge(9,1)$										
$S.find(4)$										

Prueba colectiva

Grupo:
Asistentes:

Estructuras de datos y costes

Deseamos incorporar a la estructura de datos diccionario una función adicional para combinar diccionarios, $unir(D_1, D_2)$, que reciba dos diccionarios D_1 y D_2 y devuelva un nuevo diccionario D que contenga todos los elementos de los dos diccionarios. En el caso en que hubiera elementos en D_1 y D_2 con la misma clave, sólo debería almacenarse en D el elemento (obviamente, una vez) si además de la clave coinciden sus valores en D_1 y D_2; si sólo coincidieran las claves, no se almacenaría ninguno de los dos elementos.

Debes describir brevemente (y con claridad) como implementarías esta nueva operación, teniendo en cuenta que puedes utilizar los métodos ya disponibles en la interfaz de la estructura, y de forma que sea lo más eficiente posible. Debes indicar, justificándolos, los costes temporal y espacial asociados a la función.

Divide y vencerás
12 de marzo de 2008 (Mañana)

Prueba individual

Nombre: **Grupo:**

1. Algoritmo *partition* (traza)

Haz la traza del comportamiento del algoritmo de partición de un vector empleado en *quicksort* (pág. 5-36) para la llamada: *partition*`([13, 4, 5, 8, 12, 9, 7, 20, 22, 6, 3, 17, 15, 14, 14], 7, 15)`.

Sobre la siguiente tabla debes detallar el valor de las variables i y j y el estado del vector justo antes de entrar en el bucle, después de cada una de las iteraciones del bucle y al final de la ejecución del algoritmo (si lo prefieres, puedes rellenar únicamente los valores de las casillas que se modifican entre una iteración y la siguiente). Marca también de forma especial la posición del pivote antes y después de la ejecución del algoritmo:

		vector														
j	i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		13	4	5	8	12	9	7	20	22	6	3	17	15	14	14

2. Planteamiento y análisis de un problema

Dada f, una función creciente en un intervalo cerrado de valores enteros $[a, b]$ ($a < b$, $a, b \in \mathbb{Z}$) queremos averiguar si la función llega a valer 0 para algún punto incluido en dicho intervalo (si $f(y) = 0$, donde $y \in \mathbb{R}$) y, en tal caso, cuál sería el valor entero z ($z \in \mathbb{Z}$, $a \leq z < b$) anterior a dicho punto ($z \leq y < z + 1$).

Describe con la mayor claridad posible cómo aplicarías la estrategia de divide y vencerás para abordar y resolver el problema de la forma más eficiente posible. Indica asimismo cuál crees que sería el coste temporal asociado a la estrategia planteada, dando una breve justificación del mismo.

Divide y vencerás
12 de marzo de 2008 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema

Dada f, una función creciente en un intervalo cerrado de valores enteros $[a, b]$ ($a < b$, $a, b \in \mathbb{Z}$) queremos averiguar si la función llega a valer 0 para algún punto incluido en dicho intervalo (si $f(y) = 0$, donde $y \in \mathbb{R}$) y, en tal caso, cuál sería el valor entero z ($z \in \mathbb{Z}$, $a \leq z < b$) anterior a dicho punto ($z \leq y < z + 1$).

a) Diseñad un algoritmo recursivo *mediante la técnica de divide y vencerás* que devuelva el valor z (en el caso en que no existiera, deberá devolver un valor especial). El algoritmo desarrollado debe ser lo más eficiente posible, tanto desde el punto de vista de la complejidad temporal como de la espacial.

b) El algoritmo que habéis desarrollado ¿tiene mejor y peor caso o se comporta de manera uniforme? Describid con claridad el mejor y el peor caso si los hubiera y realizad los correspondientes análisis de complejidad temporal y espacial justificando los resultados obtenidos.

c) ¿Presenta vuestro algoritmo recursión por cola? Si es así, eliminadla e indicad si ello afecta a alguno de los costes asintóticos previamente calculados y en que sentido.

Divide y vencerás
12 de marzo de 2008 (Tarde)

Prueba individual

Nombre: **Grupo:**

1. Algoritmo *convex_min* (traza de la recursión)

Dado el vector

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
a	22	20	14	13	9	8	7	6	5	8	12	14	15	17

escribe todas las llamadas recursivas que se generan a partir de la ejecución de *convex_min*(*a*) (la versión de la página 5-31). Para cada llamada debes señalar los valores de los parámetros i y k en la función recursiva. ¿Cúal es el valor devuelto por el algoritmo?

2. Planteamiento y análisis de un problema

Disponemos de un vector a que contiene valores enteros y en el que se cumple que todos los valores impares del vector son consecutivos y aparecen ordenados de menor a mayor y agrupados en una parte del vector, y lo mismo sucede con los valores pares, que se agrupan en la otra parte. Un ejemplo de vector sería el siguiente: $a = [7, 9, 11, 13, 15, 10, 12, 14, 16, 18, 20, 22, 24]$. Queremos obtener la suma de los valores pares y la de los valores impares del vector, sabiendo que en un rango consecutivo de n valores ordenados (pares o impares) entre i y j (a_i es el primer elemento del rango y a_j el último) la suma de todos los elementos del rango es $(a_i + a_j) * (j - i + 1)/2$.

Describe con la mayor claridad posible cómo aplicarías la estrategia de divide y vencerás para abordar y resolver el problema de la forma más eficiente posible. Indica asimismo cuál crees que sería el coste temporal asociado a la estrategía planteada, dando una breve justificación del mismo.

Divide y vencerás
12 de marzo de 2008 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema

Disponemos de un vector a que contiene valores enteros y en el que se cumple que todos los valores impares del vector son consecutivos y aparecen ordenados de menor a mayor y agrupados en una parte del vector, y lo mismo sucede con los valores pares, que se agrupan en la otra parte. Un ejemplo de vector sería el siguiente: $a = [7, 9, 11, 13, 15, 10, 12, 14, 16, 18, 20, 22, 24]$. Queremos obtener la suma de los valores pares y la de los valores impares del vector, sabiendo que en un rango consecutivo de n valores ordenados (pares o impares) entre i y j (a_i es el primer elemento del rango y a_j el último) la suma de todos los elementos del rango es $(a_i + a_j) * (j - i + 1)/2$.

a) Diseñad un algoritmo recursivo *mediante la técnica de divide y vencerás* que devuelva la suma de los valores impares y la de los pares (podéis asumir que los valores impares aparecen en el vector antes que los pares y que como mínimo el vector contendra un valor par y uno impar). El algoritmo desarrollado debe ser lo más eficiente posible, tanto desde el punto de vista de la complejidad temporal como de la espacial.

   Tened en cuenta que la función recursiva que desarrolléis no tiene porque calcular directamente las sumas de los grupos, sino que puede devolver algún resultado que permita, una vez acabado el cálculo recursivo (y, por tanto, fuera de la recursión), obtener los valores pedidos. En cualquier caso, la estrategia principal deberá ser la de divide y vencerás y deberéis incluir todas las instrucciones necesarias para calcular el resultado pedido.

b) El algoritmo que habéis desarrollado ¿tiene mejor y peor caso o se comporta de manera uniforme? Describid con claridad el mejor y el peor caso si los hubiera y realizad los correspondientes análisis de complejidad temporal y espacial justificando los resultados obtenidos.

c) Adaptad vuestro algoritmo para que (si no lo hace ya) contemple la posibilidad de que en el vector pudieran estar al principio tanto los pares como los impares y que, además, el vector pudiera no contener valores de un tipo (en tal caso, la suma de ese tipo debería ser 0). En ningún caso deberán empeorar los costes del algoritmo con respecto al anterior.

Búsqueda con retroceso
3 de abril de 2008 (Mañana)

Prueba individual

Nombre: **Grupo:**

Traza problema de la suma del subconjunto **50 puntos**

En el problema de la suma del subconjunto disponemos de N objetos con pesos $w_1, w_2, \ldots, w_N$ y de una mochila con capacidad para soportar una carga W. Deseamos cargar la mochila con una selección arbitraria de objetos cuyo peso sea *exactamente* W. La función `subset_sum` (*la versión refinada del algoritmo de la página 6-23*) permite encontrar una solución factible para el problema aplicando la técnica de búsqueda con retroceso:

```
def subset_sum(w, W):
  w = OffsetArray(sorted(w))
  sum_w = OffsetArray([0]* (len(w)+1))
  sum_w[len(w)] = w[len(w)]
  for i in xrange(len(w)-1, 0, -1): sum_w[i] = sum_w[i+1] + w[i]
  x = OffsetArray([0] * len(w))

  def backtracking(i, W):
    if W == 0:
      for j in xrange(i, len(w)+1): x[j] = 0
      return x
    elif i <= len(w):
      for x[i] in 1, 0:
        if W-x[i]*w[i] >= 0 and sum_w[i+1] >= W-x[i]*w[i]:
          found = backtracking(i+1, W-x[i]*w[i])
          if found != None: return found
        else:
          return None
    return None

  return backtracking(1, W)
```

Realiza una traza de la ejecución del método para una mochila con capacidad para $W = 11$ y $N = 6$ objetos de pesos $w = [7, 1, 9, 2, 6, 4]$.

Debes dibujar el árbol de estados con tan sólo aquellos estados *que se exploran* durante el proceso de búsqueda y señalando el valor del peso total que queda por rellenar en la mochila al lado de cada estado. Incluye los estados que se estudian pero se descartan por no superar la condición de ser prometedores y márcalos de forma especial. Indica también claramente cuál es el resultado devuelto por el algoritmo. ¿Qué objetos son los seleccionados?

Búsqueda con retroceso
3 de abril de 2008 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

El problema del coloreado de un grafo con un máximo de k colores consiste en asignar un color a cada uno de los vértices de un grafo no dirigido de modo tal que dos vértices adyacentes no tengan asignado el mismo color y no se utilicen más de k colores.

Dado un grafo no dirigido $G = (V, E)$ y un número máximo de colores k, debes diseñar un algoritmo de *búsqueda con retroceso* que devuelva como resultado el número de color (entre 1 y k) asignado a cada uno de los vértices para que se cumplan las condiciones del problema.

Se pide que:

a) Describas cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolverlo.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

Si os sobra tiempo, podríais intentar responder también a las siguientes **cuestiones adicionales:**

1. ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

2. En el problema original del coloreado de un grafo no se proporciona un número máximo de colores k, se propone la búsqueda del coloreado que emplea *el menor número de colores*. ¿Serías capaz de adaptar tu algoritmo para que resolviera dicho problema?

Búsqueda con retroceso
3 de abril de 2008 (Tarde)

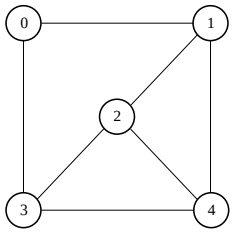
Prueba individual

Nombre: **Grupo:**

Traza problema de la existencia de un ciclo hamiltoniano **50 puntos**

En el problema de la existencia de un ciclo hamiltoniano, dado un grafo $G = (V, E)$, deseamos encontrar un camino que parta de un vértice, termine en el mismo vértice y visite todos los vértices del grafo una sola vez (excepto en el caso del vértice de partida, que deberá coincidir con el de llegada). El algoritmo `hamiltonian_cycle` (*la versión refinada de la página 6-32*) resuelve el problema aplicando la técnica de búsqueda con retroceso.

Realiza una traza de la ejecución del método para la llamada `hamiltonian_cycle(G)`, donde G es el grafo que se muestra a continuación. Para la traza, considera que el vértice de partida seleccionado aleatoriamente es el 0 y que la función $G.succ(v), \forall v \in G.V$, devuelve *siempre* una lista con los vértices sucesores de v ordenados de menor a mayor valor númerico).



Debes dibujar el árbol de estados con tan sólo aquellos estados *que se exploran* durante el proceso de búsqueda. Incluye los estados que se estudian pero se descartan por no superar la condición de ser prometedores y márcalos de forma especial. Haz lo mismo con los no factibles. Indica también claramente cuál es el resultado devuelto por el algoritmo.

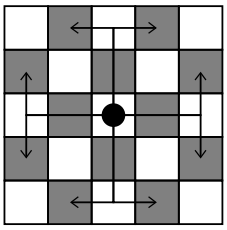
Búsqueda con retroceso
3 de abril de 2008 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

El caballo de ajedrez puede, desde una casilla del tablero, desplazarse a una cualquiera de las ocho casillas señaladas con una flecha en la siguiente figura:



Debes diseñar un algoritmo de *búsqueda con retroceso* que, dada una casilla de partida en la que se sitúa un caballo, encuentre un recorrido en un tablero de ajedrez de $n \times n$ con el caballo de modo que visite todas las casillas, pero que ninguna la visite más de una vez.

Se pide que:

a) Describas cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolverlo.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

Si os sobra tiempo, podríais intentar responder también a las siguientes **cuestiones adicionales:**

1. ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

2. En el problema original del recorrido del caballo no se facilita ninguna casilla de partida (en algunos casos salir de una casilla concreta podría dar lugar a que no hubiera solución). ¿Serías capaz de adaptar tu algoritmo para que resolviera dicho problema?

Algoritmos voraces
21 de abril de 2008 (Mañana)

Prueba individual

Nombre: **Grupo:**

Selección y adaptación de un algoritmo voraz **35 puntos**

Un organismo regional encargado del control de epidemias de origen animal tiene una estimación del número de horas que tardaría en propagarse un brote de gripe aviaria entre cada par de granjas de la región dedicadas a la cria y venta de pollos.

Por ejemplo, si en la región hay 5 granjas (A, B, C, D y E), se sabe que desde la granja A se puede propagar la gripe a la granja B en 48 horas, a D en 30 horas y a E en 56 horas; desde B se puede propagar la enfermedad a A en 60 horas, a C en 72 horas y a D en sólo 24 horas; desde C no se puede propagar a ninguna otra granja; desde D sólo se puede propagar a C en 10 horas y desde E sólo a D en 15 horas. En los casos no indicados, no hay posibilidad de extensión de la epidemia.

A partir de los tiempos de propagación entre pares de granjas, dadas también las granjas en las que se detecta el inicio de un brote vírico (que podría surgir *simultaneamente* en más de una a la vez) y proporcionado un tiempo máximo estimado de respuesta, los responsables del organismo desean disponer de un programa informático capaz de determinar qué granjas se han contagiado.

Siguiendo con el anterior ejemplo, si la epidemía se ha iniciado en B y el tiempo de respuesta es de 48 horas, el programa devolvería $\{B, D, C\}$: obviamente, B ya está infectado, D se contagiaría directamente de B a las 24 horas (≤ 48) y C se contagiaría a través de D en $24 + 10 = 34$ horas (≤ 48). En otro caso, con A y E como focos simultaneos de la infección y un tiempo de respuesta de 30 horas, la solución devuelta sería $\{A, E, D, C\}$.

Se pide:

a) Indica qué estructura de datos elegirías para representar las granjas y tiempos de propagación de la gripe entre granjas. Representa gráficamente el contenido de la estructura de datos escogida para los datos del ejemplo del segundo párrafo.

b) Alguno de los algoritmos que hemos estudiado en el tema permite abordar problemas similares a este. Es posible que haya que adaptarlo para que resuelva aspectos concretos de este problema. Indica qué algoritmo emplearías y qué modificaciones concretas le aplicarías para resolver el problema (no es necesario que las codifiques, pero si que las describas con precisión).

c) ¿Qué costes temporales presenta la versión del algoritmo que has descrito previamente en el mejor y el peor de los casos?

Algoritmos voraces
21 de abril de 2008 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Análisis de la solución voraz de un problema — 65 puntos

El problema de la *cobertura del conjunto* se define con un conjunto A y una familia $F = \{F_1, F_2, ..., F_N\}$ de N subconjuntos de A cuya unión coincide con A. El objetivo es seleccionar el menor número de subconjuntos de F tal que su unión contenga a todos los elementos de A. Un ejemplo ayudará a entender el problema. Sea $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$ y sea $F = \{F_1, F_2, F_3, F_4\}$, donde $F_1 = \{1, 2, 3, 4, 5\}$, $F_2 = \{4, 5, 6, 9\}$, $F_3 = \{1, 2, 7, 8, 9\}$ y $F_4 = \{6, 7, 8\}$. Nótese que A es la unión de todos los conjuntos de F. La unión $F_1 \cup F_2 \cup F_3$ es igual a A, así que $\{F_1, F_2, F_3\}$ es una cobertura del conjunto y tiene talla 3. Como no hay otra cobertura posible con un número menor de elementos de F, se trata de una cobertura óptima. Me han propuesto el siguiente algoritmo voraz para resolver el problema:

```
 1 def greedy_set_cover(A, F):
 2   AA = A.copy()  # A.copy() crea una copia del conjunto.
 3   FF = F.copy()
 4   C = set()
 5   while len(AA) > 0:
 6     # Selecciono el subconjunto Fi de FF que añade más elementos de AA.
 7     inAA = -1
 8     for Fi in FF:
 9       num = len(Fi.intersection(AA))
10       if num > inAA:
11         inAA = num
12         S = Fi
13     FF.remove(S)
14     C.add(S)
15     # Actualizo AA, que es el conjunto de elementos de A que aún no he añadido.
16     AA.difference_update(S)
17   return C
```

He aquí un ejemplo de uso de esa función:

```
1 A = set(xrange(1,10))
2 F = set([frozenset([1,2,3,4,5]), frozenset([4,5,6,9]), frozenset([1,2,7,8,9]), frozenset([6,7,8])])
3 print greedy_set_cover(A, F)
```

Responde *razonadamente* a las siguientes preguntas:

a) ¿Encuentra siempre una solución factible?

b) ¿Encuentra siempre una solución óptima?

c) ¿Qué coste temporal presenta en el mejor y el peor caso?

d) ¿Qué coste espacial presenta?

Algoritmos voraces
21 de abril de 2008 (Tarde)

Prueba individual

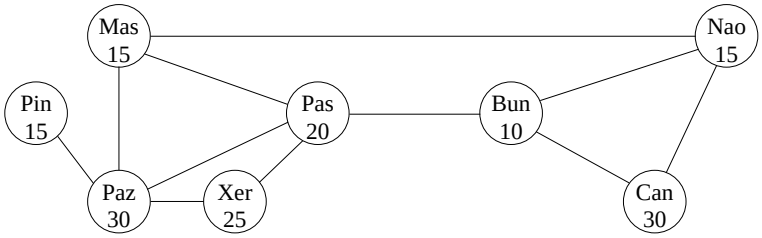
Nombre: **Grupo:**

Dijkstra **35 puntos**

Tenemos un mapa de carreteras con ciudades que podemos modelar con un grafo. Un coche fúnebre debe transportar un cadáver desde una ciudad s a otra ciudad t. La legislación obliga a pagar un impuesto funerario en cada *ciudad* por la que pase el coche (incluyendo las ciudades inicial y final). Nos gustaría saber el mínimo precio con el que podemos ir de s a t, así que hemos pensado en aplicar el algoritmo de Dijkstra (los impuestos que hemos de pagar son siempre cantidades no negativas), pero nos hemos encontrado con una seria dificultad: ¡el impuesto grava el tránsito por la ciudad, no por la carretera! ¿Qué hemos de hacer para poder utilizar Dijkstra en este problema?

Cuando lo tengas claro, cuéntamelo. Puede que tengas que modificar el algoritmo de Dijkstra y puede que no. Puede que tengas que modificar el grafo y puede que no. Hagas lo que hagas, cuéntamelo.

A continuación haz una traza de Dijkstra (posiblemente modificado) sobre el siguiente mapa (posiblemente modificado) al tratar de ir de Mas a Bun (en cada ciudad hemos puesto el nombre de la ciudad y el impuesto funerario asociado):



Puedes presentar la traza gráficamente mostrando la evolución del algoritmo mediante figuras (como hicimos en clase al presentar el ejemplo de Mallorca) o bien mostrando el contenido de las principales estructuras de datos del algoritmo. No es necesario que detalles cada paso de la traza, pero sí, como mínimo, debes indicar el contenido final del diccionario D y la solución devuelta por el algoritmo.

Algoritmos voraces
21 de abril de 2008 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Análisis de la solución voraz de un problema — 65 puntos

He de matricularme de K créditos para acabar mis estudios. La universidad me ofrece N asignaturas (numeradas entre 1 y N), cada una con una carga expresada en créditos y que denotamos con $c_1, c_2, \ldots, c_N$, respectivamente. Consultando las actas del pasado curso he confeccionado una tabla con la tasa de aprobados en cada asignatura. Con $a_1, a_2, \ldots, a_N$ (números entre 0 y 1) denoto la tasa de aprobados en cada asignatura. Ese valor constituye para mí una estimación de la probabilidad con la que aprobaré la asignatura en caso de cursarla. Una matrícula es un conjunto de asignaturas $\{x_1, x_2, \ldots, x_m\}$ (es decir, $x_i \in [1..N]$ y $x_i \neq x_j, 1 \leq i \neq j \leq m$) y entiendo por matrícula óptima la que hace máximo el valor de $\sum_{1 \leq i \leq m} a_{x_i} c_{x_i}$. Trabajaremos bajo dos supuestos:

Uno Que mi pereza llega al extremo de no querer cursar ni un crédito más de los K que necesito para completar la carrera.

Dos Que mi deseo de aprender llega al punto de aceptar cursar algunos créditos de más, pero sólo si son el excedente al que me obliga la asignatura con la que completo los K créditos que he de cursar.

Me propongo escoger las asignaturas siguiendo uno de estos dos algoritmos:

1. Me voy matriculando de asignaturas en orden de mayor a menor valor de a_{x_i}.
2. Me voy matriculando de asignaturas en orden de mayor a menor valor del producto $a_{x_i} c_{x_i}$.

Responde *razonadamente* a las siguientes cuestiones:

a) ¿Obtendremos una matrícula óptima en el supuesto **Uno** con el algoritmo 1?

b) ¿Obtendremos una matrícula óptima en el supuesto **Uno** con el algoritmo 2?

c) ¿Obtendremos una matrícula óptima en el supuesto **Dos** con el algoritmo 1?

d) ¿Obtendremos una matrícula óptima en el supuesto **Dos** con el algoritmo 2?

e) ¿Qué coste temporal presentan los algoritmos 1 y 2 en el supuesto **Uno**? (Expresa una cota tan ajustada como te sea posible.)

f) ¿Qué coste temporal presentan los algoritmos 1 y 2 en el supuesto **Dos**? (Expresa una cota tan ajustada como te sea posible.)

Programación dinámica
30 de abril de 2008 (Mañana)

Prueba individual

Nombre: **Grupo:**

Problema de la mochila discreta **50 puntos**

Dada una mochila con capacidad para cargar $W = 4$ unidades de peso y $N = 6$ objetos que podemos cargar en ella, con pesos $w = [3, 4, 2, 1, 3, 1]$ y valores $v = [20, 30, 15, 9, 25, 5]$, se pide:

a) Dibuja, aprovechando la siguiente cuadrícula, el *grafo extendido de dependencias entre estados* asociado a dicha instancia del problema (utiliza las cajas que necesites, tachando el resto, y traza los arcos entre estados correspondientes al grafo para el problema propuesto):

(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)

b) Haz una traza del algoritmo `knapsack` (pág 7-22), el que permite recuperar la secuencia óptima de objetos. Para ello, escribe arriba de cada uno de los estados del grafo del apartado anterior el valor que el algoritmo almacena en el diccionario `V` y debajo del estado el valor que se almacena en el diccionario de punteros hacia atrás para el estado (alternativamente, puedes utilizar flechas tal y como se hacía en la figura 7-16).

c) ¿Cuál es el valor de la solución óptima? ¿En qué estado se encuentra? ¿Qué resultado devuelve el algoritmo? ¿Qué significa este último resultado?

Programación dinámica
30 de abril de 2008 (Mañana)

Prueba colectiva

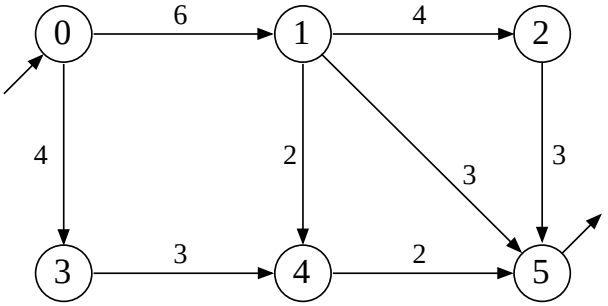
Grupo:
Asistentes:

Recuperación voraces **+25 puntos (añadir a la puntuación de voraces)**

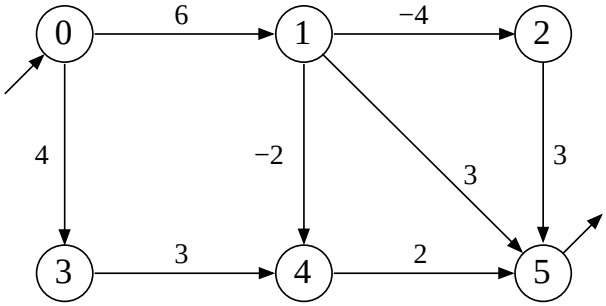
Conoceis dos algoritmos capaces de calcular el camino más corto entre dos vértices en un grafo dirigido ponderado: el algoritmo de Dijkstra (voraz) y el del camino más corto en un grafo acíclico (de programación dinámica).

Para cada uno de los siguientes grafos, en los que se quiere obtener el camino más corto entre el vértice 0 y el vértice 5, escribe debajo **"Ninguno", "Dijkstra", "PD", "Los dos"** para indicar qué algoritmo o algoritmos podrían aplicarse para resolver el problema. En el caso en que se pudieran emplear los dos, señala cuál consideras más adecuado:

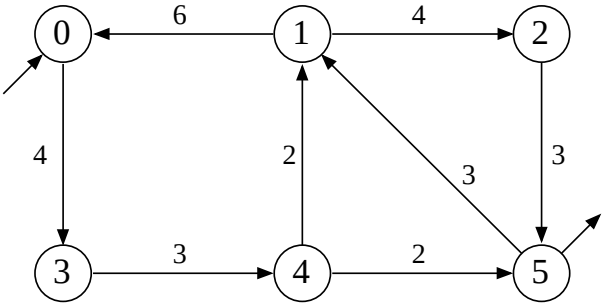
a)



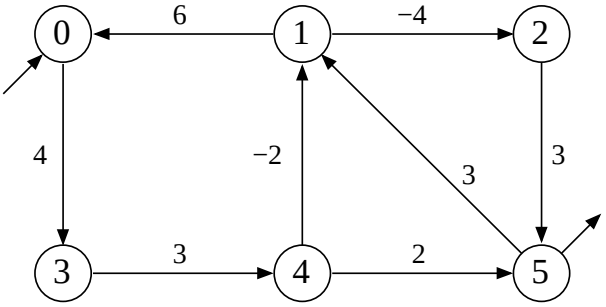
b)



c)



d)



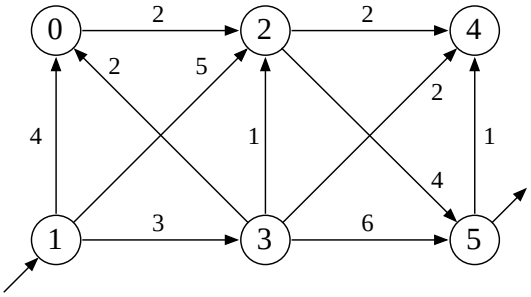
Programación dinámica
30 de abril de 2008 (Tarde)

Prueba individual

Nombre: **Grupo:**

Problema del camino más corto en un grafo acíclico **50 puntos**

Debes encontrar cuál es el camino más corto que empieza en el vértice $s = 1$ y acaba en el vértice $t = 5$ en el siguiente grafo:



Para ello, se pide:

a) Dibuja el grafo ordenado topológicamente.

b) Haz una traza del algoritmo `dag_shortest_path` (pág 7-34), el que permite recuperar el camino óptimo. Para ello puedes, bien utilizar como base el grafo obtenido en el apartado anterior de forma similar a cómo se hacía en el apartado (g) de la figura 7.23 (pág 7-35), bien rellenar los diccionarios D y B. *No* se pide una traza iteración a iteración, basta con que se muestre el contenido final de las dos estructuras citadas.

c) ¿Cuál es el valor de la solución óptima? ¿En qué estado se encuentra? ¿Qué resultado devuelve el algoritmo? ¿Qué significa este último resultado?

Programación dinámica
30 de abril de 2008 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Recuperación voraces **+25 puntos (añadir a la puntuación de voraces)**

La representación formal del problema de la mochila *sin fraccionamiento* sabemos que es la siguiente:

- Conjunto de soluciones factibles:

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in \{0,1\}^N \middle| \sum_{1 \leq i \leq N} x_i w_i \leq W \right\}.$$

- Optimización y función objetivo:

$$\arg \max_{(x_1, x_2, \ldots, x_N) \in X} \sum_{1 \leq i \leq N} x_i v_i.$$

¿Eres capaz de indicar qué variaciones se han introducido en el *enunciado original* del problema para cada uno de los siguientes casos en los que la parte de la formalización que se modifica aparece en un recuadro?:

a) Modificación en la definición del conjunto de soluciones factibles:

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in \{0,1\}^N \middle| \boxed{W' \leq} \sum_{1 \leq i \leq N} x_i w_i \leq W \right\}.$$

b) Modificación en la definición de la función objetivo:

$$\arg \max_{(x_1, x_2, \ldots, x_N) \in X} \sum_{1 \leq i \leq N} \boxed{x_i w_i}.$$

c) Modificación en la definición del conjunto de soluciones factibles:

$$X = \left\{ (x_1, x_2, \ldots, x_N) \in \{0,1\}^N \middle| \sum_{1 \leq i \leq N} x_i w_i \leq W; \boxed{N/5 \leq \sum_{1 \leq i \leq N} x_i \leq N/2} \right\}.$$

Programación dinámica
13 de mayo de 2008 (Mañana)

Prueba individual

Nombre: **Grupo:**

Problema **50 puntos**

Una importante empresa de inversiones ha diseñado una prueba para los aspirantes a trabajar en ella: les asigna una cantidad de dinero y, durante T meses, deben hacer un total de M inversiones de dicha cantidad con el objetivo de obtener el máximo beneficio. Para ello, pueden invertir el dinero a plazo fijo en N diferentes productos financieros. Cada producto i ($1 \leq i \leq N$) tiene asociado el plazo de recuperación $d(i)$ y el beneficio $b(i)$. El procedimiento de inversión consiste en seleccionar un producto i, invertir el capital completo en él y, transcurrido el tiempo $d(i)$, recoger el beneficio y repetir el proceso. No hay inconveniente en invertir en un mismo producto más de una vez. Durante todo el periodo T, el capital debe estar invertido en algún producto y, una vez finalizado el periodo, hay que haber recuperado el capital inicial.

Deseas saber en qué productos realizarás las inversiones para obtener el máximo beneficio posible.

La siguiente ecuación recursiva te devuelve el valor del máximo beneficio que puedes obtener al hacer m inversiones durante t meses:

$$B(t,m) = \begin{cases} 0, & \text{si } t = 0 \text{ y } m = 0; \\ -\infty, & \text{si } (t > 0 \text{ y } m = 0) \text{ o } (t = 0 \text{ y } m > 0); \\ \max\limits_{\substack{1 \leq i \leq N: \\ d(i) \leq t}} B(t - d(i), m - 1) + b(i), & \text{si } t > 0 \text{ y } m > 0. \end{cases}$$

donde la llamada $B(T, M)$ proporciona el máximo beneficio para el tiempo total con todas las inversiones que hay que realizar. Se pide:

a) Escribe el algoritmo recursivo con memorización asociado a la anterior ecuación.

b) Representa el grafo de dependencias asociado a la ecuación recursiva para $T = 6$, $M = 3$ y $N = 4$ productos financieros de duraciones $d = [1, 3, 5, 2]$ y beneficios $b = [10, 33, 57, 19]$.

c) Indica cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de la ecuación presentada. Justifica brevemente tu respuesta.

Programación dinámica
13 de mayo de 2008 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

Sea el anterior problema:

****

Una importante empresa de inversiones ha diseñado una prueba para los aspirantes a trabajar en ella: les asigna una cantidad de dinero y, durante T meses, deben hacer un total de M inversiones de dicha cantidad con el objetivo de obtener el máximo beneficio. Para ello, pueden invertir el dinero a plazo fijo en N diferentes productos financieros. Cada producto i ($1 \leq i \leq N$) tiene asociado el plazo de recuperación $d(i)$ y el beneficio $b(i)$. El procedimiento de inversión consiste en seleccionar un producto i, invertir el capital completo en él y, transcurrido el tiempo $d(i)$, recoger el beneficio y repetir el proceso. No hay inconveniente en invertir en un mismo producto más de una vez. Durante todo el periodo T, el capital debe estar invertido en algún producto y, una vez finalizado el periodo, hay que haber recuperado el capital inicial.

Deseas saber en qué productos realizarás las inversiones para obtener el máximo beneficio posible.

La siguiente ecuación recursiva te devuelve el valor del máximo beneficio que puedes obtener al hacer m inversiones durante t meses:

$$B(t,m) = \begin{cases} 0, & \text{si } t = 0 \text{ y } m = 0; \\ -\infty, & \text{si } (t > 0 \text{ y } m = 0) \text{ o } (t = 0 \text{ y } m > 0); \\ \max\limits_{\substack{1 \leq i \leq N: \\ d(i) \leq t}} B(t - d(i), m - 1) + b(i), & \text{si } t > 0 \text{ y } m > 0. \end{cases}$$

donde la llamada $B(T, M)$ proporciona el máximo beneficio para el tiempo total con todas las inversiones que hay que realizar.

****

Sobre el problema original se pide:

a) Escribid el algoritmo iterativo *con reducción de la complejidad espacial* que permite obtener el valor del máximo beneficio de la inversiones.

A continuación, se proponen las siguientes modificaciones en el enunciado del problema de las inversiones:

1. No es obligatorio hacer exactamente M inversiones.
2. No es obligatorio mantener el dinero invertido durante todos los meses, si conviene puede estar algún mes disponible.
3. El beneficio que se obtiene con un producto depende del mes en que se haya invertido, es decir, ahora cada producto i tendrá asociados T valores, $b(i,t)$, $(1 \leq t < T)$.

El resto de condiciones se mantienen.
Para este *nuevo problema* se pide:

a) Plantead la ecuación recursiva de programación dinámica que calcule el máximo beneficio que se puede obtener con cualquier combinación válida de inversiones, indicando cuál debe ser la llamada inicial que se efectúe a dicha función para obtener el valor óptimo.

b) Representad el grafo de dependencias entre las llamadas de la ecuación recursiva para $T = 6$ meses y $N = 4$ productos financieros de duraciones $d = [1, 3, 5, 2]$.

c) Indicad cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de vuestra ecuación recursiva que devolviera la *secuencia de decisiones óptima*. Justificad brevemente la respuesta.

d) ¿Sería posible reducir el coste espacial indicado en el apartado anterior si hubiera que diseñar un algoritmo iterativo que sólo encontrara *el valor* del máximo beneficio? Si es así, indicad cuál sería dicho coste. Justificad vuestra respuesta.

Puntuación extra: Formalizad el nuevo problema en términos de optimización.

Programación dinámica
13 de mayo de 2008 (Tarde)

Prueba individual

Nombre: **Grupo:**

Problema **50 puntos**

Se dispone de una cierta cantidad de dinero que se quiere invertir en un periodo de T meses. Al comienzo de cada mes t ($0 \leq t < T$) se puede elegir entre invertir todo el dinero en bonos, en cuyo caso el dinero invertido no se recupera hasta d_b meses después y produce un beneficio $b_b(t)$ que depende del mes de la inversión, o almacenar el dinero en la hucha, en cuyo caso estará disponible el mes siguiente pero sin producir beneficio alguno. Los beneficios que se van obteniendo no se vuelven a invertir, sino que se destinan directamente a una ONG, por lo que la cantidad de dinero disponible para cada inversión siempre es la misma. Una vez finalizado el periodo, el dinero tiene que estar nuevamente disponible.

Deseas saber cómo tienes que hacer las inversiones para obtener el máximo beneficio posible.

La siguiente ecuación recursiva te devuelve el valor del máximo beneficio que puedes obtener de tus inversiones durante t meses:

$$B(t) = \begin{cases} 0, & \text{si } t = 0; \\ B(t-1), & \text{si } 0 < t < d_b; \\ \text{máx}(B(t-1), B(t-d_b) + b_b(t-d_b)), & \text{si } t \geq d_b. \end{cases}$$

donde la llamada $B(T)$ proporciona el máximo beneficio para el tiempo total disponible. Se pide:

a) Escribe el algoritmo recursivo con memorización asociado a la anterior ecuación.

b) Representa el grafo de dependencias asociado a la ecuación recursiva para $T = 8$, $d_b = 3$ y beneficios $b_b = [10, 20, 30, 25, 40, 10, 13, 35]$.

c) Indica cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de la ecuación presentada. Justifica brevemente tu respuesta.

Programación dinámica
13 de mayo de 2008 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

Sea el anterior problema:

****

Se dispone de una cierta cantidad de dinero que se quiere invertir en un periodo de T meses. Al comienzo de cada mes t ($0 \leq t < T$) se puede elegir entre invertir todo el dinero en bonos, en cuyo caso el dinero invertido no se recupera hasta d_b meses después y produce un beneficio $b_b(t)$ que depende del mes de la inversión, o almacenar el dinero en la hucha, en cuyo caso estará disponible el mes siguiente pero sin producir beneficio alguno. Los beneficios que se van obteniendo no se vuelven a invertir, sino que se destinan directamente a una ONG, por lo que la cantidad de dinero disponible para cada inversión siempre es la misma. Una vez finalizado el periodo, el dinero tiene que estar nuevamente disponible.

Deseas saber cómo tienes que hacer las inversiones para obtener el máximo beneficio posible.

La siguiente ecuación recursiva te devuelve el valor del máximo beneficio que puedes obtener de tus inversiones durante t meses:

$$B(t) = \begin{cases} 0, & \text{si } t = 0; \\ B(t-1), & \text{si } 0 < t < d_b; \\ \max(B(t-1), B(t-d_b) + b_b(t-d_b)), & \text{si } t \geq d_b. \end{cases}$$

donde la llamada $B(T)$ proporciona el máximo beneficio para el tiempo total disponible.

****

Sobre el problema original se pide:

a) Escribid el algoritmo iterativo que proporciona *la secuencia de decisiones óptima, concretamente, debe proporcionar una lista de los meses en que se debe invertir en bonos* para obtener el máximo beneficio.

A continuación, se propone la siguiente modificación en el enunciado del problema:

1. Adicionalmente a invertir en bonos o dejarlo en la hucha, los meses en que tengamos el capital disponible también podemos invertirlo en *comprar acciones*. En el caso de las acciones, no tienen una fecha de recuperación fija, sino que pueden venderse en cualquier mes posterior al de compra. Asimismo, disponemos de una estimación del beneficio $b_a(t', t)$ que obtendremos si vendemos en el mes t unas acciones compradas en el mes t'.

El resto de condiciones se mantienen.
Para este *nuevo problema* se pide:

a) Plantead la ecuación recursiva de programación dinámica que calcule el máximo beneficio que se puede obtener con cualquier combinación válida de inversiones, indicando cuál debe ser la llamada inicial que se efectúe a dicha función para obtener el valor óptimo.

b) Representa el grafo de dependencias asociado a la ecuación recursiva para $T = 8$ y $d_b = 3$.

c) Indicad cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de vuestra ecuación recursiva que devolviera la *secuencia de decisiones óptima*. Justificad brevemente la respuesta.

d) ¿Sería posible reducir el coste espacial indicado en el apartado anterior si hubiera que diseñar un algoritmo iterativo que sólo encontrara *el valor* del máximo beneficio? Si es así, indicad cuál sería dicho coste. Justificad vuestra respuesta.

Puntuación extra: Formalizad el nuevo problema en términos de optimización.

Primer control parcial de Esquemas Algorítmicos (IG24)

5 de abril de 2008

1. Divide y vencerás — 5 puntos

El problema de la pirámide maya consiste en determinar la altura de una pirámide a partir del pérfil de un fragmento de la misma que incluye su cima. Para ello, en un vector a disponemos de los datos de n mediciones con la altura de los distintos escalones, de izquierda a derecha, incluidos en el fragmento. Los valores de altura están ordenados de menor a mayor hasta la cima y de mayor a menor a partir de esta. Como la anchura de los escalones es variable, y las mediciones se han hecho tomando un ancho fijo que garantice incluir hasta el escalón más estrecho, el vector podría contener en varias posiciones seguidas un mismo valor, que se correspondería con el de un mismo escalón. Veamos un ejemplo de vector de alturas: a = [3, 3, 5, 5, 5, 5, 8, 8, 12, 12, 12, 9, 7, 7].

Disponemos asimismo de un vector auxiliar i_f que contiene, para cada posición/medición del vector a, las posiciones en que comienza y acaba el escalón correspondiente. El contenido del vector auxiliar asociado al anterior ejemplo sería i_f = [(0,1), (0,1), (2,5), (2,5), (2,5), (2,5), (6,7), (6,7), (8,10), (8,10), (8,10), (11,11), (12,13), (12,13)], donde, por ejemplo, la posición 8 (y la 9 y la 10) contiene el valor $i_f[8] = (8, 10)$, que indica que el escalón comienza en la posición 8 y acaba en la 10. Se pide:

a) Diseña un algoritmo recursivo mediante la técnica de divide y vencerás que devuelva *la altura de la pirámide* representada en a. El algoritmo debe ser lo más eficiente posible.

b) Indica si el algoritmo presenta un comportamiento uniforme o existe un mejor y un peor caso. Si el comportamiento es uniforme, trata de modificarlo para que ante determinadas instancias pueda finalizar antes su ejecución. Señala en qué condiciones (ante que tipos de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor.

c) Realiza un análisis de la complejidad temporal y espacial del algoritmo presentado, justificando los costes en el mejor y en el peor de los casos.

d) Si el algoritmo propuesto presenta recursividad por cola, elimínala e indica si alguno de los costes calculados en el apartado anterior varía.

e) En caso de no disponer del vector auxiliar i_f, ¿seguiría siendo válido el algoritmo que has desarrollado? Si no es así, ¿podrías describir que modificaciones introducirías en tu algoritmo para seguir resolviendo el problema de la forma más eficiente posible y cuál sería el coste temporal que tendría asociado?

2. Búsqueda con retroceso — 5 puntos

Tenemos una lista compuesta por n números enteros positivos, la mitad de los cuales son pares y la otra mitad impares (si n es impar, habrá un elemento más en uno de los dos tipos), dispuestos en cualquier orden.

Dados un valor de umbral mínimo d y un valor de umbral máximo D, queremos encontrar una ordenación de los números de la lista que cumpla que los pares y los impares se alternen y que la diferencia (en valor absoluto) entre cada dos números consecutivos de la ordenación sea mayor que d y menor que D.

Por ejemplo, si la lista fuese L = [2, 5, 19, 29, 14, 10, 25, 22, 13, 31, 26, 30, 36] y los umbrales $d = 4$ y $D = 12$, una posible solución podría ser $[2, 13, 22, 29, 36, 31, 26, 19, 30, 25, 14, 5, 10]$. Queremos encontrar un algoritmo de *búsqueda con retroceso* que resuelva el problema de encontrar una ordenación que cumpla los requisitos indicados.

Se pide que:

a) Describas cómo vas a representar los estados s en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolver el problema.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

e) Realiza una traza de ejecución de tu algoritmo (de la versión que prefieras, indica cuál) para el ejemplo $d = 3$, $D = 8$ y L = [10, 12, 5, 15]. Debes representar el árbol de llamadas efectuadas, incluyendo solamente aquellos estados que se generan durante el proceso de búsqueda. Marca de forma especial los estados que se estudian pero se descartan por no superar la condición de ser prometedores. Indica cuál es el resultado devuelto por el algoritmo.

Segundo control parcial de Esquemas Algorítmicos (IG24)

17 de mayo de 2008

1. Algoritmos voraces — 3.25 puntos

Alquilamos un coche para hacer un viaje por Europa por una ruta previamente fijada de K kilómetros. En nuestro coche caben m litros de gasolina y tiene una autonomia (distancia que puede recorrer con el depósito lleno de gasolina) de n kilómetros (con $m'(< m)$ litros el coche puede recorrer una distancia proporcionalmente inferior: $\frac{m'}{m}n$ kilómetros).

A lo largo de la ruta hay G gasolineras, númeradas de 1 a G según su orden de aparición en la ruta, y para cada una de ellas conocemos tanto el punto kilométrico de nuestra ruta en el que se encuentran, g_i, como el precio al que tienen el litro de la gasolina que emplea nuestro vehículo, p_i, siendo i la gasolinera i-ésima de la ruta ($1 \leq i \leq G$). La distancia entre dos gasolineras consecutivas siempre será menor o igual que la autonomía del coche n.

Quisieramos poder calcular antes de salir de viaje, *mediante un algoritmo voraz*, en qué gasolineras debemos parar a repostar y cuanta cantidad de combustible pondremos en el depósito en cada una de las paradas para gastarnos el menor dinero posible en gasolina en el viaje.

Inicialmente tenemos el deposito vacio, pero no importa, ya que la primera gasolinera está justo a la salida de la agencia de alquiler ($g(1) = 0$). Al llegar al destino (K) deberemos dejar el coche en una sucursal de la agencia (volvemos en avión), por lo que nos interesa que el depósito esté lo más vacio posible.

Ejemplo simplificado: Ruta de $K = 2250$ kilómetros. Depósito de $m = 50$ litros, con el que se pueden recorrer hasta $n = 750$ kilómetros (15 kilómetros por litro). Hay $G = 8$ gasolineras en la ruta, en las posiciones kilométricas $g = [0, 150, 750, 1050, 1350, 1650, 1950, 2100]$ y con precios por litro $p = [1.25, 1.1, 1.12, 1.05, 1.3, 1.5, 1.6, 1.15]$. Una solución para nuestro problema sería [(1,50), (3,50), (5,40), (8,10)], con un coste de $50*1.25+50*1.12+40*1.3+10*1.15 = 182$ euros. (La solución óptima sería [(1,10), (2,50), (3,10), (4,50), (5,20), (8,10)], con un coste de $10*1.25+50*1.1+10*1.12+50*1.05+20*1.3+10*1.15 = 168.7$ euros.)

Se pide:

a) Describe con la mayor precisión (puedes ayudarte de diagramas si lo consideras conveniente) una estrategia voraz que proporcione la lista de gasolineras y litros que tenemos que repostar en cada una de ellas para gastarnos la menor cantidad de dinero posible en gasolina.

b) Indica cuál es el coste temporal de la estrategia que has diseñado, justificándolo adecuadamente.

c) ¿Es la estrategia que has diseñado la más eficiente posible? Si crees que puedes mejorar sus costes (por ejemplo, haciendo uso de determinadas estructuras de datos), indica cómo y señala cuál sería el nuevo coste temporal de la estrategia.

2. Programación dinámica — 4 puntos

Disponemos de una cisterna con una cantidad ilimitada de agua y deseamos llenar una cuba con exactamente M litros. Tenemos un juego de J jarras, y conocemos la capacidad en litros $l(j)$ de cada jarra j ($1 \leq j \leq J$) (todas las capacidades son cantidades enteras). Nuestro objetivo es llenar completamente la cuba de agua usando las jarras para llevar el agua desde la cisterna.

Concretamente, queremos realizar exactamente K viajes para llenar completamente la cuba, teniendo en cuenta que en un viaje sólo podemos trasladar una jarra de agua, que la jarra debe estar completamente llena y que toda el agua que se lleva se debe depositar en la cuba (no se puede desechar agua). Es posible utilizar una misma jarra en más de un viaje.

Cada viaje supone un esfuerzo que podemos cuantificar en función de la cantidad de agua que trasladamos como el cuadradado del peso de la cantidad de agua trasladada (es decir, si usamos la jarra j, el esfuerzo asociado sería $l(j)^2$). Si el esfuerzo total realizado es la suma de los esfuerzos de cada viaje, ¿cuál es la combinación de jarras que debemos utilizar en K viajes para llenar los M litros de la cuba de forma que hagamos el *mínimo* esfuerzo total?

1. Con las condiciones anteriormente expuestas:

   a) Formaliza el problema en términos de optimización.

   b) Plantea la ecuación recursiva asociada al hecho de llenar la cuba con agua haciendo K viajes de forma que estos supongan el menor esfuerzo posible. Indica cuál debe ser la llamada que se efectue a dicha función para obtener el valor óptimo.

   c) Representa el grafo de dependencias (sin extender) entre llamadas de la ecuación recursiva para el ejemplo $M = 10$, $K = 3$, $J = 4$ y $l = [1, 2, 4, 5]$, razonando los costes espacial y temporal con los que un algoritmo iterativo podría efectuar los cálculos para resolver el problema (indica explícitamente, justificando tu respuesta, si es posible efectuar una reducción de la complejidad espacial si sólo nos interesara obtener el valor del esfuerzo mínimo).

   d) ¿Podría haber instancias del problema para las que no existiera una solución factible? Razona la respuesta y, si es así, indica las condiciones que se deberían cumplir para que pueda haber una solución factible.

2. Una variante del problema sería la siguiente: no restringimos los viajes a un número determinado K, nos interesa simplemente la combinación de viajes que presenta el menor esfuerzo; ahora bien, hemos encontrado una forma más ajustada de medir el esfuerzo, y es que este depende, no sólo de los litros que transportemos en un viaje, sino del número de viajes que previamente hayamos realizado, con la siguiente formula: $l(j)^2 + k^2$, que representa el esfuerzo de transportar la jarra j habiendo realizado hasta el momento un total de k viajes. Resumiendo, el problema pasa a

ser cómo llenar la cuba haciendo cualquier número de viajes con jarras de manera que el esfuerzo total sea mínimo (aplicando la nueva fórmula de esfuerzo de un viaje que tiene en cuenta el número de viajes previos realizados).

¿Cómo modificarías la formulación del problema para que recogiera este nuevo planteamiento? ¿Cuál sería ahora la ecuación recursiva que resuelve el problema y la llamada que harías a la misma? ¿Con qué coste temporal y espacial se podría resolver?

3. Programación dinámica — 2.75 puntos

Sean dos sucursales de una casa de suministro de tornillos que sirven a N tiendas, siendo $d(i)$ la demanda de tornillos de la tienda i ($1 \leq i \leq N$). El número de tornillos que tiene disponible cada sucursal es R_j ($j \in \{1, 2\}$). Se cumple que la demanda total de tornillos, $\sum_{1 \leq i \leq N} d(i)$, coincide con los tornillos totales disponibles, $R_1 + R_2$. Enviar los $d(i)$ tornillos que solicita la tienda i desde la sucursal j tiene un coste de $c(j, i)$.

Debes averiguar, empleando la técnica de programación dinámica, qué sucursal deber realizar cada envío de forma que se cubra toda la demanda con el menor coste posible para la casa de suministros. Como ayuda, la siguiente ecuación recursiva de programación dinámica permite resolver el problema de obtener el valor del suministro más barato:

$$C(i, r_1, r_2) = \begin{cases} 0, & \text{si } i = 0; \\ +\infty, & \text{si } i > 0 \text{ y } r_1 < d(i) \text{ y } r_2 < d(i); \\ C(i-1, r_1, r_2 - d(i)) + c(2, i), & \text{si } i > 0 \text{ y } r_1 < d(i) \text{ y } r_2 \geq d(i); \\ C(i-1, r_1 - d(i), r_2) + c(1, i), & \text{si } i > 0 \text{ y } r_1 \geq d(i) \text{ y } r_2 < d(i); \\ \text{mín}(C(i-1, r_1 - d(i), r_2) + c(1, i), C(i-1, r_1, r_2 - d(i)) + c(2, i)), & \text{en otro caso.} \end{cases}$$

La llamada $C(N, R_1, R_2)$ proporciona dicho valor.
Se pide:

a) Representa el grafo de dependencias (sin extender) para el problema en el que $N = 5$, $d = [3, 4, 2, 1, 6]$, $R_1 = 7$, $R_2 = 9$, $c[1] = [2, 3, 4, 3, 5]$ y $c[2] = [3, 4, 1, 2, 6]$ (no te agobies si no te sale muy ordenado). Indica qué representa el estado $(3, 6, 3)$ y cómo interpretarías el significado de un arco entre los estados $(2, 6, 1)$ y $(3, 6, 3)$ dentro del grafo.

b) Diseña un algoritmo iterativo que devuelva, para cada tienda, *cuál es la sucursal que se encarga de su envío* para garantizar que se realiza el menor gasto.

c) Indica cuál es el coste temporal y espacial del algoritmo diseñado justificando brevemente tu respuesta.

d) ¿Crees que sería posible reducir el coste espacial si únicamente nos interesará obtener el valor del mínimo gasto? Si es así, índica cuál sería dicho coste y justifícalo.

Examen convocatoria ordinaria de Esquemas Algorítmicos (IG24)

13 de junio de 2008

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo):
Parte 1 (Preg. 1 y 2) ☐ **Parte 2 (Preg. 3 y 4)** ☐ **Partes 1 y 2** ☐ **Renuncio a la evaluación alternativa** ☐
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Divide y vencerás — 2,5 puntos

Tenemos una foto con un punto de luz y la iluminación que produce. Dada una matriz de $n \times m$ que representa la digitalización de los distintos tonos de gris de la foto, debes localizar en qué posición de la matriz se encuentra el punto de luz. Los valores almacenados en la matriz pueden variar entre el 0 (blanco) y el 65535 (negro). Debes tener en cuenta que la matriz contiene una gradación de forma concéntrica de tonos (valores enteros crecientes) desde el punto central y en círculo hasta los puntos más alejados. Veamos un ejemplo de matriz de 10×15:

```
luz =
[[ 25, 25, 25, 25, 25, 25, 25, 30, 35, 40, 45, 50, 55, 60,  65],
 [ 20, 20, 20, 20, 20, 20, 25, 30, 35, 40, 45, 50, 55, 60,  65],
 [ 15, 15, 15, 15, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60,  65],
 [ 15, 10, 10, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60,  65],
 [ 15, 10,  5, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60,  65],
 [ 15, 10, 10, 10, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60,  65],
 [ 15, 15, 15, 15, 15, 20, 25, 30, 35, 40, 45, 50, 55, 60,  65],
 [ 20, 20, 20, 20, 20, 20, 25, 30, 35, 40, 45, 50, 55, 60,  65],
 [ 25, 25, 25, 25, 25, 25, 25, 30, 35, 40, 45, 50, 55, 60,  65],
 [ 30, 30, 30, 30, 30, 30, 30, 30, 35, 40, 45, 50, 55, 60,  65]]
```

Como puedes observar, el punto de luz está en la posición (4,2), ya que es la posición de la matriz que presenta un menor valor. También puedes ver la organización del resto de la matriz alrededor de dicho punto, con valores crecientes en cuadrados concéntricos.

Todas las posibles matrices tendrán la estructura de la del ejemplo aunque, evidentemente, en otros casos el punto de luz podría estar situado en cualquier posición de la matriz y los valores contenidos en la matriz podrán variar en función de la luminosidad original.

Has de diseñar un algoritmo de *divide y vencerás* que, dada una matriz como la descrita, averigüe de la manera más eficiente posible la posición del punto de luz. Se pìde:

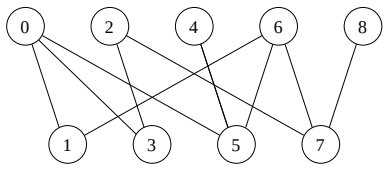
a) Diseña un algoritmo recursivo mediante la técnica de divide y vencerás que permita resolver este problema.

b) Analiza y justifica la complejidad temporal y espacial del algoritmo desarrollado en el apartado anterior, indicando si presenta un comportamiento uniforme o existe un mejor y un peor caso.

2. Búsqueda con retroceso — 2,5 puntos

Un grafo no dirigido $G = (V, E)$ es bipartito si su conjunto de vértices V puede partirse en dos subconjuntos disjuntos V_1 y V_2 de modo que sólo haya aristas de la forma (u, v) con $u \in V_1$ y $v \in V_2$ (es decir, no puede haber ni una sóla arista entre vértices de un mismo subconjunto).

Por ejemplo, el siguiente grafo es bipartito:



ya que si dividimos V en los dos subconjuntos $V_1 = \{0, 2, 4, 6, 8\}$ y $V_2 = \{1, 3, 5, 7\}$, todas las aristas de E unen vértices de V_1 con vértices de V_2.

Debes desarrollar un algoritmo de *búsqueda con retroceso* que, a partir de $G = (V, E)$, devuelva, si existe, la bipartición de V en dos conjuntos. Se pide:

a) Describe cómo vas a representar los estados en la resolución del problema planteado.

b) Escribe un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que realizarías.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Crees que es posible mejorar la eficiencia del algoritmo que has desarrollado introduciendo alguna variable o estructura de datos adicional o haciendo algún tipo de preproceso? Si es así, describe con claridad la mejora o mejoras que introducirías en el algoritmo e indica si ello afectaría a los costes previamente calculados.

3. Voraces — 1,5 puntos

En un concurso de programación hay n participantes que debemos organizar en equipos de 3, 2 o 1 persona. De cada persona conocemos su grado de afinidad con los demás participantes, recogida en una matriz A, de forma que $A[i, j]$ nos indica la afinidad entre la persona i y la j (no necesariamente simétrica y que puede ser negativa si se llevan mal).

Queremos desarrollar un *algoritmo voraz* que organice los equipos de manera que se máximice la suma de afinidades entre los miembros de los equipos formados.

Un ejemplo para $n = 10$ participantes podría venirnos dado a partir de la siguiente matriz de afinidades:

A	1	2	3	4	5	6	7	8	9	10
1	-	-3	4	11	6	3	5	7	-8	-5
2	4	-	0	3	2	4	12	0	-1	5
3	-1	0	-	6	9	8	7	-10	-9	3
4	9	3	1	-	-2	-5	6	8	3	0
5	0	-2	15	-1	-	3	-7	-5	0	1
6	-2	3	4	1	6	-	-5	-3	0	-7
7	0	11	3	-1	2	-3	-	7	6	9
8	5	3	4	9	-1	-3	-8	-	4	3
9	3	5	0	-2	1	7	0	2	-	-3
10	-3	8	5	-4	-7	5	7	5	-9	-

Una posible solución para esta matriz sería hacer los grupos ((1,3,7), (9,6), (2,4), (8,5), (10)), cuya suma de afinidades sería: $(4 + 5 - 1 + 7 + 0 + 3) + (7 + 0) + (3 + 3) + (-1 - 5) + (0) = 18 + 7 + 6 - 6 + 0 = 25$. Contesta a las siguientes cuestiones:

a) Ya que se trata de un problema de optimización, debes indicar las características que posee una *solución factible*, cuál es la función bjetivo que se pretende optimizar y qué propiedad debe cumplir una solución factible para ser considerada óptima.

b) Para resolver este problema, se plantea una estrategia voraz basada en ir construyendo equipos de manera incremental (distinguiendo en el proceso entre equipos definitivos y equipos candidatos) y la idea es ir modificando los equipos candidatos para, en un momento dado, pasarlos a definitivos. La estrategia es la siguiente: "Inicialmente cada persona es un equipo candidato (de un sólo miembro). Mientras queden al menos 3 participantes pendientes de clasificación definitiva, buscamos en A el valor máximo de $A[i, j]$ entre los participantes i y j que no estén en equipos definitivos y que no pertenezcan al mismo equipo candidato, y fusionamos los equipos candidatos a los que pertenecen i y j en un nuevo equipo candidato, teniendo en cuenta que si formamos un equipo de 3 personas, el equipo pasa de candidato a definitivo. En el momento en que queden menos de 3 participantes por considerar, se fusionan como un equipo definitivo y acabamos."

   Con respecto a esta estrategia:

   1. ¿Garantiza encontrar una solución factible? Justifica tu respuesta (si la respuesta es no, debes incluir un contraejemplo).
   2. ¿Encuentra siempre una solución óptima? Justifica la respuesta (si la respuesta es no, debes incluir un contraejemplo).
   3. ¿Cuál sería la complejidad temporal de la estrategia para implementarla de la manera más eficiente posible (indica para ello las estructuras de datos concretas que utilizarías, describiendo los detalles que justifiquen el coste señalado)?

c) Planteamos ahora esta otra estrategia voraz basada en construir los equipos directamente a partir de los participantes más afines aún no asignados a ningún equipo, de manera que en el momento en que se construye un equipo este es definitivo. La estrategia es la siguiente: "Inicialmente ningún participante forma parte de ningún equipo. Mientras queden al menos 3 participantes pendientes de asignar, buscamos el participante i cuyas suma de valores $A[i, j] + A[i, k]$ sea máxima, donde i, j y k deben ser participantes aún no asignados, y creamos un equipo con esos tres participantes. En el momento en que queden menos de 3 participantes por considerar, se crea un equipo con ellos y acabamos." Responde a las mismas preguntas (1, 2 y 3) del anterior apartado para esta nueva estrategia.

4. Programación dinámica — 3,5 puntos

Se desea hacer una copia de seguridad de N ficheros en un DVD con capacidad (entera) de M megas de manera que podamos llenarlo al máximo, pero siempre y cuando no metamos más de F ficheros. Cada fichero tiene un tamaño (entero) de $t(i)$ megas ($1 \leq i \leq N$).

Deseamos diseñar un algoritmo mediante la técnica de *programación dinámica* que resuelva el problema de obtener el valor de la máxima ocupación en el DVD que podamos conseguir con las condiciones fijadas. Se pide:

a) Formaliza el problema en términos de optimización.

b) Diseña una ecuación recursiva que calcule el valor de la máxima ocupación posible del DVD con un máximo de F ficheros, indicando cuál debe ser la llamada que se haga a dicha ecuación para resolver el problema.

c) Diseña un algoritmo iterativo que recorra el anterior grafo y devuelva el valor solicitado empleando la menor ocupación espacial (en términos asintóticos) posible. Indica y justifica sus costes espacial y temporal.

Examen segunda convocatoria de Esquemas Algorítmicos (IG24)

2 de septiembre de 2008

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo):
Parte 1 (Preg. 1 y 2) ☐ **Parte 2 (Preg. 3 y 4)** ☐ **Partes 1 y 2** ☐ **Renuncio a la evaluación alternativa** ☐
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Divide y vencerás — 2,5 puntos

Dado un vector *desordenado* de n enteros, diseña un algoritmo mediante la técnica de *divide y vencerás* que devuelva `true` si la suma de los k menores elementos del vector es mayor que un valor dado V y `false` en caso contrario.

Se pide:

a) Diseña un algoritmo recursivo *mediante la técnica de divide y vencerás* que permita resolver este problema con un coste temporal *esperado* de $O(n)$.

Pistas: para facilitar la elaboración del algoritmo, puedes asumir que dispones de la función `partition` empleada en el algoritmo `quicksort`. La función de *divide y vencerás* que desarrolles no tiene porque calcular directamente el resultado pedido, sino que puede devolver algún resultado intermedio que permita, una vez acabado su cálculo, y mediante otra función, obtener la solución. En cualquier caso, la estrategia principal de resolución deberá ser la de *divide y vencerás* y todas las funciones necesarias para el cálculo solicitado deberás incluirlas (la función `partition` no es necesario que la incluyas).

b) Analiza y justifica la complejidad temporal y espacial del algoritmo desarrollado en el apartado anterior en el mejor y el peor caso. Señala ante qué tipo de entradas puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor.

c) Si el algoritmo propuesto presenta recursividad por cola, elimínala e indica si alguno de los costes calculados en el apartado anterior varía.

2. Búsqueda con retroceso — 2,5 puntos

Dado un grafo $G = (V, E)$, diseña un algoritmo de *búsqueda con retroceso* que encuentre un camino que pase por todas las aristas del grafo *una sóla vez*. Para ello se pide que:

a) Describas cómo vas a representar los estados s en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolverlo.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Crees que es posible mejorar la eficiencia del algoritmo que has desarrollado introduciendo alguna variable o estructura de datos adicional o haciendo algún tipo de preproceso? Si es así, describe con claridad la mejora o mejoras que introducirías en el algoritmo e indica si ello afectaría a los costes previamente calculados.

3. Voraces — 1,5 puntos

El paintball es un juego de guerra entre dos equipos armados con balas de pintura. Participamos en una competición en la que el equipo enemigo ha tomado n objetivos y queremos reconquistarlos. En cada uno de los i objetivos ($1 \leq i \leq n$) el enemigo ha dejado a x_i personas defendiéndolo. Nuestro equipo está formado por n comandos de intervención, cada uno de los cuales tiene y_i personas ($1 \leq i \leq n$). Nos interesa reconquistar la mayor cantidad posible de objetivos. Para *garantizar* el éxito de la intervención en un objetivo, es necesario que el comando de ataque tenga el menos tantos efectivos como el número de defensores.

Debes diseñar una *estrategia voraz* lo más eficiente posible que nos permita conquistar el mayor número de objetivos, describiéndola con claridad e indicando y justificando su coste temporal.

4. Programación dinámica — 3,5 puntos

Se necesita llenar la bodega de un barco con B millones de litros de agua procedentes de una planta desalinizadora. La capacidad de producción de agua de la planta supera esa cantidad. Para cargar la bodega del barco, la planta dispone de un juego de C cubas (numeradas de 1 a C). La cuba i-ésima tiene una capacidad de l_i millones de litros, siendo dicha cantidad un número entero. Sólo es posible trasvasar agua de la planta a la bodega del barco utilizando las cubas (debe llenarse completamente cada cuba usada y volcarse toda el agua de la cuba en la bodega), pudiendo emplearse más de una vez una misma cuba. Si también conocemos el tiempo t_i que utilizamos en el proceso de llenado, traslado y vaciado de cada cuba i ($1 \leq i \leq C$), ¿cuál es el menor tiempo que podemos emplear para llenar completamente la bodega del barco utilizando las cubas disponibles?

Se pide:

a) Formaliza el problema en términos de optimización,

b) plantea la ecuación recursiva de *programación dinámica* que calcula el menor tiempo con el que llenar la bodega, indicando cuál debe ser la llamada recursiva que se efectue a dicha ecuación para resolver el problema,

c) representa el grafo de dependencias entre las llamadas recursivas asociadas a la anterior ecuación para la instancia $B = 13$, $C = 5$, $l = [3, 2, 4, 6, 1]$ y $t = [5, 6, 7, 10, 1]$, y

d) diseña un algoritmo iterativo de *programación dinámica* lo más eficiente posible que recorra el anterior grafo y devuelva el tiempo mínimo con que puede llenarse la bodega y las cubas empleadas para realizar dicho trasvase, indicando y justificando sus costes espacial y temporal.

Análisis de algoritmos y Estructuras de datos
28 de febrero de 2007 (Mañana)

Prueba individual

Nombre: **Grupo:**

Análisis de algoritmos y estructuras de datos

La siguiente función recibe dos vectores, A y B, que contienen n y m valores respectivamente, y devuelve un tercer vector, C, que contiene todos aquellos elementos que *están en A pero no están en B*:

```
1  def solo_en_A(A, B):
2      C = []
3      for v1 in A:
4          esta = False
5          for v2 in B:
6              if v1 == v2:
7                  esta = True
8                  break
9          if not esta:
10             C.append(v1)
11     return C
```

a) Indica y justifica cuál es el coste temporal y espacial asintótico del algoritmo, distinguiendo entre el mejor y el peor de los casos si los hubiera.

b) ¿Podrías utilizar alguna estructura de datos auxiliar que te permitiera reducir el coste temporal del algoritmo propuesto? Si es así, indica cuál, di como la emplearías y cuál sería el coste temporal resultante.

Prueba colectiva

Grupo:
Asistentes:

Estructuras de datos y costes

Deseamos disponer de una estructura de datos Mm que nos permita almacena valores y realizar, de la forma más eficiente posible, las siguientes operaciones:

a) añadir un valor a Mm,

b) consultar el valor mínimo de Mm,

c) consultar el valor máximo de Mm,

d) eliminar el valor mínimo de Mm,

e) eliminar el valor máximo de Mm,

Debes señalar brevemente (y con claridad) como implementarías esta estructura de datos y como y con que coste temporal se podría ejecutar cada una de las operaciones antes citadas. Señala también el coste espacial asociado a la estructura.

Análisis de algoritmos y Estructuras de datos
28 de febrero de 2007 (Tarde)

Prueba individual

Nombre: **Grupo:**

Análisis de algoritmos y estructuras de datos

La siguiente función recibe un vector de valores (desordenado) A y una variable *margen* y devuelve un vector B *ordenado de mayor a menor* que contiene los valores más grandes de A que cumplen que la diferencia entre dos valores consecutivos es menor o igual que el valor de *margen* ($B[i] - B[i+1] \leq margen, 0 \leq i < len(B) - 1$). Por ejemplo, si $A = [30, 25, 42, 32, 12, 50, 48, 1, 44]$ y $margen = 5$ entonces $B = [50, 48, 44, 42]$:

```
1  def selecciona_mayores_con_margen (A, margen):
2      B = []
3      x = max(A)
4      A.remove(x)
5      B.append(x)
6      while len(A) > 0 and (x-max(A) <= margen):
7          x = max(A)
8          A.remove(x)
9          B.append(x)
10     return B
```

a) Indica y justifica cuál es el coste temporal y espacial asintótico del algoritmo, distinguiendo entre el mejor y el peor de los casos si los hubiera.

b) ¿Podrías utilizar alguna estructura de datos auxiliar que te permitiera reducir el coste temporal del algoritmo propuesto? Si es así, indica cuál, di como la emplearías y cuál sería el coste temporal resultante.

Prueba colectiva

Grupo:
Asistentes:

Estructuras de datos y costes

Deseamos incorporar a la estructura MFset la operación de eliminar un elemento de un conjunto sin modificar los excelentes costes del resto de operaciones que ofrece la estructura. Dicha operación, $S.delete(x)$, debe garantizar que el resto de elementos del conjunto al que pertenece x sigan perteneciendo a él.

Para incorporar la nueva operación es posible que necesites utilizar alguna estructura auxiliar. Dicha estructura no podrá aumentar el coste espacial *asintótico* asociado al MFset. Por simplicidad, asume que los elementos sobre los que se aplique la operación $S.delete(x)$ no podrán ser los representantes de un conjunto.

Indica como modificarías el MFset para poder ofrecer esta nueva operación con el menor coste temporal posible. Detalla como afectaría a las tres operaciones básicas sobre MFsets ($S.new(x)$, $S.find(x)$ y $S.merge(x, y)$), así como el coste temporal de la operación.

Divide y vencerás
8 de marzo de 2007 (Mañana)

Prueba individual

Nombre: **Grupo:**

1. Eliminación de recursión por cola

El siguiente algoritmo recursivo ha sido desarrollado mediante la técnica de divide y vencerás y permite realizar la búsqueda ternaria de un valor v en un vector a ordenado de forma creciente, devolviendo *True* si el elemento está en el vector y *False* en caso contrario:

```
1  def search(a, v):
2    def _search(i, k):
3      if k-i == 1:          return (v == a[i])
4      elif k-i == 2:        return (v == a[i] or v == a[i+1])
5      elif k-i > 2:
6        j1 = i + (k-i)/3
7        j2 = i + 2*(k-i)/3
8        if v == a[j1] or v==a[j2]: return True
9        elif v < a[j1]:            return _search(i, j1)
10       elif v < a[j2]:            return _search(j1+1, j2)
11       else:                      return _search(j2+1, k)
12     return False
13
14   return _search(0, len(a))
```

Se pide que lo conviertas en un algoritmo iterativo mediante la técnica de eliminar la recusión por cola. Analiza la complejidad *espacial* del nuevo algoritmo iterativo y compárala con la del algoritmo recursivo original.

2. Resolución de recurrencias (Teorema maestro y corolarios)

Encuentra (justificándola) una cota asintótica superior para cada una de las siguientes relaciones de recurrencia $T(n)$, que es una constante en $n = 1$, y cuyo término recursivo general (para $n > 1$) es:

a) $T(n) = 2T(n/3) + n^2c_1 + nc_2$.

b) $T(n) = 2T(n/3) + n^{1/2}c_1$.

c) $T(n) = 2T(n/4) + \lg nc_1$.

d) $T(n) = T(n/2) + n^2 \lg nc_1 + n^2c_2$.

(Nota: c_1 y c_2 son constantes positivas arbitrarias.)

Divide y vencerás
8 de marzo de 2007 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema

Dados un vector *ordenado de forma creciente a* de reales positivos (sin elementos repetidos) y un valor real x (que puede estar o no en el vector):

a) Diseñad un algoritmo recursivo *mediante la técnica de divide y vencerás* que devuelva el número de elementos menores o iguales que x que contiene el vector y el número de elementos mayores que x del vector. El algoritmo desarrollado debe ser lo más eficiente posible, tanto desde el punto de vista de la complejidad temporal como de la espacial.

b) El algoritmo que habéis desarrollado ¿tiene mejor y peor caso o se comporta de manera uniforme? Describid con claridad el mejor y el peor caso si los hubiera y realizad los correspondientes análisis de complejidad temporal y espacial justificando los resultados obtenidos.

c) ¿Es vuestro algoritmo más eficiente que una aproximación directa para el mismo problema? Justificadlo.

Divide y vencerás
8 de marzo de 2007 (Tarde)

Prueba individual

Nombre: **Grupo:**

1. Eliminación de recursión por cola

El siguiente algoritmo recursivo ha sido desarrollado mediante la técnica de divide y vencerás y permite calcular, dado un vector a de reales positivos distintos ordenado de forma creciente y un valor $x \in \mathbb{R}$, el número de elementos del vector menores o iguales que x y el número de elementos mayores que x:

```
1  def men_y_may(a,x):
2    def _men_y_may(i,k):
3      if (k-i)==1:
4        if a[i]<=x: return (i+1,len(a)-(i+1))
5        else:       return (i,len(a)-i)
6      else:
7        j=(k+i)/2
8        if a[j-1]<=x<a[j]: return (j,len(a)-j)
9        elif a[j]<=x:      return _men_y_may(j,k)
10       else:              return _men_y_may(i,j)
11
12   if len(a)>0:
13     return _men_y_may(0,len(a))
14   return(0,0)
```

Se pide que lo conviertas en un algoritmo iterativo mediante la técnica de eliminar la recusión por cola. Analiza la complejidad *espacial* del nuevo algoritmo iterativo y compárala con la del algoritmo recursivo original.

2. Resolución de recurrencias (Teorema maestro y corolarios) Modelo 1

Encuentra (justificándola) una cota asintótica superior para cada una de las siguientes relaciones de recurrencia $T(n)$, que es una constante en $n = 1$, y cuyo término recursivo general (para $n > 1$) es:

a) $T(n) = 5T(n/5) + n \lg n c_1 + n^2 c_2$.

b) $T(n) = 3T(n/2) + c_1$.

c) $T(n) = 2T(n/4) + \sqrt{n} c_1$.

d) $T(n) = 4T(n/2) + nc_1 + n \lg n c_2$.

(Nota: c_1 y c_2 son constantes positivas arbitrarias.)

2. Resolución de recurrencias (Teorema maestro y corolarios) Modelo 2

Encuentra (justificándola) una cota asintótica superior para cada una de las siguientes relaciones de recurrencia $T(n)$, que es una constante en $n = 1$, y cuyo término recursivo general (para $n > 1$) es:

a) $T(n) = 4T(n/2) + c_1$.

b) $T(n) = 5T(n/5) + n^2 \lg n c_1 + nc_2$.

c) $T(n) = 4T(n/2) + n \lg n c_1 + n^2 c_2$.

d) $T(n) = 2T(n/4) + \sqrt{n} c_1$.

(Nota: c_1 y c_2 son constantes positivas arbitrarias.)

Divide y vencerás
8 de marzo de 2007 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema

Dado un vector a de enteros positivos (sin ordenar):

a) Diseñad un algoritmo recursivo *mediante la técnica de divide y vencerás* que devuelva la suma de los valores pares y la suma de los valores impares de a. El algoritmo desarrollado debe ser lo más eficiente posible, tanto desde el punto de vista de la complejidad temporal como de la espacial.

b) El algoritmo que habéis desarrollado ¿tiene mejor y peor caso o se comporta de manera uniforme? Describid con claridad el mejor y el peor caso si los hubiera y realizad los correspondientes análisis de complejidad temporal y espacial justificando los resultados obtenidos.

c) ¿Es vuestro algoritmo más eficiente que una aproximación directa para el mismo problema? Justificadlo.

Búsqueda con retroceso
29 de marzo de 2007 (Mañana)

Prueba individual

Nombre: **Grupo:**

Traza problema de las n reinas **50 puntos**

En el problema de las n reinas hay que disponer n reinas en un tablero de $n \times n$ escaques de modo que ninguna amenace a otra. El método `bactracking` de la clase `NQueensSolver1` permite encontrar una solución factible para el problema aplicando la técnica de búsqueda con retroceso:

```
class NQueensSolver1:

    def is_complete(self, s):
        return len(s) == self.n
    def is_promising(self, s):
        return all(s[j]!=s[i] and j-i!=abs(s[j]-s[i]) for i in xrange(len(s)) \
                                                      for j in xrange(i+1, len(s)))

    def backtracking(self, s):
        if self.is_complete(s): return s
        for row in xrange(self.n):
            s' = s + [row]
            if self.is_promising(s'):
                found = self.backtracking(s')
                if found != None: return found
        return None
```

El método también permite encontrar una solución factible que tenga un prefijo determinado.

Realiza una traza de la ejecución del método para la llamada `NQueensSolver1(5).backtracking([1,4])` (es decir, el problema de ubicar 5 reinas sin que se amenacen en un tablero de 5×5 donde las dos primeras tienen que ocupar las posiciones 1 y 4 respectivamente).

Debes dibujar el árbol de estados incluyendo tan sólo aquellos *que se generan* durante el proceso de búsqueda. Incluye también los estados que se estudian pero se descartan por no superar la condición de ser prometedores, y márcalos de forma especial. Indica también claramente cuál es el resultado devuelto por el algoritmo.

Búsqueda con retroceso
29 de marzo de 2007 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

Entre las películas que se exhiben en un multicine de una gran capital se proyectan todos los estrenos de la semana. Los viernes las proyecciones se realizan de manera ininterrumpida desde las 12 del mediodia hasta la madrugada, con varios pases de cada película (en función de su comercialidad).

Un crítico de cine que tiene que ver cada semana todas las películas estrenadas para hacer una crónica en el dominical de su periodico, decide invertir todo el viernes en ver los estrenos (el sábado escribe las críticas).

Debes diseñar un algoritmo de *búsqueda con retroceso* que, a partir de las duraciones (en minutos) y horas de comienzo de cada sesión de todos los estrenos, devuelva como resultado la sesión de cada película a la que debe acudir el crítico para poder verlas todas enteras (o un valor especial si la combinación no es posible).

El siguiente ejemplo muestra unos posibles datos de entrada para un problema con 7 estrenos (númerados entre 1 y 7), en el que las duraciones de las películas se facilitan en una lista y las sesiones de cada película en una lista de listas:

```
duracion=[90, 120, 105, 90, 135, 110, 120]     #Película 1: 90 min; película 2: 120 min ...
pases=[[12:00, 14:00, 16:00, 18:00, 20:00, 22:00, 00:00],               #Película 1
       [12:30, 14:45, 17:00, 19:15, 20:30, 22:45, 01:00],               #Película 2
       [12:00, 14:00, 16:00, 18:00, 20:00, 22:00, 00:00],               #Película 3
       [12:00, 13:00, 14:00, 15:00, 16:00, 17:00, 18:00, 19:00, 20:00], #Película 4
       [16:00, 19:00, 22:30, 01:00],                                    #Película 5
       [12:30, 14:30, 16:30, 18:30, 20:30, 22:30, 00:30],               #Película 6
       [12:30, 14:00, 16:30, 19:00, 21:30, 00:00]]                      #Película 7
```

Una posible solución que podría devolver el algoritmo sería `[2,6,1,8,4,5,3]`, que se correspondería con ver la película 1 a las 14:00, la 2 a las 22:45, la 3 a las 12:00, la 4 a las 19:00, la 5 a la 1:00, la 6 a las 20:30 y la 7 a las 16:30.

Se pide que:

a) Describas cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolverlo (si prefieres, puedes utilizar el esquema, en cuyo caso deberás escribir el código de los métodos `is_complete(s)`, `is_factible(s)`, `branch(s)` e `is_promising(s)` e indicar si efectuas alguna acción adicional).

Nota: Para simplificar la programación, asume que dispones de una función `solapa(s1,d1,s2,d2)` que, dada la hora de comienzo de una película $s1$ y su duración $d1$, y la hora de comienzo de otra película $s2$ y su duración $d2$, devuelve `True` si se solapan las proyecciones y `False` si no es así.

Búsqueda con retroceso
29 de marzo de 2007 (Tarde)

Prueba individual

Nombre: **Grupo:**

Traza problema de la suma del subconjunto **50 puntos**

En el problema de la suma del subconjunto disponemos de N objetos con pesos $w_1, w_2, \ldots, w_N$ y de una mochila con capacidad para soportar una carga W. Deseamos cargar la mochila con una selección arbitraria de objetos cuyo peso sea *exactamente* W. La función `subset_sum` (*la versión refinada del algoritmo*) permite encontrar una solución factible para el problema aplicando la técnica de búsqueda con retroceso:

```
def subset_sum(w, W):
  w = OffsetArray(sorted(w))
  sum_w = OffsetArray([0]* (len(w)+1))
  sum_w[len(w)] = w[len(w)]
  for i in xrange(len(w)-1, 0, -1): sum_w[i] = sum_w[i+1] + w[i]
  x = OffsetArray([0] * len(w))

  def backtracking(i, W):
    if W == 0:
      for j in xrange(i, len(w)+1): x[j] = 0
      return x
    elif i <= len(w):
      for x[i] in 1, 0:
        if W-x[i]*w[i] >= 0 and sum_w[i+1] >= W-x[i]*w[i]:
          found = backtracking(i+1, W-x[i]*w[i])
          if found != None: return found
        else:
          return None
    return None

  return backtracking(1, W)
```

Modelo 1

Realiza una traza de la ejecución del método para una mochila con capacidad para $W = 17$ y $N = 5$ objetos de pesos $w = [8, 1, 9, 2, 7]$.

Debes dibujar el árbol de estados incluyendo tan sólo aquellos *que se generan* durante el proceso de búsqueda y señalando, al lado de cada estado, el valor del peso total asociado al estado que queda por rellenar en la mochila. Incluye también los estados que se estudian pero se descartan por no superar la condición de ser prometedores, y márcalos de forma especial. Indica también claramente cuál es el resultado devuelto por el algoritmo.

Modelo 2

Realiza una traza de la ejecución del método para una mochila con capacidad para $W = 10$ y $N = 5$ objetos de pesos $w = [2, 7, 10, 4, 6]$.

Debes dibujar el árbol de estados incluyendo tan sólo aquellos *que se generan* durante el proceso de búsqueda y señalando, al lado de cada estado, el valor del peso total asociado al estado que queda por rellenar en la mochila. Incluye también los estados que se estudian pero se descartan por no superar la condición de ser prometedores, y márcalos de forma especial. Indica también claramente cuál es el resultado devuelto por el algoritmo.

Búsqueda con retroceso
29 de marzo de 2007 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

En una asignatura de ITIG los alumnos pueden realizar n prácticas no obligatorias a lo largo del curso. La puntuación de cada práctica puede ser: 0 (muy mal), 2.5 (mal), 5 (regular), 7.5 (bien) o 10 (excelente). A partir de las experiencias de otros años, el profesor da una estimación para cada práctica del tiempo (en minutos) que necesita invertir un alumno medio para llegar a cada una de las puntuaciones.

Por ejemplo, si el número de prácticas fuera seis ($n = 6$), los tiempos estimados para cada práctica y puntuación podrían ser:

```
tiempos = [[0, 10, 30, 50, 100],  #Práctica 1: 0 min. para 0 puntos, 10 para 2.5, 30 para 5.
           [0, 20, 40, 50,  60],  #Tiempos puntuaciones práctica 2
           [0, 10, 20, 40,  80],  #Tiempos puntuaciones práctica 3
           [0, 50, 65, 75,  80],  #Tiempos puntuaciones práctica 4
           [0, 15, 25, 40,  60],  #Tiempos puntuaciones práctica 5
           [0, 20, 50, 55,  70]]  #Tiempos puntuaciones práctica 6
```

Como alumno de la asignatura, has visto que no puedes invertir más de T minutos en las prácticas (¡también estás matriculado de la IG24!), pero te gustaría saber si ello te va a permitir alcanzar como mínimo una puntuación P y, si es así, conocer al menos una forma de repartir el tiempo T entre las prácticas para lograrlo. Siguiendo con el ejemplo:

Si $T = 210$ minutos y $P = 35$ puntos, la secuencia de índices $[1, 4, 2, 4, 3, 0]$ se corresponde con la combinación de tiempos $10 + 60 + 20 + 80 + 40 + 0 = 210$, que permite alcanzar la puntuación 2.5 $+10 + 5 + 10+$ 7.5 $+0 = 35$. Sin embargo, con $T = 100$ minutos no es posible sumar $P = 30$ puntos.

Por supuesto, cualquier combinación en la que se alcance *o supere* la puntuación gastando el tiempo disponible *o un tiempo menor* es también válida.

Debes diseñar un algoritmo de *búsqueda con retroceso* que, a partir del número de prácticas, las estimaciones de tiempos para cada puntuación de cada práctica, el tiempo que se quiere invertir y la puntuación que se pretende alcanzar, devuelva una combinación válida (si la hay).

Para ello, se pide que:

a) Describas cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolverlo (si prefieres, puedes utilizar el esquema, en cuyo caso deberás escribir el código de los métodos `is_complete(s)`, `is_factible(s)`, `branch(s)` e `is_promising(s)` e indicar si efectuas alguna acción adicional).

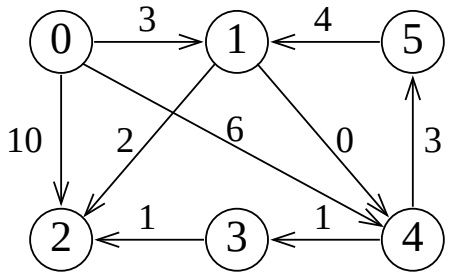
Algoritmos voraces
25 de abril de 2007 (Mañana)

Prueba individual

Nombre: **Grupo:**

Algoritmo de Dijkstra **35 puntos**

Dado el siguiente grafo dirigido y ponderado:



a) Haz una traza, iteración a iteración, de la ejecución de `all_shortest_paths_from_source2` (pág. 7-48, la primera versión del algoritmo del algoritmo de Dijkstra con punteros hacia atrás) sobre el grafo anterior para encontrar los caminos más cortos desde el vértice 0 a todos los demás. Para ello, rellena la siguiente tabla (haciendo uso de las filas que necesites):

	D						*backpointer*							
iter	0	1	2	3	4	5	0	1	2	3	4	5	*frontier*	*added*
inic.														
1														
2														
3														
4														
5														
6														
7														
8														
9														
10														
...														

b) Indica cuál es el *árbol* de caminos más cortos desde el vértice 0 a todos los demás que devuelve el algoritmo (puedes dibujarlo sobre el grafo).

Algoritmos voraces
25 de abril de 2007 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema **65 puntos**

Un sistema informático con P procesadores iguales debe ejecutar n tareas, cada una con un tiempo de ejecución t_i $(1 \leq i \leq n)$. Un procesador sólo puede ejecutar una tarea en un instante de tiempo y no están permitidas las interrupciones (una vez comienza la ejecución de una tarea, hasta que el procesador no acaba con ella no puede comenzar la ejecución de otra).

Queremos minimizar el tiempo total de espera de todas las tareas (la suma de los tiempos que tiene que esperar cada una de ellas).

Se pide:

a) Señalad en que consistirá una solución del problema, las restricciones que debe cumplir para ser una solución factible y cómo se calcularía el valor de la función objetivo sobre ella (no es preciso hacer una definición formal, pero si lo más precisa posible).

b) Proponed una estrategia voraz (describidla con precisión) que devuelva la solución óptima para el problema ante cualquier entrada.

c) Diseñad un algoritmo que siga dicha estrategia. El algoritmo deberá devolver, para cada tarea, el procesador en el que habrá que ejecutarla y el instante de tiempo en que comienza su ejecución para que la solución devuelta sea la óptima.

d) Indicad (y justificad) cuál es el coste temporal de la estrategia (y del algoritmo) propuestos.

e) Si cada procesador tuviera una velocidad distinta, ¿seguiría siendo válida la estrategia que habéis desarrollado? Justificad vuestra respuesta. Si no es válida, ¿podríais modificarla o diseñar otra que garantizara encontrar la solución óptima para cualquier instancia del problema? Si la respuesta es positiva indicad cómo y, si es negativa, justificadla.

Algoritmos voraces
25 de abril de 2007 (Tarde)

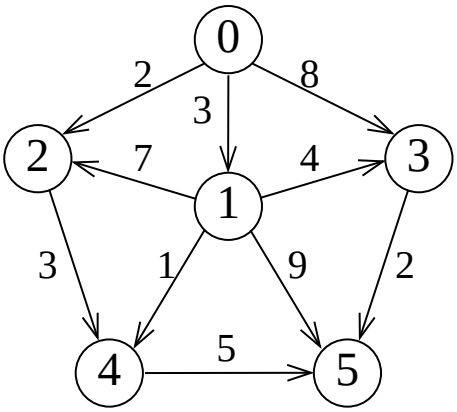
Prueba individual

Nombre: **Grupo:**

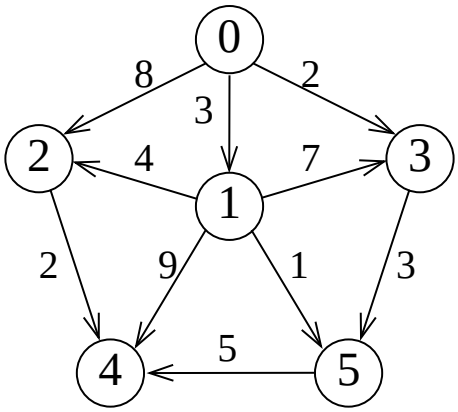
Algoritmo de Dijkstra — 35 puntos

Dado el siguiente grafo dirigido y ponderado:

Modelo 1:



Modelo 2:



a) Haz una traza, iteración a iteración, de la ejecución de `all_shortest_paths_from_source2` (pág. 7-48, la primera versión del algoritmo del algoritmo de Dijkstra con punteros hacia atrás) sobre el grafo anterior para encontrar los caminos más cortos desde el vértice 0 a todos los demás. Para ello, rellena la siguiente tabla (haciendo uso de las filas que necesites):

	D						*backpointer*							
iter	0	1	2	3	4	5	0	1	2	3	4	5	*frontier*	*added*
inic.														
1														
2														
3														
4														
5														
6														
7														
8														
9														
10														
...														

b) Indica cuál es el *árbol* de caminos más cortos desde el vértice 0 a todos los demás que devuelve el algoritmo (puedes dibujarlo sobre el grafo).

Algoritmos voraces
25 de abril de 2007 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema **65 puntos**

En una asignatura de ITIG los alumnos realizan n prácticas a lo largo del curso. La puntuación de cada práctica puede ser: 0 (muy mal), 2.5 (mal), 5 (regular), 7.5 (bien) o 10 (excelente). A partir de las experiencias de otros años, el profesor da una estimación del tiempo t_i ($1 \leq i \leq n$) que necesita invertir un alumno medio en cada práctica i para obtener la máxima puntuación. Si el tiempo invertido por el alumno en la práctica i es menor que t_i, la puntuación obtenida se ve reducida de forma proporcional al tiempo invertido: con un tiempo entre el 75 % y menor que el 100 % de t_i, la puntuación es de 7.5; con un tiempo entre el 50 % y menor que el 75 % de t_i, la puntuación sería de 5; ...

Para obtener el apto en prácticas, hay que obtener un total de P puntos entre todas las prácticas. ¿Cuál sería el tiempo que tendrías que invertir en cada práctica para conseguir el apto de forma que, globalmente, el tiempo dedicado a la asignatura fuera el menor posible?

Se pide:

a) Formalizad el conjunto de soluciones factibles y la función objetivo para el planteamiento del problema.

b) Proponed una estrategia voraz (describidla con precisión) que devuelva la solución óptima para el problema ante cualquier entrada.

c) Diseñad un algoritmo que siga dicha estrategia. El algoritmo deberá devolver, para cada práctica, el tiempo que se deberá emplear para alcanzar la solución óptima así como el tiempo global invertido.

d) Indicad (y justificad) cuál es el coste temporal de la estrategia (y del algoritmo) propuestos.

e) Suponiendo que el tiempo invertido para sacar un 7.5 en cada práctica fuera la mitad que para sacar un 10, que para sacar un 5 bastara con invertir una cuarta parte de t_i y que para sacar un 2.5 fuera suficiente con dedicar una décima parte de t_i a la práctica, ¿seguiría siendo válida la estrategia que habéis desarrollado? Justificad vuestra respuesta. Si no es válida, ¿podríais modificarla o diseñar otra que garantizara encontrar la solución óptima para cualquier instancia del problema? Si la respuesta es positiva indicad cómo y, si es negativa, justificadla.

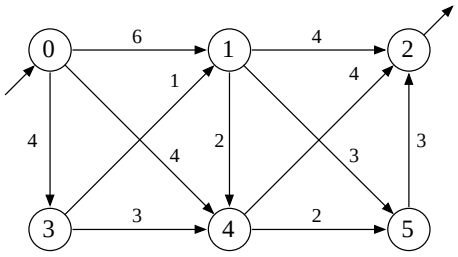
Programación dinámica
10 de mayo de 2007 (Mañana)

Prueba individual

Nombre: **Grupo:**

Problema del camino más corto en un grafo acíclico **50 puntos**

Debes encontrar cuál es el camino más corto que empieza en el vértice $s = 0$ y acaba en el vértice $t = 2$ en el siguiente grafo:



Para ello, se pide:

a) Dibuja el grafo ordenado topológicamente.

b) Haz una traza del algoritmo `dag_shortest_path` (pág 7-34), el que permite recuperar el camino óptimo. Para ello puedes, bien utilizar como base el grafo obtenido en el apartado anterior de forma similar a cómo se hacía en el apartado (g) de la figura 7.23 (pág 7-35), bien rellenar los diccionarios D y B. *No* se pide una traza iteración a iteración, basta con que se muestre el contenido final de las dos estructuras citadas.

c) ¿Cúal es el valor de la solución óptima? ¿En qué estado se encuentra? ¿Qué resultado devuelve el algoritmo? ¿Qué significa este último resultado?

Programación dinámica
9/10 de mayo de 2007 (Tarde)

Prueba individual

Nombre: **Grupo:**

Problema de la mochila discreta **50 puntos**

Dada una mochila con capacidad para cargar $W = 4$ unidades de peso y $N = 5$ objetos que podemos cargar en ella, con pesos $w = [1, 2, 1, 2, 3]$ y valores $v = [2, 15, 9, 5, 13]$, se pide:

a) Dibuja, aprovechando la siguiente cuadrícula, el *grafo extendido de dependencias entre estados* asociado a dicha instancia del problema (utiliza las cajas que necesites, tachando el resto, y traza los arcos entre estados correspondientes al grafo para el problema propuesto):

(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)

b) Haz una traza del algoritmo `knapsack` (pág 7-22), el que permite recuperar la secuencia óptima de objetos. Para ello, escribe arriba de cada uno de los estados del grafo del apartado anterior el valor que el algoritmo almacena en el diccionario `V` y debajo del estado el valor que se almacena en el diccionario de punteros hacia atrás para el estado.

c) ¿Cúal es el valor de la solución óptima? ¿En qué estado se encuentra? ¿Qué resultado devuelve el algoritmo? ¿Qué significa este último resultado?

Programación dinámica
21 de mayo de 2007 (Mañana)

Prueba individual

Nombre: **Grupo:**

Problema **50 puntos**

Un amigo que acaba de comprarse la Wii te la ha prestado durante un tiempo limitado T, junto con J juegos distintos e información sobre dichos juegos. Concretamente, para cada juego j ($1 \leq j \leq J$) sabes el tiempo $t(j)$ que te costará jugar una vez a ese juego y el placer $p(j)$ que te producirá el hacerlo. No te importa jugar más de una vez a un mismo juego pero, como eres un poco obsesivo, una vez has empezado a jugar a uno de los juegos no puedes dejarlo a medias (o dispones de tiempo para completarlo, o no lo juegas).

Deseas saber cuál es el número de veces que debes jugar a cada juego sin sobrepasar el tiempo disponible para disfrutar lo máximo posible.

La siguiente ecuación recursiva te devuelve el valor del máximo placer que puedes obtener jugando a la Wii con las restricciones anteriores si dispones de un tiempo i:

$$P(i) = \begin{cases} 0, & \text{si } i < \min\limits_{1 \leq j \leq J} t(j); \\ \max\limits_{\substack{1 \leq j \leq J: \\ t(j) \leq i}} P(i - t(j)) + p(j), & \text{si } i \geq \min\limits_{1 \leq j \leq J} t(j). \end{cases}$$

donde la llamada $P(T)$ proporciona el máximo placer para el tiempo total en que dispones de la Wii. Se pide:

a) Escribe el algoritmo recursivo con memorización asociado a la anterior ecuación.

b) Representa el grafo de dependencias asociado a la ecuación recursiva para $J = 4$ juegos de duraciones $t = [3, 2, 5, 2]$ si se efectúa la llamada $P(7)$.

c) Indica cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de la ecuación presentada. Justifica brevemente tu respuesta.

Programación dinámica
21 de mayo de 2007 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

Sea el anterior problema:

****

Un amigo que acaba de comprarse la Wii te la ha prestado durante un tiempo limitado T, junto con J juegos distintos e información sobre dichos juegos. Concretamente, para cada juego j ($1 \leq j \leq J$) sabes el tiempo $t(j)$ que te costará jugar una vez a ese juego y el placer $p(j)$ que te producirá el hacerlo. No te importa jugar más de una vez a un mismo juego pero, como eres un poco obsesivo, una vez has empezado a jugar a uno de los juegos no puedes dejarlo a medias (o dispones de tiempo para completarlo, o no lo juegas).

Deseas saber cuál es el número de veces que debes jugar a cada juego sin sobrepasar el tiempo disponible para disfrutar lo máximo posible.

La siguiente ecuación recursiva te devuelve el valor del máximo placer que puedes obtener jugando a la Wii con las restricciones anteriores si dispones de un tiempo i:

$$P(i) = \begin{cases} 0, & \text{si } i < \min\limits_{1 \leq j \leq J} t(j); \\ \max\limits_{\substack{1 \leq j \leq J; \\ t(j) \leq i}} P(i - t(j)) + p(j), & \text{si } i \geq \min\limits_{1 \leq j \leq J} t(j). \end{cases}$$

donde la llamada $P(T)$ proporciona el máximo placer para el tiempo total en que dispones de la Wii.

****

Sobre el problema original se pide:

a) Escribid el algoritmo iterativo que proporciona *la secuencia de decisiones óptima, es decir, las veces que debes jugar a cada juego* para obtener el máximo placer.

A continuación, se proponen las siguientes modificaciones en el enunciado del problema de la Wii:

1. Tanto el placer como el tiempo que tardas en jugar a un juego dependen del número de veces que hayas jugado previamente. Por tanto, la información de la que dispones para cada juego j ($1 \leq j \leq J$) es $p(j, n)$ (el placer que te proporciona jugar n veces al juego) y $t(j, n)$ (el tiempo que tardas en jugar n veces al juego).
2. Como mínimo, quieres jugar una vez a cada uno de los juegos y no estás dispuesto a jugar más de N veces a ninguno de los juegos.

El resto de condiciones se mantienen.
Para este *nuevo problema* se pide:

a) Formalizad el problema en términos de optimización.

b) Plantead la ecuación recursiva de programación dinámica que calcule el máximo placer que se puede obtener con cualquier combinación válida de juegos, indicando cuál debe ser la llamada inicial que se efectúe a dicha función para obtener el valor óptimo.

c) Representad el grafo de dependencias entre las llamadas de la ecuación recursiva para $J = 4$ juegos de duraciones $t = [[3, 6, 9], [2, 4, 6], [5, 10, 15], [2, 4, 6]]$, $T = 7$ y $N = 2$.

d) Indicad cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de vuestra ecuación recursiva que devolviera la *secuencia de decisiones óptima*. Justificad brevemente la respuesta.

e) ¿Sería posible reducir el coste espacial indicado en el apartado anterior si hubiera que diseñar un algoritmo iterativo que sólo encontrara *el valor* del máximo placer? Si es así, indicad cuál sería dicho coste y justificadlo.

Programación dinámica
21 de mayo de 2007 (Tarde)

Prueba individual

Nombre: **Grupo:**

Problema **50 puntos**

Tienes que estudiar para el examen de una asignatura teórica de ITIG. La asignatura consta de M temas y tienes un tiempo limitado T para prepararlos. De cada tema m ($1 \leq m \leq M$) sabes el peso $p(m)$ que tiene en la asignatura y el tiempo $t(m)$ que te cuesta leerte el tema una vez (asumimos que $t(m) < T$ para todo m). En cada lectura de un tema aumenta el nivel de comprensión del mismo (situado entre el 0 % y el 100 %), dato que también conocemos: $c(m, n)$, que representa el nivel de comprensión del tema m tras haberlo leído n veces. Para estar tranquilo has decidido que, como mínimo, vas a leer cada tema al menos una vez. ¡Ah!, y no vale dejarse un tema a medias, si comienzas a leerlo tienes que estar seguro de que vas a poder acabarlo, en caso contrario ni empiezas.

Deseas saber cuantas veces debes leer cada uno de los temas para obtener la máxima calificación en el examen sin superar el tiempo dedicado a prepararlo.

La siguiente ecuación recursiva devuelve el valor de la máxima nota que puedes obtener si tienes que preparar m temas disponiendo de un tiempo i:

$$N(i,m) = \begin{cases} 0, & \text{si } m = 0; \\ -\infty, & \text{si } m > 0 \text{ e } i < t(m); \\ \max\limits_{1 \leq n \leq \lfloor i/t(m) \rfloor} N(i - t(m) * n, m-1) + p(m) * c(m,n), & \text{si } m > 0 \text{ e } i \geq t(m). \end{cases}$$

donde la llamada $N(T, M)$ proporciona el valor de la máxima nota posible. Se pide:

a) Escribe el algoritmo recursivo con memorización asociado a la anterior ecuación.

b) Representa el grafo de dependencias asociado a la ecuación recursiva para los siguientes tiempos $t = [1, 3, 1]$ si se efectúa la llamada $N(6, 3)$.

c) Indica cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de la ecuación presentada. Justifica brevemente tu respuesta.

Programación dinámica
21 de mayo de 2007 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema **100 puntos**

Sea el anterior problema:

****

Tienes que estudiar para el examen de una asignatura teórica de ITIG. La asignatura consta de M temas y tienes un tiempo limitado T para prepararlos. De cada tema m ($1 \leq m \leq M$) sabes el peso $p(m)$ que tiene en la asignatura y el tiempo $t(m)$ que te cuesta leerte el tema una vez (asumimos que $t(m) < T$ para todo m). En cada lectura de un tema aumenta el nivel de comprensión del mismo (situado entre el 0 % y el 100 %), dato que también conocemos: $c(m, n)$, que representa el nivel de comprensión del tema m tras haberlo leído n veces. Para estar tranquilo has decidido que, como mínimo, vas a leer cada tema al menos una vez. ¡Ah!, y no vale dejarse un tema a medias, si comienzas a leerlo tienes que estar seguro de que vas a poder acabarlo, en caso contrario ni empiezas.

Deseas saber cuantas veces debes leer cada uno de los temas para obtener la máxima calificación en el examen sin superar el tiempo dedicado a prepararlo.

La siguiente ecuación recursiva devuelve el valor de la máxima nota que puedes obtener si tienes que preparar m temas disponiendo de un tiempo i:

$$N(i,m) = \begin{cases} 0, & \text{si } m = 0; \\ -\infty, & \text{si } m > 0 \text{ e } i < t(m); \\ \max\limits_{1 \leq n \leq \lfloor i/t(m) \rfloor} N(i - t(m) * n, m-1) + p(m) * c(m,n), & \text{si } m > 0 \text{ e } i \geq t(m). \end{cases}$$

donde la llamada $N(T, M)$ proporciona el valor de la máxima nota posible.

****

Sobre el problema original se pide:

a) Escribid el algoritmo iterativo *con reducción de la complejidad espacial* que permite calcular el valor de la máxima nota posible.

A continuación, se proponen las siguientes modificaciones en el enunciado del problema:

1. Tras cada lectura de un tema, su nivel de comprensión se incrementa de forma lineal, por lo que la función c pasa a tener un único valor por tema: $c(m)$, para todo tema m. Por ejemplo, si el nivel de comprensión del tema 3 es $c(3) = 5\,\%$, cada vez que se haga una lectura del tema la puntuación en el examen se incrementaría en un 5 %, es decir, en $p(3) * 5\,\%$.
2. No es obligatorio leer cada tema al menos una vez (es decir, pueden dejarse temas sin leer).
3. Para reducir el estrés, después de leer un tema hay que descansar obligatoriamente H minutos.

El resto de condiciones se mantienen.
Para este *nuevo problema* se pide:

a) Formalizad el problema en términos de optimización.

b) Plantead la ecuación recursiva de programación dinámica que calcule la máxima nota que se puede obtener con cualquier combinación válida de lecturas, indicando cuál debe ser la llamada inicial que se efectúe a dicha función para obtener el valor óptimo.

c) Representad el grafo de dependencias entre las llamadas de vuestra ecuación recursiva para $T = 9$, $M = 4$, $t = [1, 3, 1, 2]$ y $H = 2$.

d) Indicad cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de vuestra ecuación recursiva que devolviera la *secuencia de decisiones óptima*. Justificad brevemente la respuesta.

e) ¿Sería posible reducir el coste espacial indicado en el apartado anterior si hubiera que diseñar un algoritmo iterativo que sólo encontrara *el valor* de la máxima nota? Si es así, indicad cuál sería dicho coste y justificadlo.

Primer control parcial de Esquemas Algorítmicos (IG24)

31 de marzo de 2007

1. Divide y vencerás — 5 puntos

Sean dos rectas, representadas mediante dos vectores f y g de n elementos en los que los pares $(i, f[i])$ e $(i, g[i])$ representan un punto de cada una de las rectas (para $0 \leq i \leq n-1$), donde i es la coordenada del correspondiente punto en el eje X y $f[i]$ (y $g[i]$) su coordenada en el eje Y. Las dos rectas son estrictamente crecientes con relación a los dos ejes (es decir, $f[0] < f[1] < \ldots < f[n-1]$ y $g[0] < g[1] < \ldots < g[n-1]$), aunque una de ellas crece más rapidamente que la otra (no nos indican cuál). Nos interesaría averiguar, para todo posible valor i dentro del intervalo $(0 \leq i \leq n-1)$, cuáles son los valores de $f[i]$ y $g[i]$ más cercanos (es decir, que puntos de f y g con la misma coordenada en X son más cercanos). Se pide:

a) Diseña un algoritmo recursivo mediante la técnica de divide y vencerás que devuelva los valores $f[i]$ y $g[i]$ más cercanos. El algoritmo debe ser lo más eficiente posible.

b) Indica si el algoritmo presenta un comportamiento uniforme o existe un mejor y un peor caso. Si el comportamiento es uniforme, trata de modificarlo para que ante determinadas instancias pueda finalizar antes su ejecución. Señala en qué condiciones (ante que tipos de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor (da algún ejemplo de cada una de esas entradas).

c) Realiza un análisis de la complejidad temporal y espacial del algoritmo presentado, justificando los costes en el mejor y en el peor de los casos.

d) Si el algoritmo propuesto presenta recursividad por cola, elimínala e indica si alguno de los costes calculados en el apartado anterior varía.

e) ¿Se te ocurre algún algoritmo iterativo sencillo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás (no es necesario que lo implementes, tan sólo descríbelo)? Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el que has diseñado.

2. Búsqueda con retroceso — 5 puntos

Una máquina expendedora de productos debe devolver una cantidad de dinero Q. Para ello, dispone de un número limitado de monedas de N valores distintos. La máquina debe encontrar una combinación de las monedas disponibles que le permita devolver la cantidad solicitada (o indicar que no puede hacerlo).

Por ejemplo, si la cantidad a devolver son Q =3.27 euros, los valores de las monedas son v = [0.01, 0.02, 0.05, 0.1, 0.2, 0.5, 1, 2] y el número de monedas disponibles de cada valor es $m = [4, 3, 2, 6, 4, 3, 2, 0]$ (en el mismo orden de los valores de las monedas), una posible combinación válida se podría expresar cómo $(2, 0, 1, 2, 0, 2, 2, 0)$ (2 monedas de valor 0.01, 1 de valor 0.05, 2 de valor 0.1, 2 de valor 0.5 y dos de valor 1).

Queremos encontrar un algoritmo de *búsqueda con retroceso* que resuelva el problema de encontrar el cambio de una determinada cantidad.

Se pide que:

a) Describas cómo vas a representar los estados s en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolver el problema (si prefieres, puedes utilizar el esquema, en cuyo caso deberás escribir el código de los métodos `is_complete(s)`, `is_factible(s)`, `branch(s)` e `is_promising(s)` e indicar si efectuas alguna acción adicional).

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Es el algoritmo que has desarrollado el más eficiente posible? Si crees que puedes realizar algunas modificaciones que permitan reducir su coste, descríbelas (y justifícalas).

e) Realiza una traza de ejecución de tu algoritmo (de la versión que prefieras, indica cuál) para el ejemplo $Q = 1$, v = [0.1,0.2,0.5,1] y $m = [3, 3, 2, 1]$. Debes representar el árbol de llamadas efectuadas, incluyendo solamente aquellos estados que se generan durante el proceso de búsqueda. Marca de forma especial los estados que se estudian pero se descartan por no superar la condición de ser prometedores. Indica cuál es el resultado devuelto por el algoritmo.

f) Por último, modifica lo que consideres oportuno del algoritmo para que, si hubiera más de una combinación de monedas válida, devuelva aquella que gaste menos monedas.

Segundo control parcial de Esquemas Algorítmicos (IG24)

26 de mayo de 2007

1. Algoritmos voraces — 3.25 puntos

Dos equipos de n miembros, a y b, van a enfrentarse en un concurso de inteligencia (los miembros de cada equipo están numerados entre 1 y n). En el concurso, cada miembro de un equipo participa una única vez en un enfrentamiento cara a cara directo con un miembro del otro equipo. El ganador del enfrentamiento consigue un punto para su equipo. El objetivo de cada equipo es obtener la máxima cantidad de puntos.

Antes de que comience el concurso, cada equipo debe presentar una lista de sus miembros ordenada según el orden de participación en los enfrentamientos. El equipo b ha conseguido con antelación conocer el orden en que el equipo a va a presentar a sus participantes y, además, saber los cocientes intelectuales de todos los miembros de los dos equipos. Asumiendo que no hay dos concursantes con el mismo cociente y que cuando se enfrentan dos personas siempre ganará la de mayor cociente, diseña una estrategia voraz para que el equipo b pueda presentar la lista de sus participantes que le permite ganar la mayor cantidad posible de puntos.

Ejemplo: Cocientes equipo a: c_a = [130, 142, 125, 163, 192, 122], cocientes equipo b: c_b = [190, 143, 131, 120, 180, 119] y orden presentado por el equipo a: l_a =[2, 6, 1, 3, 4, 5]. Ejemplo de solución factible: [(2, 1), (6, 2), (1, 6), (3, 5), (4, 4), (5, 3)], donde cada par representa un enfrentamiento en el que el primer término es el número de concursante del equipo a y el segundo el del equipo b. Con esta solución, el equipo b ganaría 1+1+0+1+0+0=3 puntos.(Una posible solución óptima para este problema sería {(2, 5), (6, 4), (1, 2), (3, 3), (4, 1), (5, 6)} con 1+0+1+1+1+0=4 puntos de ganancia.)

Se pide:

a) Describe con la mayor precisión una estrategia voraz lo más eficiente posible que proporcione la mejor ordenación de los participantes del equipo con información privilegiada.

b) Indica cuál es el coste temporal de la estrategia que has diseñado, justificándolo adecuadamente. Señala las estructuras de datos empleadas.

c) Supón la siguiente variante del problema: el equipo b *sólo* conoce con anterioridad al concurso los cocientes intelectuales de los miembros de los dos equipos. Sin embargo, no hay que presentar las listas de concursantes por anticipado, sino que en este caso el mecanismo del concurso es el siguiente: antes de cada enfrentamiento, el equipo a indica qué miembro de su equipo va a participar y, una vez sabido esto, el equipo b determina cuál es el miembro de su equipo que será el adversario.

   ¿Sigue existiendo alguna solución voraz para el problema que pueda garantizar que se obtendrá siempre la máxima puntuación posible? Si la respuesta es afirmativa, responde a las misma cuestiones a) y b) propuestas para la versión original del problema y, si es negativa, justifícala.

2. Programación dinámica — 4 puntos

Los tripulantes de La Perla Negra, barco pirata capitaneado por Jack Sparrow, están malditos tras haber robado una serie de piezas de oro azteca. Las piezas de oro se pueden agrupar según su valor: hay N diferentes tipos de piezas, con valores $v(n)$, y de cada valor hay un total de $p(n)$ piezas (para n entre 1 y N).

Jack Sparrow ha descubierto un modo de librarse de la maldición sin necesidad de renunciar a todo el tesoro: tiene que llegar hasta el fin del mundo y entregarle al Kraken como ofrenda el cofre que contenía el corazón de Davy Jones completamente lleno de piezas de oro del tesoro (da igual las que sean, lo imprescindible es que llenen el cofre sin dejar espacio y que, como mínimo, haya una pieza de cada tipo).

Tras estudiar el problema, Jack averigua el espacio disponible en el cofre (E) y el espacio que ocupa cada tipo de pieza ($s(n)$, con $1 \leq n \leq N$). El objetivo de Jack es llenar el cofre con la combinación de piezas de oro que presente *menos* valor.

1. Con las condiciones anteriormente expuestas:

   a) Formaliza el problema en términos de optimización.

   b) Plantea la ecuación recursiva asociada al hecho de llenar el cofre con piezas de oro de forma que estas supongan el menor valor posible. Indica cuál debe ser la llamada que se efectue a dicha función para obtener el valor óptimo.

   c) Representa el grafo de dependencias entre llamadas del algoritmo recursivo para el ejemplo $N = 3$, $E = 9$, s = [2, 1, 3] y p = [3, 4, 2], razonando los costes espacial y temporal con los que se puede efectuar el cálculo sobre dicho grafo (indica explícitamente, justificando tu respuesta, si es posible efectuar una reducción de complejidad espacial al realizar el cálculo iterativo).

   d) ¿Podría haber instancias del problema para las que no existiera una solución factible? Razona la respuesta y, si es así, indica las condiciones que se deberían cumplir para que pueda haber una solución factible.

2. Una variante del problema sería la siguiente: Jack se ha dado cuenta de que determinadas piezas pesan mucho y que es posible que el cofre acabe rompiendose si lo rellenara con ese tipo de piezas. Por ello, ha calculado el peso máximo W que es capaz de soportar el cofre y lo que pesa cada tipo de pieza ($w(i)$, con $1 \leq n \leq N$). Por tanto, el problema pasa a ser cómo rellenar completamente el cofre con la combinación de piezas que suponga un menor valor (cogiendo al menos 1 de cada) sin que se rompa el cofre.

   ¿Cómo modificarías el conjunto de soluciones factibles para que recogiera este nuevo planteamiento? ¿Cuál sería ahora la ecuación recursiva que resolviera el problema? ¿Con qué coste temporal y espacial se podría resolver?

3. Programación dinámica **2.75 puntos**

En la preparación de una maratón por etapas se ha diseñado ya el trayecto concreto por el que han de correr los atletas, pero falta decidir dónde empieza y dónde acaba cada una de las E etapas de la maratón. El trayecto completo parte del kilómetro 0 y finaliza en el kilómetro K. Las etapas sólo pueden empezar o acabar en puntos kilómetricos enteros (por ejemplo, en el kilómetro 5, pero no en el 5.3).

El esfuerzo realizado por un corredor en una etapa depende de sus kilómetros de inicio y finalización, así como del número de etapas que ya ha cubierto. Con $f(i, j, e)$ cuantificamos el esfuerzo que supone recorrer los kilómetros del i al j en la etapa e.

Por motivos publicitarios cada etapa tiene fijado su comienzo en una zona predeterminada, por lo que el punto exacto de comienzo de cada etapa e está restringido a unos pocos kilometros ya conocidos, que se encuentran entre $k_{\text{mín}}(e)$ y $k_{\text{máx}}(e)$. Por ejemplo, si $E = 4$, $k_{\text{mín}} = [0, 3, 7, 18]$ y $k_{\text{máx}} = [0, 5, 13, 20]$, la etapa 1 sólo puede comenzar en el km. 0, la etapa 2 podría comenzar entre el km. 3 y el 5, la etapa 3 entre el km. 7 y el 13 y, por último, la etapa final sólo podría empezar entre el km. 18 y el 20.(A partir de esta restricción puedes asumir que se cumple que $k_{\text{máx}}(e) < k_{\text{mín}}(e + 1)$ para todas las etapas e.)

Debes averiguar, empleando la técnica de programación dinámica, dónde debe empezar y acabar exactamente cada una de las E etapas para garantizar que el recorrido suponga el *menor* esfuerzo posible a los corredores. Como ayuda, la siguiente ecuación recursiva de programación dinámica permite resolver el problema de obtener el valor del mínimo esfuerzo:

$$F(k, e) = \begin{cases} 0, & \text{si } e = 0; \\ \min\limits_{k_{\text{mín}}(e) \le k' \le k_{\text{máx}}(e)} F(k', e-1) + f(k', k, e), & \text{si } e > 0. \end{cases}$$

La llamada $F(K, E)$ proporciona dicho valor.
Se pide:

a) Representa el grafo de dependencias para el problema en el que $E = 3$, $K = 10$, $k_{\text{mín}} = [0, 3, 6]$ y $k_{\text{máx}} = [0, 4, 8]$. Indica qué representa un estado (k, e) y un arco entre dos estados (k', e') y (k, e) dentro del grafo.

b) Diseña un algoritmo iterativo *lo más eficiente posible* que devuelva la *duración que debe tener cada etapa* para garantizar que se realiza el mínimo esfuerzo.

c) Indica cuál es el coste temporal y espacial del algoritmo diseñado justificando brevemente tu respuesta. Intenta expresar los costes de la forma más ajustada posible.

d) ¿Crees que sería posible reducir el coste espacial si únicamente nos interesará obtener el valor del mínimo esfuerzo? Si es así, índica cuál sería dicho coste y justifícalo.

Examen convocatoria ordinaria de Esquemas Algorítmicos (IG24)

15 de junio de 2007

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo):
Parte 1 (Preg. 1 y 2) ☐ **Parte 2 (Preg. 3 y 4)** ☐ **Partes 1 y 2** ☐ **Renuncio a la evaluación alternativa** ☐
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Divide y vencerás — 2,5 puntos

Dado un vector v, un desplazamiento cíclico de valor k es la operación que proporciona como resultado el vector $v[k:]+v[:k]$. Tenemos un vector de enteros dispuestos en orden no decreciente que ha sido sometido a un desplazamiento cíclico de valor k. Deseamos conocer el valor de k y hemos de diseñar un algoritmo eficiente que lo averigüe.

Si aplicamos al vector $[1,3,7,10,12,38,98]$ un desplazamiento cíclico de valor 3, obtenemos $[10,12,38,98,1,3,7]$. Pues bien, el algoritmo que has de diseñar recibirá, por ejemplo, el vector $[10,12,38,98,1,3,7]$ y devolverá el valor 3.

Has de diseñar un algoritmo de *divide y vencerás* que, dado un vector como el descrito, averigüe *en tiempo sublineal* el valor del desplazamiento cíclico sufrido. Se pìde:

a) Diseña un algoritmo recursivo mediante la técnica de divide y vencerás que permita resolver este problema.

b) Analiza y justifica la complejidad temporal y espacial del algoritmo desarrollado en el apartado anterior, indicando si presenta un comportamiento uniforme o existe un mejor y un peor caso.

c) Indica si el algoritmo presenta recursión por cola y, en caso afirmativo, implementa el algoritmo iterativo que resulta de eliminarla, además de indicar en qué mejora la eficiencia de la implementación iterativa con respecto a la recursiva.

2. Búsqueda con retroceso — 2,5 puntos

Dado un vector A con N enteros, deseamos saber si hay alguna forma de particionarlo en K subconjuntos cuya suma de elementos sea exactamente M.

Por ejemplo, sean $A=[2,1,4,2,3]$, $M=4$ y $K=3$; el conjunto $\{\{1,3\},\{2,2\},\{4\}\}$ es una partición de A tal que los 3 subconjuntos que la forman suman 4.

Debes desarrollar un algoritmo de *búsqueda con retroceso* que, a partir de A, M y K, obtenga una asignación de cada elemento de A a uno de los K subconjuntos de manera que todos ellos sumen M. Para ello, las soluciones factibles deberían representarse mediante N-tuplas. Se pide:

a) Describe cómo vas a representar los estados en la resolución del problema planteado.

b) Escribe un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolver el problema.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Crees que es posible mejorar la eficiencia del algoritmo que has desarrollado introduciendo alguna variable o estructura de datos adicional o haciendo algún tipo de preproceso? Si es así, describe con claridad la mejora o mejoras que introducirías en el algoritmo e indica si ello afectaría a los costes previamente calculados.

3. Voraces — 1,5 puntos

Hemos de almacenar N libros en una estantería y para ello hemos de diseñar la estantería. La anchura de la estantería es de L milímetros (L es un valor fijo que nos suministran), por lo que nos queda por determinar cuántos estantes contendrá y la altura que habrá que dejar entre cada par de estantes. La altura de un estante vendrá dada por la altura de su libro más alto. La altura de la estantería será la suma de las alturas de sus estantes.

Los N libros hay que colocarlos en la estantería en un orden prefijado que no podemos cambiar. El libro i-ésimo, para i entre 1 y N, tiene altura a_1 y grosor g_i (ambas cantidades expresadas en milímetros).

Tengo un algoritmo voraz consistente en lo siguiente: relleno un primer estante con tantos libros como puedo (en el orden indicado) y, cuando llego a un libro que no cabe (debido al grosor de la estantería), coloco un nuevo estante y repito el procedimiento para el resto de los libros. Se pide:

a) Implementa el algoritmo en python (debe devolver la altura que tendrá cada estante).

b) Razona si el algoritmo voraz permite construir siempre una estantería con el menor número de estantes posible. Si no es así, justifícalo con un contraejemplo.

c) Razona si el algoritmo voraz permite construir siempre una estantería de la menor altura posible. Si no es así, justifícalo con un contraejemplo.

d) Analiza el coste temporal y espacial del algoritmo.

4. Programación dinámica **3,5 puntos**

Hemos de almacenar N libros en una estantería y para ello hemos de diseñar la estantería. La anchura de la estantería es de L milímetros (L es un valor fijo que nos suministran), por lo que nos queda por determinar cuántos estantes contendrá y la altura que habrá que dejar entre cada par de estantes. La altura de un estante vendrá dada por la altura de su libro más alto. La altura de la estantería será la suma de las alturas de sus estantes.

Los N libros hay que colocarlos en la estantería en un orden prefijado que no podemos cambiar. El libro i-ésimo, para i entre 1 y N, tiene altura a_1 y grosor g_i (ambas cantidades expresadas en milímetros).

Deseamos diseñar un algoritmo mediante la técnica de *programación dinámica* que, dados los vectores a y g y el valor L, proporcione la altura de cada uno de los estantes de la estantería *con menor altura posible* que pueda albergar todos los libros en el orden en el que se nos suministran. Se pide:

a) Formaliza el problema en términos de optimización.

b) Diseña una ecuación recursiva que calcule la altura de la estantería más baja capaz de contener todos los libros, indicando cuál debe ser la llamada que se haga a dicha ecuación para resolver el problema.

c) Representa el grafo de dependencias entre las llamadas recursivas para la instancia $N = 8$, $L = 5$, $g = [2, 1, 2, 3, 2, 2, 1, 3]$ y $a = [1, 6, 5, 3, 7, 1, 2, 9]$.

d) Diseña un algoritmo iterativo que recorra el anterior grafo y devuelva la altura mínima de la estantería, indicando y justificando sus costes espacial y temporal.

Examen segunda convocatoria de Esquemas Algorítmicos (IG24)

4 de septiembre de 2007

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo):
Parte 1 (Preg. 1 y 2) ☐ **Parte 2 (Preg. 3 y 4)** ☐ **Partes 1 y 2** ☐ **Renuncio a la evaluación alternativa** ☐
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Divide y vencerás — 2,5 puntos

Dado un vector a de n enteros, se dice que contiene un elemento mayoritario si uno de sus elementos aparece en el vector más de $n/2$ veces. Diseña un algoritmo mediante la técnica de *divide y vencerás* que, a partir de un vector a, devuelva su elemento mayoritario si éste existe o un valor especial en caso contrario.
Pista: para facilitar la elaboración del algoritmo, ten en cuenta que un elemento puede ser mayoritario si es mayoritario en la primera o en la segunda mitad del vector. Además, si en un momento dado tienes un elemento candidato a mayoritario, puedes hacer uso de una función, `comprobar(a, i, k, x)`, que devuelve `True` si elemento x es mayoritario en el subvector `a[i:k]` y `False` en caso contrario. El coste de `comprobar` es $O(k-i)$.

Se pìde:

a) Diseña un algoritmo recursivo *mediante la técnica de divide y vencerás* que permita resolver este problema de la forma más eficiente posible.

b) Analiza y justifica la complejidad temporal y espacial del algoritmo desarrollado en el apartado anterior, indicando si presenta un comportamiento uniforme o existe un mejor y un peor caso. Si el comportamiento es uniforme, modifícalo para que tenga un mejor y un peor caso. Finalmente, señala en qué condiciones (ante qué tipo de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor.

2. Búsqueda con retroceso — 2,5 puntos

El servicio de espionaje británico dispone de N agentes secretos, numerados del 001 al N, pero cuyas identidades se desconocen. A partir de las misiones realizadas por dichos agentes, se sabe qué agentes se relacionan con qué otros agentes para llevar a cabo dichas misiones.

Un espía ha logrado acceder a información confidencial del servicio secreto, concretamente a las identidades privadas de todos los agentes, pero sin saber qué código le corresponde a cada agente. Tras una etapa de seguimiento a las personas, el espía también ha logrado averiguar las relaciones que mantienen dichos agentes en sus misiones.

El siguiente ejemplo ilustra cuál es la información disponible acerca de los agentes:

- Número de agentes: $N = 7$ (numerados del 001 al 007).
- Nombres de los agentes: `nombres`=[Bond, Marlowe, Holmes, Spade, Hammer, Bourne, Maigret].
- Relaciones entre los agentes (a partir de sus códigos secretos):

`rel_cod`	001	002	003	004	005	006	007
001	-	×		×	×		×
002	×	-			×	×	×
003			-		×		×
004	×			-		×	
005	×	×	×		-		×
006		×		×		-	×
007	×	×	×		×	×	-

- Relaciones entre los agentes (a partir de sus nombres reales):

`rel_nom`	Bond	Marlowe	Holmes	Spade	Hammer	Bourne	Maigret
Bond	-	×		×	×	×	×
Marlowe	×	-		×	×	×	
Holmes			-	×		×	
Spade	×	×	×	-	×		
Hammer	×	×		×	-		×
Bourne	×	×	×			-	
Maigret	×				×		-

Dados dos códigos de agentes, i y j, `rel_cod[i][j]` devuelve `True` si existe la relación y `False` si no existe, y, dados dos nómbres de agentes, $n1$ y $n2$, `rel_nom[n1][n2]` hace lo mismo.

Nos interesa conocer una posible asignación de códigos numéricos secretos a nombres de agente de forma que sea coherente con la información disponible en las dos tablas de relaciones entre los agentes.

En el caso del ejemplo anterior, la asignación ((001, Spade),(002, Marlowe), (003, Maigret), (004, Holmes), (005, Hammer), (006, Bourne), (007, Bond)) cumple lo indicado.

Debes desarrollar un algoritmo de *búsqueda con retroceso* que, dado el número de agentes, el vector de nombres reales y las dos tablas de relaciones, obtenga una asignación válida de códigos a nombres de agentes. Para ello se pide que:

a) Describas cómo vas a representar los estados s en la resolución del problema planteado (en qué van a consistir las tuplas: cuantos elementos van a contener, valor posible de esos elementos).

b) Escribas un algoritmo de búsqueda con retroceso que resuelva el problema, indicando cuál sería la llamada inicial que harías para resolver el problema.

c) A la vista del código desarrollado en el apartado anterior, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

d) ¿Crees que es posible mejorar la eficiencia del algoritmo que has desarrollado introduciendo alguna variable o estructura de datos adicional o haciendo algún tipo de preproceso? Si es así, describe con claridad la mejora o mejoras que introducirías en el algoritmo e indica si ello afectaría a los costes previamente calculados.

3. Voraces — 1,5 puntos

Un profesor con A alumnos en su asignatura ha preparado $2A$ trabajos. Cada alumno debe hacer 2 trabajos. A cada alumno se le ha pedido que puntue cada trabajo entre 0 y 10 según el interés que para él tendría realizar el trabajo. Por tanto, se dispone de una matriz P, donde cada celda $P[a][t]$ contiene la puntuación que el alumno a ($1 \leq a \leq A$) otorga al trabajo t ($1 \leq t \leq 2A$).

Deseamos asignar trabajos a los alumnos de modo que la satisfacción global (suma de puntuaciones) sea máxima. Para resolver el problema hemos diseñado las siguientes *estrategias voraces*:

1. Se considera el primer alumno y se le asignan los trabajos que prefiere; a continuación, se asignan al segundo alumno sus dos trabajos preferidos de entre los no asignados al primero; y así sucesivamente.
2. Para cada alumno se calcula la suma de las puntuaciones de sus dos trabajos preferidos y se ordena a los alumnos a partir de dicho valor de mayor a menor valor; considerando ese orden, se aplica el método del apartado anterior.
3. Se construye un máx-heap con todas la puntuaciones de la matriz (junto a las que se almacenan el alumno y trabajo asociados a cada puntuación); se van extrayendo los elementos del máx-heap y asignando los trabajos a los alumnos según el orden de extracción hasta que todos los alumnos tengan asignados sus dos trabajos (si se extrae un trabajo que le corresponde a un alumno que ya tiene asignados sus dos trabajos o se extrae un trabajo que ya ha sido asignado a otro alumno, dicho elemento se descarta).

Debes justificar *para cada estrategia* si es correcta (si está garantizado que ante una entrada válida *siempre* devuelve la solución óptima) o no (en el caso en que no lo sea, debes demostrarlo con un contraejemplo) y debes indicar, razonándolo, el coste temporal de la estrategia.

4. Programación dinámica — 3,5 puntos

Un andarín va a realizar una ruta turística a pie durante varios días. A lo largo de la ruta va a pasar por delante de M mesones numerados, en orden de salida a llegada de la ruta, del 1 al M (por simplicidad, supondremos que el punto de partida y de llegada de la ruta son dos mesones ficticios, numerados respectivamente como 0 y $M+1$).

El coste del alojamiento y/o manutención en un mesón i ($0 \leq i \leq M+1$) viene dado por $c(i)$ ($c(0)$ y $c(M+1)$ son, evidentemente, 0). El tiempo (en horas) y la distancia (en kilómetros) que le costaría llegar al andarín de un mesón i al siguiente $i+1$ son conocidos y son, respectivamente, $t(i)$ y $d(i)$, ($0 \leq i \leq M$). También sabemos que el andarín no puede desplazarse durante más de H horas ni durante más de K kilómetros sin parar a alojarse en un mesón para descansar (puedes asumir que se cumple al menos que $t(i) \leq H$ y que $d(i) \leq K$, para todo $0 \leq i \leq M$.

Queremos saber cual es el gasto mínimo que le supone al andarín realizar la ruta. Se pide:

a) Formaliza el problema en términos de optimización,

b) plantea la ecuación recursiva que calcula el mínimo gasto, indicando cuál debe ser la llamada recursiva que se efectue a dicha ecuación para resolver el problema,

c) representa el grafo de dependencias entre las llamadas recursivas para la instancia $M = 8$, $H = 5$, $K = 20$, $t = [3, 2, 2, 1, 1, 2, 4, 2, 1]$ y $d = [14, 13, 10, 5, 2, 4, 10, 6, 3]$, y

d) diseña un algoritmo iterativo lo más eficiente posible que recorra el anterior grafo y devuelva el gasto mínimo con que puede realizarse la ruta y los mesones en los que debe detenerse el andarín, indicando y justificando sus costes espacial y temporal.

Análisis de algoritmos y Estructuras de datos
27 de febrero de 2006 (Mañana)

Prueba individual

Nombre: **Grupo:**

Análisis de algoritmos

Los dos siguientes algoritmos calculan, a partir de una matriz M de valores booleanos y de talla $n \times n$, el cierre transitivo de dicha matriz:

```
1 def cierre_transitivo_1(M):
2     n = len(M)
3     R = [ [ M[i][j] for j in xrange(n)] for i in xrange(n)]
4     for k in xrange(n):
5         for i in xrange(n):
6             for j in xrange(n):
7                 R[i][j] = R[i][j] or (R[i][k] and R[k][j])
8     return R
```

```
1 def cierre_transitivo_2(M):
2     n = len(M)
3     R = [ [ M[i][j] for j in xrange(n)] for i in xrange(n)]
4     for k in xrange(n):
5         for i in xrange(n):
6             if R[i][k]:
7                 for j in xrange(n):
8                     if R[k][j]: R[i][j] = True
9     return R
```

Indica y justifica cuál es el coste temporal asintótico de ambos, distinguiendo entre el mejor y el peor de los casos si los hubiera. Si tuvieras que elegir uno de los dos algoritmos para resolver el problema, ¿con cuál te quedarías? Justifica tu respuesta.

Prueba colectiva

Grupo:
Asistentes:

1. MFset modificado

Deseamos implementar un MFset que en todo instante nos permita conocer, dado un elemento cualquiera, el número total de elementos que pertenecen al subconjunto en el que está incluido. ¿Cómo modificarías un MFset para poder ofrecer esta nueva operación con el menor coste temporal posible? Detalla como afectaría a las tres operaciones básicas sobre MFsets (creación de un nuevo subconjunto a partir de un elemento, obtención del representante de un subconjunto dado un elemento del mismo y fusión de dos subconjuntos dados un elemento de cada uno de ellos) y cual es el coste temporal y espacial de la operación suma.

2. Análisis de algoritmos

Una empresa de seguridad ha instalado un sistema de reconocimiento automático de placas de matrícula en la puerta de una fábrica. El primer día registró la entrada de n vehículos y almacenó en un vector a sus matrículas. El segundo día volvió a registrar la entrada de n vehículos y almacenó las matrículas en un segundo vector b. El jefe de seguridad quiso saber si los dos días entraron exactamente los mismos vehículos. Pensó que hacer la comprobación manualmente supondría una inversión excesiva de tiempo, así que encargó al machaca-programador del departamento de informática que diseñara una función en Python para resolver el problema. Esta es la solución que ideó:

```
1  def son_las_mismas(a, b):
2      for i in xrange(len(a)):
3          esta_lo_que_busco = False
4          for j in xrange(len(b)):
5              if a[i] == b[j]:
6                  esta_lo_que_busco = True
7                  break
8          if not esta_lo_que_busco:
9              return False
10     return True
```

Se pide:

a) Un análisis del coste temporal asintótico para el mejor y el peor de los casos (indicando expresamente cuando se produce cada uno de ellos). Expresad el coste en función de n, el tamaño de los vectores a y b.

b) El método *sort* ordena de menor a mayor el contenido de un vector (por ejemplo, si a es un vector, $a.sort()$ lo ordena) en tiempo $O(n \lg n)$ y $\Omega(n)$. Sabiendo esto, ¿podéis diseñar un algoritmo más eficiente? Si es así, describidlo (usando Python o como prefiráis, pero con absoluta claridad) y analizad su coste temporal asintótico en el mejor y en el peor de los casos.

Análisis de algoritmos y Estructuras de datos
27 de febrero de 2006 (Tarde)

Prueba individual

Nombre: **Grupo:**

Análisis de algoritmos

Los dos siguientes algoritmos calculan, a partir de un vector de enteros positivos a, un vector de igual talla cuyo elemento de índice i contiene $\sum_{0 \leq i} a[i]$:

```
1 def vector_sumas_1(a):
2     b = [0] * len(a)
3     for i in xrange(len(a)):
4         s = 0
5         for k in xrange(i+1):
6             s += a[k]
7         b[i] = s
8     return b
```

```
1 def vector_sumas_2(a):
2     b = [0] * len(a)
3     b[0] = a[0]
4     for i in xrange(1,len(a)):
5         b[i] = b[i-1] + a[i]
6     return b
```

Indica y justifica cuál es el coste temporal asintótico de ambos, distinguiendo entre el mejor y el peor de los casos si los hubiera. Si tuvieras que elegir uno de los dos algoritmos para resolver el problema, ¿con cuál te quedarías? Justifica tu respuesta.

Prueba colectiva

Grupo:
Asistentes:

1. Selección/diseño de una estructura de datos

Un científico necesita disponer de una estructura de datos T en la que almacenar un máximo de m valores enteros para realizar una serie de simulaciones. Un mismo número entero puede aparecer más de una vez, habiendo como máximo n números enteros distintos (n será normalmente mucho menor que m). La estructura debe ser lo más eficiente posible en las siguientes operaciones:

1. añadir un número a T,
2. consultar el valor mínimo de T,
3. eliminar el valor mínimo de T (si hubiera más de una aparición de dicho número, eliminar una cualquiera de ellas),

Debes señalar brevemente como implementarías esta estructura de datos y cómo y con que coste temporal se podría ejecutar cada una de las tres operaciones anteriormente citadas si nos imponen una restricción: *el espacio que podemos emplear está acotado por* $O(n)$ (recuerda que m podría ser mucho mayor que n).

2. Análisis de algoritmos

Una empresa de transporte tiene que enviar n contenedores de distintos pesos a un mismo destino utilizando su flota de camiones. Los camiones son idénticos y soportan la misma carga máxima, C. Cada contenedor tiene un peso $p_0, p_1, \ldots, p_{n-1}$ (suponemos que $p_i \leq C$ para todo i). A la empresa le gustaría organizar los contenedores de forma que se utilice el menor número posible de camiones en el transporte.

El siguiente algoritmo proporciona una buena aproximación a cuál es dicho número de camiones. Inicialmente se ordenan los contenedores de mayor a menor peso y se considera que como mínimo será necesario disponer de tantos camiones como contenedores hay, con una capacidad de carga máxima (C) (líneas 2 a 4). La estrategia seguida a continuación considera cada contenedor e intenta introducirlo en el camión en el que queda menos capacidad de carga disponible. Tras encontrar un camión en el que cabe el contenedor e introducirlo, se reordenan los camiones de forma que siempre estén organizados de menor a mayor capacidad de carga (líneas 10 a 12). Finalmente, y tras procesar todos los contenedores, se cuenta y devuelve el número mínimo de camiones necesarios (líneas 14 a 20):

```
1  def num_camiones_minimo(p, C):
2      P = sorted(p)
3      P.reverse()
4      capac_disp_camion = [C]*len(p)
5      for i in xrange(len(p)):
6          for j in xrange(len(p)):
7              if capac_disp_camion[j] >= P[i]:
8                  capac_disp_camion[j] -= P[i]
9                  k=j
10                 while k>1 and capac_disp_camion[k]<capac_disp_camion[k-1]:
11                     capac_disp_camion[k],capac_disp_camion[k-1]=capac_disp_camion[k-1],capac_disp_camion[k]
12                     k-=1
13                 break
14     num_cam=0
15     for i in xrange(len(p)):
16         if capac_disp_camion[i] < C:
17             num_cam +=1
18         else:
19             break
20     return num_cam
```

Se pide un análisis del coste temporal asintótico para el mejor y el peor de los casos (indicando expresamente cuando se produce cada uno de ellos). Expresad el coste en función de n, el número de contenedores.

Algoritmos voraces
15 de marzo de 2006 (Mañana)

Prueba individual

Nombre: **Grupo:**

Código de Huffman — Modelo 1

En un partido de voleibol se pueden producir 6 distintos resultados: $(3,0), (3,1), (3,2), (2,3), (1,3), (0,3)$. La siguiente tabla recoge, para cada posible resultado, el número total de partidos finalizados en la actual liga masculina de voleibol:

Resultado	$(3,0)$	$(3,1)$	$(3,2)$	$(2,3)$	$(1,3)$	$(0,3)$
Número de partidos	155	90	105	80	65	100

Deseamos encontrar la codificación binaria óptima de los resultados mediante el algoritmo de Huffman tomando la anterior tabla como medida de las frecuencias de cada resultado. Se pide:

a) Dibuja el árbol binario de decodificación que construiría el algoritmo de Huffman basándose en los anteriores datos.

b) Según dicho árbol, indica qué código binario le correspondería a cada resultado posible.

c) El anterior árbol es devuelto por el algoritmo mediante una representación de listas de listas: escribe la lista correspondiente a dicho árbol.

Código de Huffman — Modelo 2

En un partido de voleibol se pueden producir 6 distintos resultados: $(3,0), (3,1), (3,2), (2,3), (1,3), (0,3)$. La siguiente tabla recoge, para cada posible resultado, el número total de partidos finalizados en la actual liga femenina de voleibol:

Resultado	$(3,0)$	$(3,1)$	$(3,2)$	$(2,3)$	$(1,3)$	$(0,3)$
Número de partidos	220	50	85	30	90	120

Deseamos encontrar la codificación binaria óptima de los resultados mediante el algoritmo de Huffman tomando la anterior tabla como medida de las frecuencias de cada resultado. Se pide:

a) Dibuja el árbol binario de decodificación que construiría el algoritmo de Huffman basándose en los anteriores datos.

b) Según dicho árbol, indica qué código binario le correspondería a cada resultado posible.

c) El anterior árbol es devuelto por el algoritmo mediante una representación de listas de listas: escribe la lista correspondiente a dicho árbol.

Algoritmos voraces
15 de marzo de 2006 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Problema

Un mensajero que trabaja para una compañía de reparto de paquetes postales cobra una cantidad fija por cada entrega que realiza a lo largo de un día. Su trabajo siempre es el mismo: sale del almacén con el paquete correspondiente, llega al destino a la hora fijada para la entrega y vuelve al almacén, donde esperará hasta que le toque realizar la siguiente entrega por el mismo procedimiento.

El mensajero puede elegir los paquetes que entregará durante el día. Para ello, quiere hacer uso de la información que la compañia postal facilita al comienzo de cada jornada: de los n envíos que hay que realizar ese día (y que deben efectuar entre todos los mensajeros de la empresa) conoce, para cada envío i (están numerados entre 1 y n) la hora de la entrega del paquete (h_i) y los minutos que cuesta desplazarse hasta el lugar de destino del paquete (t_i) y regresar al almacén (t'_i). Por tanto, la información disponible para los n paquetes será $p = \{(h_1, t_1, t'_1), (h_2, t_2, t'_2), \ldots, (h_n, t_n, t'_n)\}$.

Diseña una estrategia voraz que permita al mensajero elegir aquella combinación de repartos que contenga la mayor cantidad posible de entregas durante el día para así cobrar más.

Ejemplo: Dado un día con 8 repartos con la siguiente información: $p = \{$(9:00,17,20), (12:15,13,13), (10:45,28,30), (14:25,30,25), (11:30,8,8), (16:30,20,22), (8:30,20,20), (12:00,15,10)$\}$, una solución factible (aunque no la óptima) sería la formada por las entregas ((8:30,20,20), (10:45,28,30), (12:15,13,13), (16:30,20,22)), lo que le permitiría realizar 4 entregas ese día. Tened cuidado con la interpretación correcta del triplete de valores correspondiente a un paquete: en la entrega (9:00,17,20), las 9:00 es la hora de entrega, el mensajero invierte 17 minutos en desplazarse para entregar el paquete y otros 20 para volver al almacén, por lo que tendría que salir del almacén a las 8:43 y estaría de vuelta a las 9:20. Con esta restricción no es posible, por ejemplo, que si el mensajero realiza la entrega (12:00,15,10) pueda realizar la entrega (12:15,13,13).

Se pide:

a) Formalizad el conjunto de soluciones factibles y la función objetivo para el planteamiento del problema.

b) Proponed una estrategia voraz que devuelva la solución óptima para el problema ante cualquier entrada válida.

c) Diseñad un algoritmo que siga dicha estrategia.

d) Indicad (y justificad) cuál es el coste temporal de la estrategia (y del algoritmo) propuestos.

e) Suponiendo que al mensajero le interesa, en el caso de haber más de una combinación de paquetes óptima, quedarse con aquella que le permita acabar lo antes posible la jornada laboral, ¿funcionaría correctamente la estrategía que habéis diseñado? Justificad la respuesta y, si es negativa, indicad (comentándola) si es posible emplear alguna otra estrategia voraz para resolver la variante planteada. Responded a las mismas cuestiones para el caso en el que lo que quiere el mensajero es, ante varías combinaciones óptimas, seleccionar la que le permita comenzar lo más tarde posible.

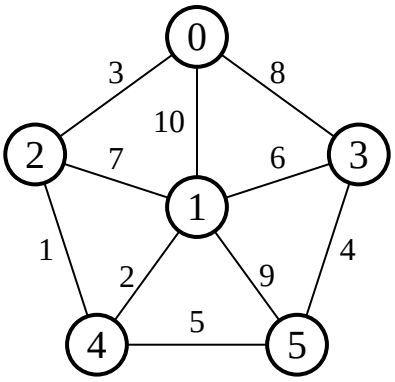
Algoritmos voraces
15 de marzo de 2006 (Tarde)

Prueba individual

Nombre: **Grupo:**

Algoritmo de Prim **Modelo 1**

Dado el siguiente grafo no dirigido y ponderado,

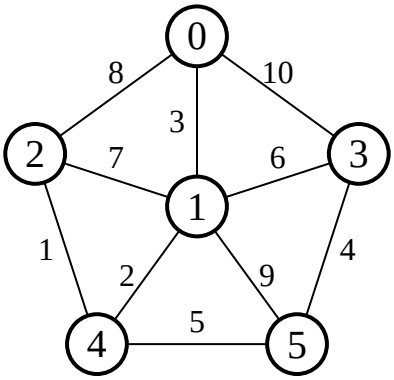


a) Indica cuál sería el árbol de recubrimiento mínimo que devolvería el algoritmo de Prim si el vértice elegido en primer lugar es el 0 (escribe las aristas en el orden en que las devolvería el algoritmo).

b) Haz una traza, iteración a iteración, del algoritmo `prim2.py` (pág. 5-25 y 5-26). Para ello, rellena la siguiente tabla:

	touched	*MST*	*frontier*
0			
1			
2			
3			
4			
5			
6			
7			
...			

Dado el siguiente grafo no dirigido y ponderado,



a) Indica cuál sería el árbol de recubrimiento mínimo que devolvería el algoritmo de Prim si el vértice elegido en primer lugar es el 5 (escribe las aristas en el orden en que las devolvería el algoritmo).

b) Haz una traza, iteración a iteración, del algoritmo `prim2.py` (pág. 5-25 y 5-26). Para ello, rellena la siguiente tabla:

	touched	*MST*	*frontier*
0			
1			
2			
3			
4			
5			
6			
7			
...			

Algoritmos voraces
15 de marzo de 2006 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Problema

Un granjero virtual tiene que mantener una granja formada por n tamagotchis de distintas formas y tamaños. El granjero dispone de una tabla $t = (t_1, t_2, \ldots, t_n) \in \mathbb{R}^n$ en la que, para cada tamagotchi i ($1 \leq i \leq n$), el valor t_i indica la ración de alimentos y de mimos (que son coincidentes) que da cada día al tamagotchi i para mantenerle muy contento.

Por un error informático, que el granjero no podrá reparar hasta el día siguiente, se ha producido una reducción de la cantidad total de alimentos y de mimos de que dispone para repartir entre sus tamagotchis (la reducción ha sido muy importante en los alimentos y algo menos en los mimos). El granjero está desesperado, ya que los tamagotchis son muy sensibles y si cada día no reciben una cantidad mínima de alimentos y/o mimos mueren.

La cantidad total de alimentos de la que dispone el granjero para pasar el día es $A \in \mathbb{R}$ y la de mimos es $M \in \mathbb{R}$ (donde $M \geq 2 * A$). Para sobrevivir durante un día un tamagotchi necesita recibir al menos una de dos opciones: o bien la mitad de la ración de alimentos que le corresponde habitualmente ($1/2 * t_i$) o bien una ración completa de los mimos que recibe diariamente (t_i). Sabiendo esto, diseña una estrategia voraz que permita al granjero repartir la comida y los mimos de que dispone entre los tamagotchis de forma que consiga salvar al mayor número posible de ellos.

Ejemplo: Raciones de comida y mimos que reciben habitualmente 5 tamagotchis: $t = [10, 19, 15, 12, 6]$. Alimentos y mimos disponibles para pasar el día de la avería: $A = 13$ y $M = 30$. Una solución factible sería ((0,1), (0.5,0), (0,0), (0,1), (0.5,0)), donde ha gastado $0.5 * 19 + 0.5 * 6 = 12.5$ alimentos y $1 * 10 + 1 * 12 = 22$ mimos. El número de tamagotchis que sobrevivirían con esta solución es 4.

Se pide:

a) Formalizad el conjunto de soluciones factibles e indicad, a partir de él, cómo se calcularía el valor de la función objetivo sobre una solución factible (esto no es preciso formalizarlo).

b) Proponed una estrategia voraz que devuelva la solución óptima para el problema ante cualquier entrada válida.

c) Diseñad un algoritmo que siga dicha estrategia. El algoritmo deberá devolver, para cada tamagotchi, el porcentaje (sobre la ración de alimentos y de mimos que le corresponden habitualmente) que recibirá para alcanzar la solución óptima.

d) Indicad (y justificad) cuál es el coste temporal de la estrategia (y del algoritmo) propuestos.

e) Suponiendo que la ración habitual de alimentos y mimos de todos los tamagotchi fuera la misma, ¿podríais proponer una estrategia más sencilla (y con menor coste temporal) para resolver el problema? ¿cuál sería su coste? En este supuesto, si sólo estuvierais interesados en saber el número de tamagotchis supervivientes, ¿cómo y con coste temporal podríais averiguarlo?

Programación dinámica
4 de mayo de 2006 (Mañana)

Prueba individual

Nombre: **Grupo:**

Problema de la mochila discreta **Modelo 1**

Dada una mochila con capacidad para cargar $W = 4$ unidades de peso y $N = 6$ objetos que podemos cargar en ella, con pesos $w = [1, 3, 1, 2, 4, 3]$ y valores $v = [8, 25, 12, 19, 32, 23]$, se pide:

a) Dibuja, aprovechando la siguiente cuadrícula, el *grafo de dependencias extendido* asociado a dicha instancia del problema (utiliza las cajas que necesites, tachando el resto, y traza los arcos entre estados correspondientes al grafo para el problema propuesto):

(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)

b) Haz una traza del algoritmo `iterative_knapsack_profit` (pág 7-20). Para ello, escribe arriba de cada uno de los estados del grafo del apartado anterior el valor que el algoritmo almacena en el diccionario `V`.

c) ¿Cuál es el valor que devuelve el algoritmo indicado? ¿Qué resultado devolvería el algoritmo que calcula la solución óptima del problema `knapsap` (pág 7-22) (para este no es preciso que realices la traza)? ¿Qué significa este último resultado?

Problema de la mochila discreta **Modelo 2**

Dada una mochila con capacidad para cargar $W = 4$ unidades de peso y $N = 6$ objetos que podemos cargar en ella, con pesos $w = [1, 3, 2, 1, 4, 3]$ y valores $v = [8, 25, 16, 12, 32, 23]$, se pide:

a) Dibuja, aprovechando la siguiente cuadrícula, el *grafo de dependencias extendido* asociado a dicha instancia del problema (utiliza las cajas que necesites, tachando el resto, y traza los arcos entre estados correspondientes al grafo para el problema propuesto):

(0,6)	(1,6)	(2,6)	(3,6)	(4,6)	(5,6)	(6,6)
(0,5)	(1,5)	(2,5)	(3,5)	(4,5)	(5,5)	(6,5)
(0,4)	(1,4)	(2,4)	(3,4)	(4,4)	(5,4)	(6,4)
(0,3)	(1,3)	(2,3)	(3,3)	(4,3)	(5,3)	(6,3)
(0,2)	(1,2)	(2,2)	(3,2)	(4,2)	(5,2)	(6,2)
(0,1)	(1,1)	(2,1)	(3,1)	(4,1)	(5,1)	(6,1)
(0,0)	(1,0)	(2,0)	(3,0)	(4,0)	(5,0)	(6,0)

b) Haz una traza del algoritmo `iterative_knapsack_profit` (pág 7-20). Para ello, escribe arriba de cada uno de los estados del grafo del apartado anterior el valor que el algoritmo almacena en el diccionario `V`.

c) ¿Cuál es el valor que devuelve el algoritmo indicado? ¿Qué resultado devolvería el algoritmo que calcula la solución óptima del problema `knapsap` (pág 7-22) (para este no es preciso que realices la traza)? ¿Qué significa este último resultado?

Programación dinámica
4 de mayo de 2006 (Mañana)

Prueba colectiva

Grupo:
Asistentes:

Camino más corto formado por k aristas en un grafo

La siguiente ecuación recursiva (pág 7-37) permite plantear la resolución del problema del camino más corto formado por k aristas en un grafo que empieza en un vértice s y acaba en un vértice t:

$$D(v,k) = \begin{cases} 0, & \text{si } v = s \text{ y } k = 0; \\ +\infty, & \text{si } v \neq s \text{ y } k = 0; \\ +\infty, & \text{si } \nexists (u,v) \in E \text{ y } k > 0; \\ \min\limits_{(u,v)\in E} \left(D(u,k-1) + d(u,v)\right), & \text{en otro caso.} \end{cases}$$

Se pide:

a) Escribid el algoritmo recursivo con memorización asociado a la anterior ecuación.

b) Escribid el algoritmo iterativo *que permite obtener la secuencia de vértices que forman el camino de distancia mínima* con k aristas entre s y t, indicando su coste temporal y espacial.

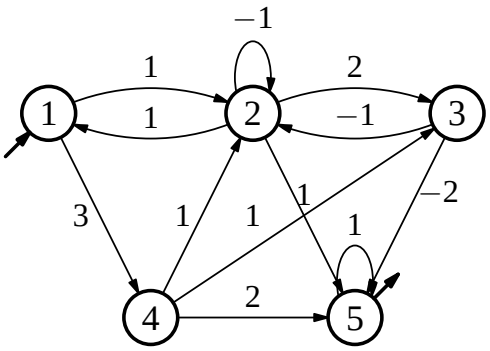
Programación dinámica
4 de mayo de 2006 (Tarde)

Prueba individual

Nombre: **Grupo:**

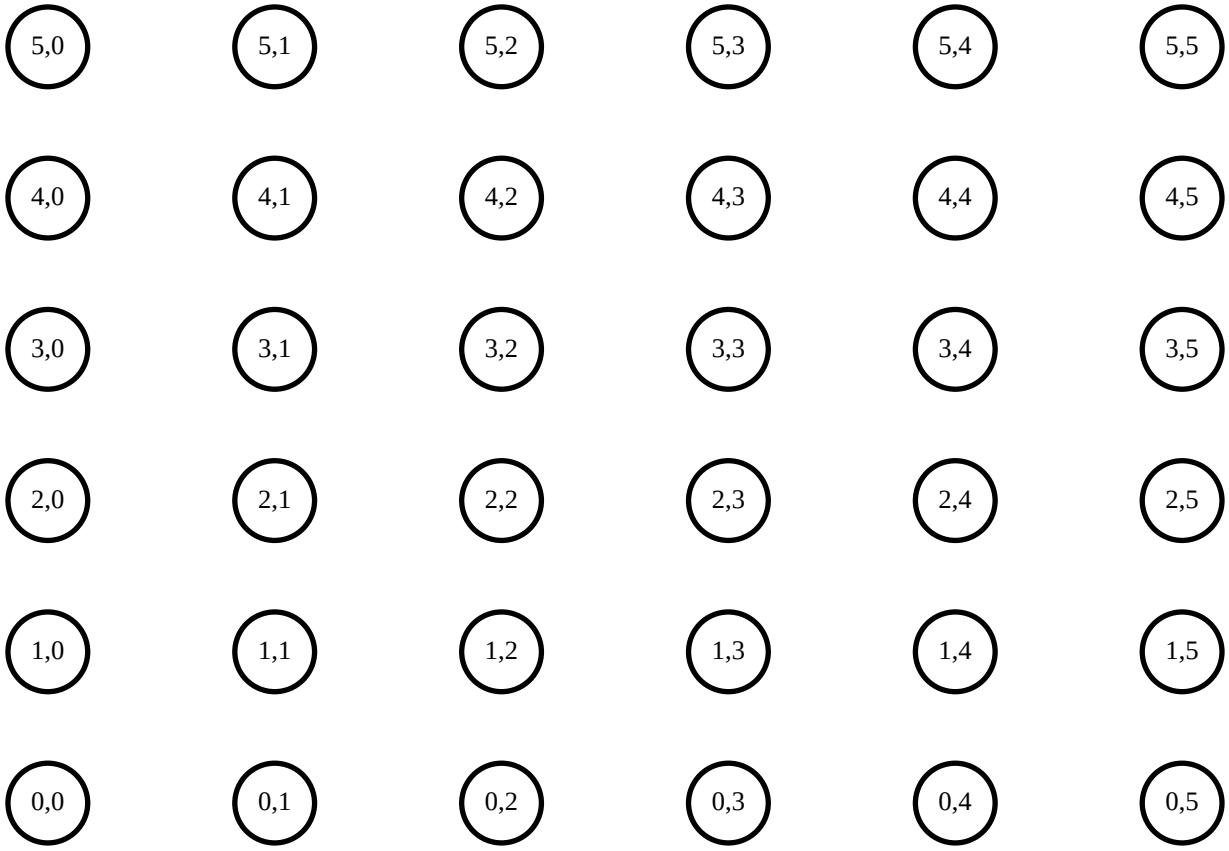
Problema del camino más corto formado por k aristas en un grafo **Modelo 1**

Debes encontrar cuál es el camino más corto que empieza en el vértice $s = 1$ y acaba en el vértice $t = 5$ formado por exactamente $k = 3$ aristas en el siguiente grafo:



Se pide:

a) Dibuja, aprovechando la siguiente figura, el *grafo multietapa de dependencias* asociado al grafo anterior y a la ecuación recursiva (7.8) (pág. 7-37) que resolvía el problema. (Utiliza los nodos que necesites, tachando el resto, y traza los arcos entre estados correspondientes al problema propuesto):

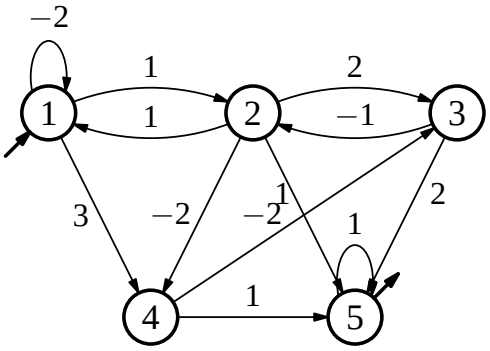


b) Haz una traza del algoritmo `k_edges_shortest_distance` (pág 7-38). Para ello, escribe arriba de cada uno de los estados del grafo del apartado anterior el valor que el algoritmo almacena en el diccionario `D`.

c) ¿Cuál es el valor que devuelve el algoritmo indicado? ¿Cuál sería la secuencia de aristas del grafo original que devolvería un algoritmo que calculara la solución óptima del problema (si hay más de una secuencia óptima, indica todas las posibles)?

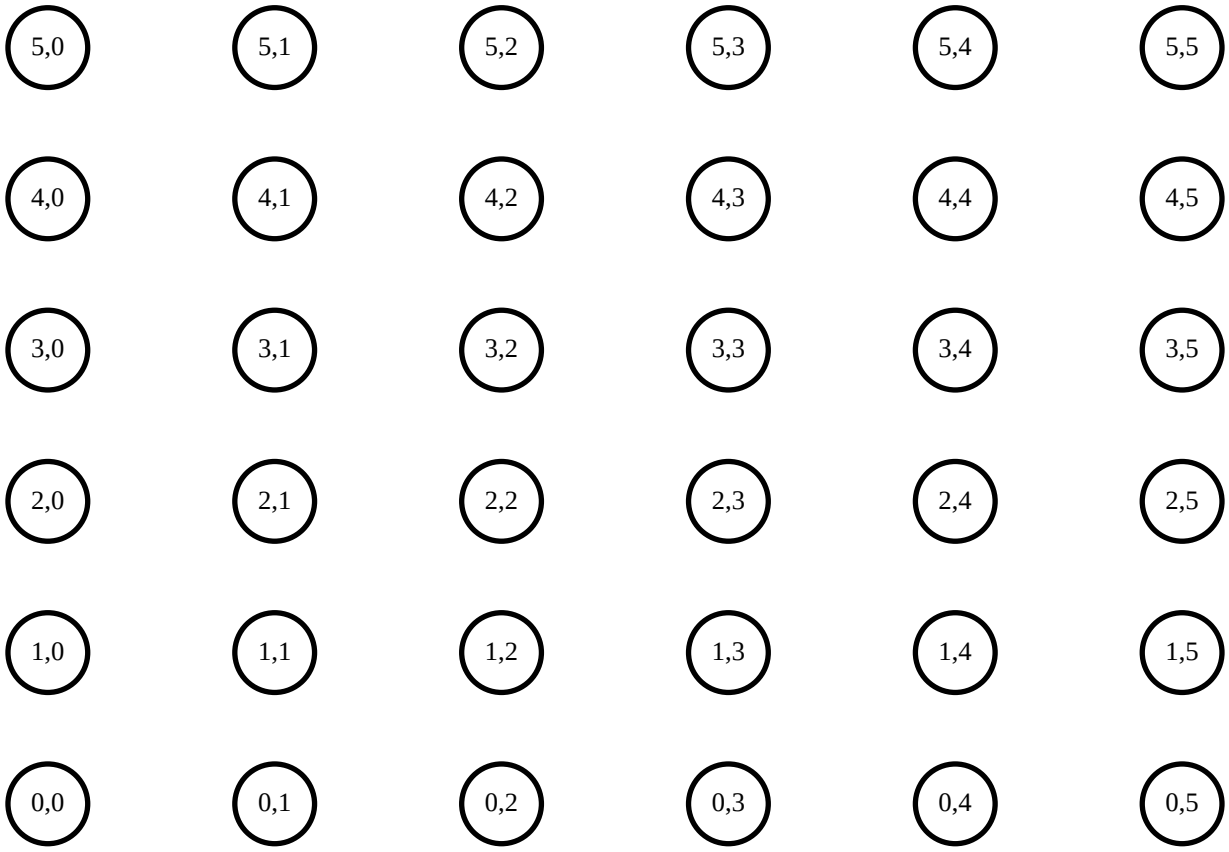
Problema del camino más corto formado por k aristas en un grafo **Modelo 2**

Debes encontrar cuál es el camino más corto que empieza en el vértice $s = 1$ y acaba en el vértice $t = 5$ formado por exactamente $k = 3$ aristas en el siguiente grafo:



Se pide:

a) Dibuja, aprovechando la siguiente figura, el *grafo multietapa de dependencias* asociado al grafo anterior y a la ecuación recursiva (7.8) (pág. 7-37) que resolvía el problema. (Utiliza los nodos que necesites, tachando el resto, y traza los arcos entre estados correspondientes al problema propuesto):



b) Haz una traza del algoritmo `k_edges_shortest_distance` (pág 7-38). Para ello, escribe arriba de cada uno de los estados del grafo del apartado anterior el valor que el algoritmo almacena en el diccionario `D`.

c) ¿Cuál es el valor que devuelve el algoritmo indicado? ¿Cuál sería la secuencia de aristas del grafo original que devolvería un algoritmo que calculara la solución óptima del problema (si hay más de una secuencia óptima, indica todas las posibles)?

Programación dinámica
4 de mayo de 2006 (Tarde)

Prueba colectiva

Grupo:
Asistentes:

Trayecto más barato en el río Congo formado por N embarcaderos

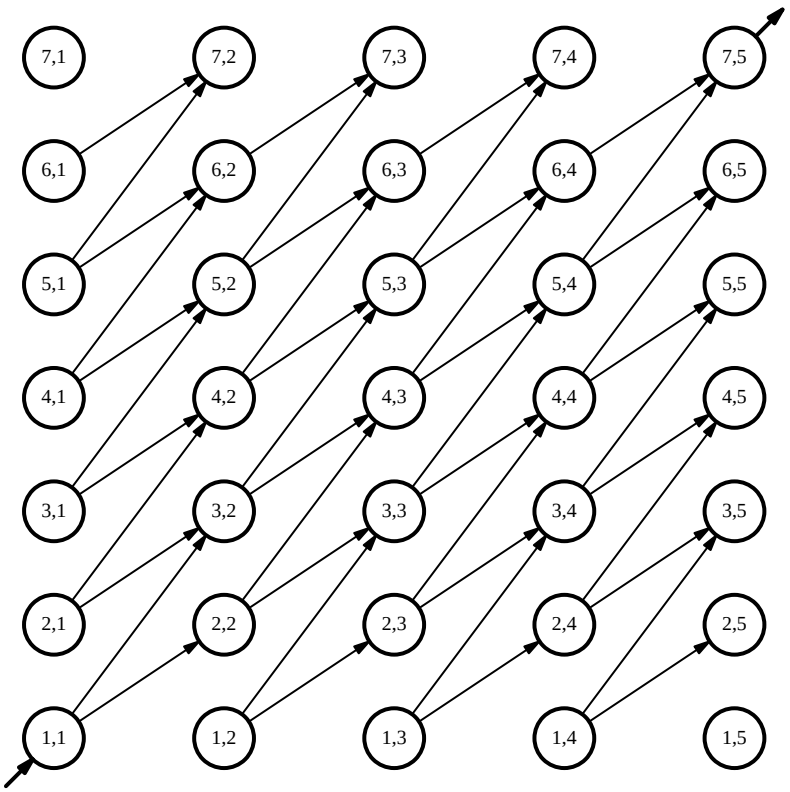
La siguiente ecuación recursiva permite plantear la resolución del problema del trayecto más barato en el río Congo que empieza en el embarcadero 1 y acaba en el embarcadero E y que está formado por *exactamente* N *embarcaderos*:

$$C(i,l) = \begin{cases} 0, & \text{si } i = 1 \text{ y } l = 1; \\ +\infty, & \text{si } i = 1 \text{ y } l \neq 1; \\ +\infty, & \text{si } i \neq 1 \text{ y } l = 1; \\ C(1,l-1) + c(1,2), & \text{si } i = 2 \text{ y } l > 1; \\ \min(C(i-2,l-1) + c(i-2,i), C(i-1,l-1) + c(i-1,i)), & \text{si } i > 2 \text{ y } l > 1, \end{cases}$$

donde el coste del trayecto con que un turista viaja del primer al último embarcadero pasando por exactamente N embarcaderos de la forma más económica es $C(E,N)$. Se pide:

a) Escribid el algoritmo recursivo con memorización asociado a la anterior ecuación.

b) Teniendo en cuenta que el grafo de dependencias entre estados asociado a la anterior ecuación recursiva es el siguiente (para un ejemplo con $E = 7$ embarcaderos y $N = 5$ escalas):



Escribid el algoritmo iterativo *con reducción de la complejidad espacial* que permite calcular el valor del trayecto más barato entre el primer y el último embarcadero que pasa por exactamente N embarcaderos. Indicad cuál es su coste temporal y espacial.

Programación dinámica (II)
15 de mayo de 2006 (Mañana, modelo 1)

Prueba individual

Nombre: **Grupo:**

Análisis de un problema **35 puntos**

Dados los números naturales N y K, una descomposición de N en K sumandos positivos es una secuencia n_1, n_2, $\ldots$, n_K, tal que

$$N = \sum_{1 \leq k \leq K} n_k \,.$$

Deseamos obtener la descomposición de N en K sumandos de modo que sea máximo su producto

$$\prod_{1 \leq k \leq K} n_k.$$

Por ejemplo, si $N = 6$ y $K = 3$ tenemos las siguientes descomposiciones posibles: $1 + 1 + 4$, $1 + 2 + 3$ y $2 + 2 + 2$. El producto de los sumandos es, respectivamente, $1 \cdot 1 \cdot 4 = 4$, $1 \cdot 2 \cdot 3 = 6$ y $2 \cdot 2 \cdot 2 = 8$. La descomposición óptima es, pues, $2 + 2 + 2$.

La siguiente ecuación recursiva calcula el producto de los sumandos asociados a la descomposición óptima en el problema anterior:

$$P(n,k) = \begin{cases} n, & \text{si } k = 1; \\ 0, & \text{si } k > 1 \text{ y } n = 1; \\ \max\limits_{1 \leq m \leq n-1} P(n-m, k-1) \cdot m, & \text{si } k > 1 \text{ y } n > 1, \end{cases}$$

donde la llamada $P(N, K)$ proporciona dicho valor. Se pide:

a) Representa el grafo de dependencias para la llamada $P(7, 3)$. Indica qué representa un estado (n, k) y un arco entre dos estados (n', k') y (n, k) dentro del grafo.

b) Indica cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de la ecuación presentada que devolviera *la descomposición óptima*. Justifica brevemente tu respuesta.

c) ¿Crees que sería posible reducir el coste espacial indicado en el apartado anterior si tuviéramos que diseñar un algoritmo iterativo que sólo encontrara *el valor* del producto de la descomposición óptima? Si es así, indica cuál sería dicho coste y justifícalo.

Programación dinámica (II)
15 de mayo de 2006 (Mañana)

Prueba colectiva

Grupo:
Asistentes:
P_2: Peso de este control en la nota de programación dinámica (mín 50 %, máx 80 %):..... %
P_1: Peso del control previo en la nota de programación dinámica (mín 20 %, máx 50 %):..... %
($P_2 + P_1 = 100\,\%$. Si se deja en blanco o no suma 100 %, el peso será del 50 % en cada control.)

Resolución de problemas **65 puntos**

Sea el anterior problema:

****

Dados los números naturales N y K, una descomposición de N en K sumandos positivos es una secuencia n_1, n_2, ..., n_K, tal que

$$N = \sum_{1 \le k \le K} n_k \,.$$

Deseamos obtener la descomposición de N en K sumandos de modo que sea máximo su producto

$$\prod_{1 \le k \le K} n_k.$$

Por ejemplo, si $N = 6$ y $K = 3$ tenemos las siguientes descomposiciones posibles: $1 + 1 + 4$, $1 + 2 + 3$ y $2 + 2 + 2$. El producto de los sumandos es, respectivamente, $1 \cdot 1 \cdot 4 = 4$, $1 \cdot 2 \cdot 3 = 6$ y $2 \cdot 2 \cdot 2 = 8$. La descomposición óptima es, pues, $2 + 2 + 2$.

La siguiente ecuación recursiva calcula el producto de los sumandos asociados a la descomposición óptima en el problema anterior:

$$P(n,k) = \begin{cases} n, & \text{si } k = 1; \\ 0, & \text{si } k > 1 \text{ y } n = 1; \\ \max\limits_{1 \le m \le n-1} P(n-m, k-1) \cdot m, & \text{si } k > 1 \text{ y } n > 1, \end{cases}$$

donde la llamada $P(N, K)$ proporciona dicho valor.

****

Se proponen las dos siguientes modificaciones en el enunciado del problema de la maximización del producto de los sumandos resultantes de la descomposición de un número N:

1. La descomposición no tiene por qué tener un número fijo K de componentes.
2. Ningún sumando puede ser mayor que un valor dado $M < N$.

El resto de condiciones se mantienen.
Con las nuevas condiciones y el ejemplo $N = 10$ y $M = 4$, las siguientes descomposiciones serían válidas: 4+2+1+3, $3 + 3 + 4$ y $1 + 2 + 1 + 1 + 2 + 2 + 1$, mientras que la descomposición $2 + 3 + 5$ no lo sería.

Se pide, *para este nuevo problema*:

a) Formalizad el problema en términos de optimización.

b) Plantead la ecuación recursiva de programación dinámica que calcule el producto de los sumandos asociados a la descomposición óptima, indicando cuál debe ser la llamada que se efectue a dicha función para obtener el valor óptimo.

c) Representad el grafo de dependencias entre las llamadas de la ecuación recursiva para $N = 6$ y $M = 3$.

d) Indicad cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de vuestra ecuación recursiva que devolviera *la descomposición óptima*. Justificad brevemente la respuesta.

e) ¿Sería posible reducir el coste espacial indicado en el apartado anterior si hubiera que diseñar un algoritmo iterativo que sólo encontrara *el valor* del producto de la descomposición óptima? Si es así, indicad cuál sería dicho coste y justificadlo.

Programación dinámica (II)

15 de mayo de 2006 (Mañana)

Prueba colectiva opcional

Grupo:
Asistentes:

Recuperación del primer control de programación dinámica — +20 puntos (a añadir a P_1)

Diseñad un algoritmo iterativo que devuelva la descomposición óptima en K sumandos de un número N *a partir del enunciado original del problema*:

****

Dados los números naturales N y K, una descomposición de N en K sumandos positivos es una secuencia n_1, n_2, ..., n_K, tal que

$$N = \sum_{1 \leq k \leq K} n_k \, .$$

Deseamos obtener la descomposición de N en K sumandos de modo que sea máximo su producto

$$\prod_{1 \leq k \leq K} n_k .$$

Por ejemplo, si $N = 6$ y $K = 3$ tenemos las siguientes descomposiciones posibles: $1 + 1 + 4$, $1 + 2 + 3$ y $2 + 2 + 2$. El producto de los sumandos es, respectivamente, $1 \cdot 1 \cdot 4 = 4$, $1 \cdot 2 \cdot 3 = 6$ y $2 \cdot 2 \cdot 2 = 8$. La descomposición óptima es, pues, $2 + 2 + 2$.

La siguiente ecuación recursiva calcula el producto de los sumandos asociados a la descomposición óptima en el problema anterior:

$$P(n,k) = \begin{cases} n, & \text{si } k = 1; \\ 0, & \text{si } k > 1 \text{ y } n = 1; \\ \max\limits_{1 \leq m \leq n-1} P(n-m, k-1) \cdot m, & \text{si } k > 1 \text{ y } n > 1, \end{cases}$$

donde la llamada $P(N, K)$ proporciona dicho valor.

****

Programación dinámica (II)
15 de mayo de 2006 (Tarde, modelo 1)

Prueba individual

Nombre: **Grupo:**

Análisis de un problema **35 puntos**

A lo largo del río Amazonas hay N poblados en los que podemos hacer escala. El poblado i está separado del poblado j por una distancia en kilómetros que podemos conocer consultando el valor $d(i, j)$. Sabemos además que es imposible nadar más de K kilómetros sin parar a descansar en un poblado (suponemos $d(i-1, i) \leq K$, para $1 < i \leq N$). Por cortesía, cada vez que llegamos a un poblado i debemos descansar en él tantos días como nos indica $c(i)$, la costumbre local (donde $2 \leq i \leq N-1$, es decir, no se cuentan los días de estancia en el primer ni en el último poblado).

Se pretende averiguar cual es el mínimo número de días necesario para poder realizar la travesía a nado desde el poblado 1 hasta el N.

Una ecuación recursiva que permite resolver el problema es la siguiente:

$$V(j) = \begin{cases} 0, & \text{si } j = 1; \\ \min\limits_{\substack{1 \leq i < j: \\ d(i,j) \leq K}} V(i) + c(i), & \text{si } j > 1, \end{cases}$$

donde la llamada $V(N)$ proporciona el número mínimo de días. Se pide:

a) Representa el grafo de dependencias para la llamada $V(7)$ con $K = 10$ y las distancias:

$$d = \{(1,2):5, (1,3):8, (1,4):12, (1,5):15, (1,6):17, (1,7):23, (2,3):3, (2,4):7, (2,5):10, (2,6):12,$$
$$(2,7):18, (3,4):4, (3,5):7, (3,6):9; (3,7):16, (4,5):3, (4,6):5, (4,7):11, (5,6):2, (5,7):8, (6,7):6\}$$

Indica qué representa un estado j y un arco entre dos estados i y j dentro del grafo.

b) Indica cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de la ecuación presentada que devolviera *la secuencia de embarcaderos óptima*. Justifica brevemente tu respuesta.

c) ¿Crees que sería posible reducir el coste espacial indicado en el apartado anterior si tuviéramos que diseñar un algoritmo iterativo que sólo encontrara *el valor* del mínimo número de días necesarios para recorrer el río? Si es así, indica cuál sería dicho coste y justifícalo.

Programación dinámica (II)
15 de mayo de 2006 (Tarde)

Prueba colectiva

Grupo:
Asistentes:
P_2: Peso de este control en la nota de programación dinámica (mín 50 %, máx 80 %):..... %
P_1: Peso del control previo en la nota de programación dinámica (mín 20 %, máx 50 %):..... %
($P_2 + P_1 = 100\,\%$. Si se deja en blanco o no suma 100 %, el peso será del 50 % en cada control.)

Resolución de problemas **65 puntos**

Sea el anterior problema:

****

A lo largo del río Amazonas hay N poblados en los que podemos hacer escala. El poblado i está separado del poblado j por una distancia en kilómetros que podemos conocer consultando el valor $d(i, j)$. Sabemos además que es imposible nadar más de K kilómetros sin parar a descansar en un poblado (suponemos $d(i-1, i) \leq K$, para $1 < i \leq N$). Por cortesía, cada vez que llegamos a un poblado i debemos descansar en él tantos días como nos indica $c(i)$, la costumbre local (donde $2 \leq i \leq N-1$, es decir, no se cuentan los días de estancia en el primer ni en el último poblado).

Se pretende averiguar cual es el mínimo número de días necesario para poder realizar la travesía a nado desde el poblado 1 hasta el N.

Una ecuación recursiva que permite resolver el problema es la siguiente:

$$V(j) = \begin{cases} 0, & \text{si } j = 1; \\ \min\limits_{\substack{1 \leq i < j: \\ d(i,j) \leq K}} V(i) + c(i), & \text{si } j > 1, \end{cases}$$

donde la llamada $V(N)$ proporciona el número mínimo de días.

****

Se proponen las tres siguientes modificaciones en el enunciado del problema del cálculo de la travesía a nado en menos días a través del río Amazonas:

1. Hay que realizar exactamente M escalas (incluyendo el poblado de salida y el de llegada).
2. Como reto personal, el nadador se ha propuesto que, una vez sale de un poblado, no puede parar en el inmediatamente posterior.
3. No conocemos las distancias $d(i, j)$ entre poblados. Sin embargo, para cada poblado $1 < j \leq N$, sí que nos indican $p(j)$, que es el poblado más lejano desde el que podemos llegar nadando sin parar hasta j. Por ejemplo, $p(7) = 3$ significa que al poblado 7 podemos llegar directamente como muy lejos desde el poblado 3, (también, por tanto, desde todos los que hay entre el 3 y el 7, como el 4, el 5 o el 6) pero no es posible llegar al 7 desde el 1 o el 2 sin hacer ninguna escala.

Con las nuevas condiciones y el ejemplo $N = 8$, $M = 4$ y $p = (-, 1, 1, 1, 2, 4, 4, 4)$, las siguientes travesías serían válidas: $(1, 3, 5, 8)$ y $(1, 4, 6, 8)$. Sin embargo, no serían válidas: $(1, 4, 8)$, por la restricción 1 (hay 3 escalas, no 4); $(1, 2, 5, 8)$, por la restricción 2 (no se puede parar en el 2 si el anterior es el 1); ni $(1, 3, 6, 8)$, por la restricción 3 ($p(6) = 4$, por lo que no se puede ir del 3 al 6).

Se pide, *para este nuevo problema*:

a) Formalizad el problema en términos de optimización.

b) Plantead la ecuación recursiva de programación dinámica que calcula el número de días mínimo para realizar la travesía óptima, indicando cuál debe ser la llamada que se efectue a dicha función para obtener el valor óptimo.

c) Representad el grafo de dependencias entre las llamadas de la ecuación recursiva para $N = 6$, $M = 3$ y $p = (-, 1, 1, 1, 2, 4)$.

d) Indicad cuál sería el coste temporal y espacial de un algoritmo iterativo diseñado a partir de vuestra ecuación recursiva que devolviera *la travesía óptima*. Justificad brevemente la respuesta.

e) ¿Sería posible reducir el coste espacial indicado en el apartado anterior si hubiera que diseñar un algoritmo iterativo que sólo encontrara *el valor* del mínimo número de días necesarios para recorrer el río? Si es así, indicad cuál sería dicho coste y justificadlo.

Programación dinámica (II)

15 de mayo de 2006 (Tarde)

Prueba colectiva opcional

Grupo:
Asistentes:

Recuperación del primer control de programación dinámica — +20 puntos (a añadir a P_1)

Diseñad un algoritmo iterativo que devuelva la travesía óptima en el río Amazonas *según la definición del enunciado original del problema*:

****

A lo largo del río Amazonas hay N poblados en los que podemos hacer escala. El poblado i está separado del poblado j por una distancia en kilómetros que podemos conocer consultando el valor $d(i, j)$. Sabemos además que es imposible nadar más de K kilómetros sin parar a descansar en un poblado (suponemos $d(i-1, i) \leq K$, para $1 < i \leq N$). Por cortesía, cada vez que llegamos a un poblado i debemos descansar en él tantos días como nos indica $c(i)$, la costumbre local (donde $2 \leq i \leq N-1$, es decir, no se cuentan los días de estancia en el primer ni en el último poblado).

Se pretende averiguar cual es el mínimo número de días necesario para poder realizar la travesía a nado desde el poblado 1 hasta el N.

Una ecuación recursiva que permite resolver el problema es la siguiente:

$$V(j) = \begin{cases} 0, & \text{si } j = 1; \\ \min\limits_{\substack{1 \leq i < j: \\ d(i,j) \leq K}} V(i) + c(i), & \text{si } j > 1, \end{cases}$$

donde la llamada $V(N)$ proporciona el número mínimo de días.

****

Primer control parcial de Esquemas Algorítmicos (IG24)

8 de abril de 2006

1. Algoritmos voraces — 5 puntos

Un nadador profesional se prepara para acudir a las olimpiadas. El nadador es capaz de nadar con solvencia en n pruebas. Para cada prueba tiene la estimación del esfuerzo que necesita realizar para quedar en tercer lugar: $e = (e_1, e_2, \ldots, e_n) \in \mathbb{N}^n$. Quedar en segundo lugar en cada prueba le costará el doble del esfuerzo que hacerlo en tercer lugar y para ganar la prueba deberá realizar el triple de esfuerzo que para quedar tercero. El premio que recibirá por cada prueba ganada es de 100.000 euros, 40.000 por cada segundo puesto y 10.000 por cada tercera posición.

El nadador sabe las fuerzas totales de las que va a disponer a lo largo de la competición, $F \in \mathbb{N}$, y quiere repartirlas entre las distintas pruebas para ganar la mayor cantidad posible de dinero. Se pide:

a) Formaliza el conjunto de soluciones factibles e indica, a partir de él, cómo se calcularía el valor de la función objetivo sobre una solución factible (esto no es preciso formalizarlo).

b) Si el nadador prepara 5 pruebas y $e = [40, 100, 30, 50, 20]$ y $F = 200$, indica, utilizando la notación definida en el apartado anterior, cuál sería una solución óptima sobre el ejemplo propuesto y su valor.

c) Propón una estrategia voraz lo más eficiente posible que devuelva la solución óptima para el problema ante cualquier entrada válida.

d) Diseña un algoritmo que siga dicha estrategia. El algoritmo deberá devolver, para cada prueba, el *esfuerzo* que debe realizar el nadador para alcanzar la solución óptima.

e) Indica (y justifica) cuál es el coste temporal de la estrategia (y del algoritmo) propuestos.

f) Supón que los premios por quedar en primer, segundo y tercer lugar fueran un dato de entrada al algoritmo: $p = (p_1, p_2, p_3) \in \mathbb{N}^3$, donde $p_1 > p_2 > p_3$ (por ejemplo, $p =$ (100.000, 75.000, 50.000)). ¿Seguiría siendo válida la estrategia que has diseñado? Justifica la respuesta. Si no es válida, ¿podrías modificarla o diseñar otra que garantizará encontrar la solución óptima para cualquier instancia del problema? Si la respuesta es positiva, indica cómo, y si es negativa, justifícala.

2. Divide y vencerás — 5 puntos

Disponemos de un vector v que contiene números enteros. Los elementos del vector se encuentran divididos en dos grupos de valores crecientes consecutivos, el primero de los grupos formado por números negativos y el segundo por números positivos. Sin embargo, no disponemos de información acerca de cuáles pueden ser esos valores ni la cantidad de valores que contiene cada grupo (aunque, al menos, de cada grupo hay un valor).

El vector $v = [-12, -11, -10. - 9, -8, -7, 9, 10, 11, 12, 13, 14, 15, 16]$ cumple dichas propiedades, donde el primer grupo empieza en la posición 0 y acaba en la 5 (del -12 al -7), y el segundo grupo empieza en la 6 y acaba en la 13 (del 9 al 16).

Queremos conocer la suma de los valores del primer grupo y la de los valores del segundo grupo, sabiendo que en un rango consecutivo de n valores, en el que v_i es el primer elemento del rango y v_j el último elemento, se cumple que la suma es $(v_i + v_j) * n/2$, donde $n = j - i + 1$. En el ejemplo, la suma del primer grupo es $(-12 + -7) * (5 - 0 + 1)/2 = -19 * 6/2 = -57$, mientras que la suma del segundo grupo es $(9 + 16) * (13 - 6 + 1)/2 = 100$. Se pide:

a) Diseña un algoritmo recursivo mediante la técnica de divide y vencerás que devuelva como resultado el par de valores correspondientes a la suma del primer grupo y a la suma del segundo grupo (para el ejemplo, debería devolver $(-57, 100)$). El algoritmo debe ser lo más eficiente posible tanto desde el punto de vista de la complejidad temporal como de la espacial.

   Una pista para obtener la versión más eficiente: *la función recursiva* que desarrolles no tiene porque calcular directamente las sumas de los grupos, sino que puede devolver algún resultado que permita, una vez acabado el cálculo recursivo y *mediante otra función*, obtener los valores pedidos. En cualquier caso, la estrategia principal de resolución deberá ser la de divide y vencerás y todas las funciones necesarias para el cálculo solicitado deberás incluirlas.

b) Indica si el algoritmo presenta un comportamiento uniforme o existe un mejor y un peor caso. Si el comportamiento es uniforme, trata de modificarlo para que ante determinadas instancias pueda finalizar antes su ejecución. Señala en qué condiciones (ante que tipos de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor.

c) Realiza un análisis de la complejidad temporal y espacial del algoritmo presentado, justificando los costes en el mejor y en el peor de los casos.

d) Si el algoritmo propuesto presenta recursividad por cola, elimínala e indica si alguno de los costes calculados en el apartado anterior varía.

e) ¿Se te ocurre algún algoritmo iterativo sencillo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás (no es necesario que lo implementes, tan sólo descríbelo)? Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el que has diseñado.

Segundo control parcial de Esquemas Algorítmicos (IG24)

27 de mayo de 2006

1. Programación dinámica — 4 puntos

En la preparación de una maratón por etapas se ha diseñado ya el trayecto concreto por el que han de correr los atletas, pero falta decidir dónde empieza y dónde acaba cada una de las E etapas de la maratón. El trayecto completo parte del kilómetro 0 y finaliza en el kilómetro K. Las etapas sólo pueden empezar o acabar en puntos kilómetricos enteros (por ejemplo, en el kilómetro 5, pero no en el 5.3).

El esfuerzo realizado por un corredor en una etapa depende de sus kilómetros de inicio y finalización, así como del número de etapas que ya ha cubierto. Con $f(i, j, e)$ cuantificamos el esfuerzo que supone recorrer los kilómetros del i al j en la etapa e. Hay una limitación en el diseño de las etapas: una etapa no puede medir menos de k_1 kilómetros ni más de k_2.

Debes averiguar, empleando la técnica de programación dinámica, dónde debe empezar y acabar cada una de las E etapas para garantizar que el recorrido suponga el *mayor* esfuerzo posible a los corredores.

1. Con las condiciones anteriormente expuestas:

   a) Formaliza el problema en términos de optimización (conjunto de soluciones factibles y función objetivo).

   b) Plantea la ecuación recursiva de programación dinámica que calcula el esfuerzo que requiere la maratón más costosa de realizar, indicando cuál debe ser la llamada que se efectue a dicha función para obtener el valor óptimo.

   c) Representa el grafo de dependencias entre las llamadas del algoritmo recursivo para el siguiente ejemplo de mentirijillas: $E = 3$ etapas, $K = 8$, $k_1 = 2$ y $k_2 = 5$. *Razona* los costes espacial y temporal con los que se puede efectuar el cálculo sobre dicho grafo (indica explícitamente, justificando tu respuesta, si es posible efectuar una reducción de complejidad espacial al realizar el cálculo iterativo).

   d) Si sabemos que la diferencia entre el máximo y el mínimo número de kilometros que es posible realizar en una etapa es $D = k_2 - k_1$, y que $D < K$, ¿podrías expresar de forma más ajustada el coste temporal del algoritmo? Justifica tu respuesta.

   e) ¿Podría haber instancias del problema para las que no existiera una solución factible? Razona la respuesta y, si es así, indica las condiciones que se deberían cumplir para que pueda haber una solución factible y pon algún ejemplo de instancia que no tenga solución factible.

2. El año siguiente, la organización de la carrera decide cambiar parte de los criterios para el diseño de la carrera: dada la duración total de la carrera K y la duración mínima y máxima de cada etapa, k_1 y k_2, desea realizar el mejor diseño en etapas, pero *sin fijar previamente el número total de etapas de la carrera.*

   ¿Cómo modificarías el conjunto de soluciones factibles para que recogiera este nuevo planteamiento? ¿Cómo resolverías ahora el problema? ¿Con qué coste temporal y espacial?

2. Programación dinámica — 2.75 puntos

Un vigilante nocturno de un polígono industrial tiene que fichar cada noche R veces en distintos puntos del polígono para demostrar que se dedica a su trabajo. En total hay N naves industriales (por simplicidad, asumiremos que están numeradas entre la 1 y la N) y en cada una de ellas hay un punto de fichaje.

El vigilante debe comenzar y acabar la ronda en un mismo punto de control, situado en la nave principal, cuyo número es el p. Para asegurarse de que no se instala a pasar la noche en una nave, no esta permitido que el vigilante fiche dos veces seguidas en un mismo puesto, aunque que no hay inconveniente en que el vigilante fiche más de una vez en un mismo punto durante la noche siempre que respete la regla anterior.

El vigilante, que sabe la distancia que hay entre cada par de naves ($d(n', n)$, donde $1 \leq n' \leq N, 1 \leq n \leq N, n' \neq n$), desea que le indiques en que naves debe fichar para recorrer la menor distancia a lo largo de una jornada. Como ayuda, la siguiente ecuación recursiva de programación dinámica permite resolver el problema de obtener el valor de la mínima distancia:

$$D(r, n) = \begin{cases} 0, & \text{si } r = 1 \text{ y } n = p; \\ +\infty, & \text{si } r = 1 \text{ y } n \neq p; \\ \min\limits_{\substack{1 \leq n' \leq N: \\ n' \neq n}} D(r-1, n') + d(n', n), & \text{si } r > 1. \end{cases}$$

La llamada $D(R, p)$ proporciona dicho valor.

Se pide:

a) Representa el grafo de dependencias para el problema en el que $R = 4$, $N = 5$ y $p = 2$. Indica qué representa un estado (r, n) y un arco entre dos estados (r', n') y (r, n) dentro del grafo.

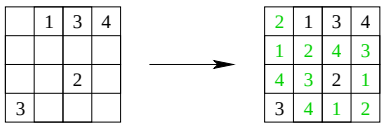
b) Diseña un algoritmo iterativo que devuelva la *secuencia de naves óptima* en las que debe hacer los fichajes para obtener la mínima distancia.

c) Indica cuál es el coste temporal y espacial del algoritmo diseñado justificando brevemente tu respuesta.

d) ¿Crees que sería posible reducir el coste espacial si únicamente nos interesará obtener el valor de la mínima distancia? Si es así, índica cuál sería dicho coste y justifícalo.

3. Búsqueda con retroceso — 3.25 puntos

El simpleku es una variante del juego del sudoku que consiste en ubicar en un tablero de n filas por n columnas los números del 1 al n de forma que en todas las filas del tablero y en todas las columnas aparezcan los n números (o, lo que es lo mismo, un mismo número aparecerá una vez en cada columna y una vez en cada fila). Para darle sentido al juego, se falicita un tablero en el que ya aparecen ubicados algunos de los números, teniendo que encargarse el jugador de rellenar las restantes casillas. Por ejemplo, para un problema con $n = 4$, la siguiente figura muestra una configuración inicial y una posible solución:



Queremos encontrar un algoritmo de *búsqueda con retroceso* que rellene un tablero de $n \times n$ a partir de una configuración inicial. El tablero con la configuración inicial se facilita en forma de matriz, estando rellenas con el valor 0 las casillas inicialmente vacias. También se proporciona una lista con las posiciones libres, en las que cada elemento de la lista es un par de valores con las coordenadas de la posición en el tablero. En el ejemplo anterior, los datos de entrada del problema serían:

```
tab = [[0,1,3,4],
       [0,0,0,0],
       [0,0,2,0],
       [3,0,0,0]]
libres = [(0,0),(1,0),(1,1),(1,2),(1,3),(2,0),(2,1),(2,3),(3,1),(3,2),(3,3)]
```

A continuación, se presenta el esquema de un posible algoritmo de *backtracking* que resuelve este problema:

```
def simpleku(n,tab,libres):
  def backtracking(pos):
    if is_complete(pos):
      if is_factible(pos): return tab
    else:
      i,j=libres[pos]
      for num in xrange(1, n+1):
        if is_promising(i,j,num):
          tab[i][j]=num
          found = backtracking(pos+1)
          if found != None: return found
      tab[i][j]=0
    return None
  return backtracking(0)
```

Se pide que:

a) Escribas el código correspondiente a la función `is_complete(pos)`, es decir, que definas cuál es el criterio que permite determinar si un estado es una *solución completa*.

b) Escribas el código correspondiente a la función `is_factible(pos)`, es decir, que definas cuál es el criterio que permite determinar si una *solución completa* es una *solución factible* o no.

c) Escribas el código correspondiente a la función `is_promising(i,j,num)`, es decir, que definas cuál es el criterio que permite determinar si un *estado parcial* puede conducir a una solución factible (en cuyo caso hay que ramificarlo) o no.

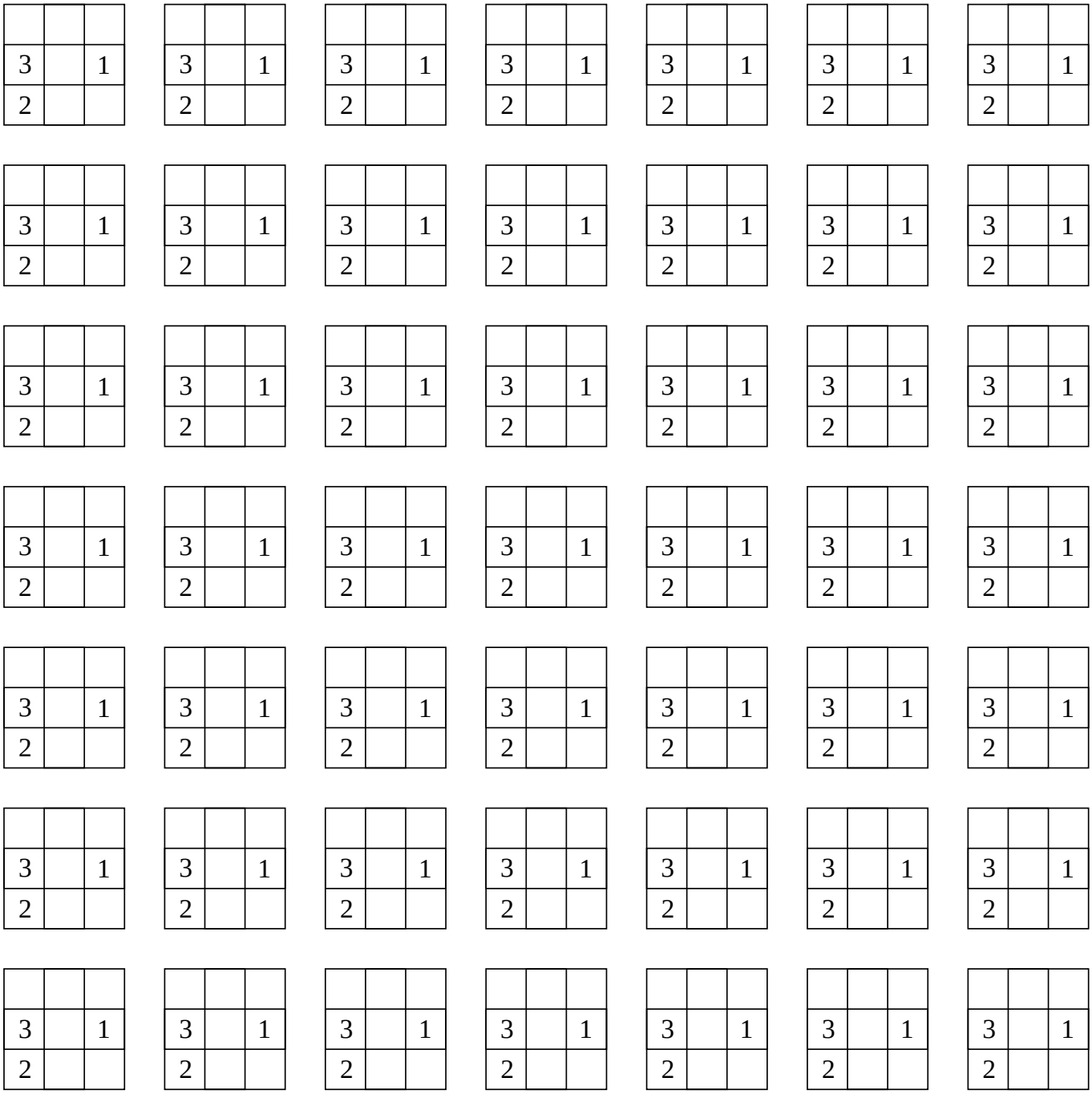
d) A la vista del código desarrollado en los apartados anteriores, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*? Justifícalas.

e) Realiza una traza de ejecución de este algoritmo para la llamada `simpleku(n,tab,libres)` donde $n = 3$ y

```
tab = [[0,0,0],
       [3,0,1],
       [2,0,0]]
libres = [(0,0),(0,1),(0,2),(1,1),(2,1),(2,2)]
```

Para ello, haz uso de las figuras que aparecen en la siguiente hoja (sólo de aquellas que necesites). Debes representar el árbol de llamadas efectuadas, incluyendo solamente aquellos estados que se generan durante el proceso de búsqueda. Marca de forma especial los estados que se estudian pero se descartan por no superar la condición de ser prometedores. Indica cuál es el resultado devuelto por el algoritmo.

f) Por último, modifica lo que consideres oportuno del algortimo para introducir una restricción adicional: todos los números deben aparecer también en las diagonales principales del tablero (es decir, un número no puede aparecer más de una vez en cada diagonal principal).


3 1
2

Examen convocatoria ordinaria de Esquemas Algorítmicos (IG24)

16 de junio de 2006

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo) :
Parte 1 (Preg. 1 y 2) ( ) Parte 2 (Preg. 3 y 4) ( ) Todo ( ) Renuncio a la evaluación alternativa ( )
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Voraces — 2,5 puntos

El algoritmo de Huffman permite obtener una codificación binaria prefija para los n símbolos de un alfabeto de los que se conoce su frecuencia de aparición. Para ello construye un árbol binario. Dicho árbol cumple la propiedad de que, de entre todos los posibles, es aquel que minimiza el número esperado de bits en la codificación de los símbolos del alfabeto. El coste temporal del algoritmo es $O(n \lg n)$.

Estudia cada uno de los cuatro supuestos que se plantean a continuación. Para cada caso, indica si es posible modificar el algoritmo de Huffman o plantear alguna estrategia alternativa que iguale o mejore el coste de este. Si es así, indica cómo (señalando las estructuras de datos necesarias para la resolución) y acompaña la explicación con el coste temporal de los algoritmos resultantes:

a) Los símbolos del alfabeto se nos proporcionan inicialmente ordenados de menor a mayor frecuencia de aparición.

b) Los símbolos del alfabeto presentan todos la misma frecuencia de aparición.

c) Los símbolos del alfabeto cumplen que hay uno de ellos que se presenta la mitad de las veces (su frecuencia es 1/2), otro se presenta un cuarto de las veces (1/4), otro 1/8, otro 1/16 y así sucesivamente. Los símbolos se nos facilitan desordenados.

d) El mismo caso que el anterior, pero los símbolos se nos proporcionan ordenados de mayor a menor frecuencia.

En el caso de los supuestos b) y c), ¿podrías justificar cuál es el número máximo de bits que se utilizarán para representar un símbolo? ¿y el número mínimo?

2. Divide y vencerás — 2,5 puntos

Dado un vector (sin ordenar) compuesto por n números enteros, X, y dos números a y m, se desea saber si el número a aparece más de m veces en el vector X.

a) Diseña un algoritmo recursivo mediante la técnica de divide y vencerás que permita resolver este problema (con una *complejidad espacial no superior* a $O(\log n)$). Recuerda que debes especificar la función que recibe los parámetros y realiza la llamada inicial que desencadena el procedimiento recursivo que efectúa los cálculos necesarios (no olvides indicar igualmente esta llamada).

   Pista: para facilitar la elaboración del algoritmo, puedes hacer que el procedimiento recursivo se dedique a contar el número de veces que aparece a en el (trozo de) vector correspondiente y que la función que recibe los parámetros y realiza la llamada inicial sea la que realmente devuelva el resultado `True` o `False` requerido.

b) Indica si el algoritmo que has diseñado en el apartado anterior presenta un comportamiento uniforme o existe un mejor y un peor caso. Si el comportamiento es uniforme, modifícalo para que tenga un mejor y un peor caso. Finalmente, señala en qué condiciones (ante qué tipo de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor.

c) Analiza la *complejidad temporal* del algoritmo resultante tras la realización del apartado anterior. Debes justificar adecuadamente los costes calculados.

d) ¿Se te ocurre algún algoritmo iterativo sencillo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás (no es necesario que lo implementes, tan sólo descríbelo)? Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el que has diseñado.

3. Programación dinámica — 3,5 puntos

Una cadena de televisión dedica una franja *consecutiva* de su horario de madrugada a la publicidad, con la emisión de publireportajes y anuncios. La duración de dicha franja es de M minutos. Para cubrirla, dispone de N anuncios y publis, numerados de 1 a N, de los que conoce su duración en *segundos*, $s_1, s_2, \ldots, s_N$. Asimismo, sabe cuál es el precio, $p_i, 1 \leq i \leq N$, que están dispuestas a pagar las compañías anunciantes por la emisión de sus anuncios.

La cadena desea averiguar cuál es el máximo beneficio que puede obtener con la emisión de cualquier combinación de anuncios. Para ello, la cadena asume que no tiene por qué emitir todos los anuncios disponibles y que en la franja de publicidad puede repetir la emisión de un anuncio tantas veces como le interese, teniendo, eso sí, que ocupar completamente dicha franja con los anuncios disponibles. Resuelve el problema de la forma más eficiente posible utilizando la técnica de programación dinámica, para lo que se te pide:

a) Formaliza el problema en términos de optimización,

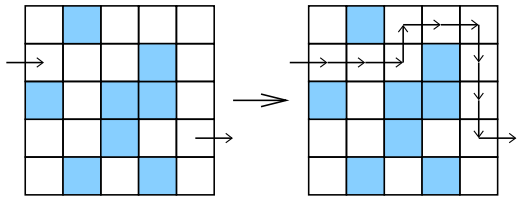
b) plantea la ecuación recursiva que calcula el máximo beneficio posible,

c) diseña un algoritmo iterativo que devuelva dicho valor, razonando su coste temporal y espacial (indica explícitamente, justificando tu respuesta, si es posible efectuar una reducción de complejidad espacial al realizar el cálculo iterativo), y

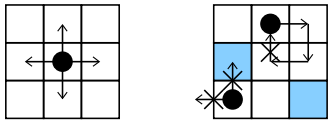
d) razona si puede haber instancias del problema para las que no exista una solución factible.

4. Búsqueda con retroceso — 1,5 puntos

Queremos encontrar un algoritmo de *búsqueda con retroceso* que, partiendo de una casilla inicial (pos_ini= (i, j)), atraviese un laberinto representado mediante un tablero de $n \times n$ y alcance una casilla final (pos_fin= (i', j')).



(a)



(b)

Figura 1: (a) Ejemplo de laberinto y recorrido a traves del laberinto. (b) Izquierda: desplazamientos posibles desde una casilla. Derecha: desplazamientos y caminos no permitidos.

El laberinto, que se facilita como una matriz booleana, está formado por casillas libres (valor `True`), a través de las que se puede transitar, y por casillas bloquedas (valor `False`), a las que no se puede acceder. En una trayectoria a traves del laberinto se pasa de una casilla libre a otra. Para ello hay que tener en cuenta que, dada una casilla libre (i, j), es posible desplazarse directamente desde ella a una de las cuatro casillas adyacentes (en horizontal o vertical, casillas $(i+1, j)$, $(i, j+1)$, $(i-1, j)$ e $(i, j-1)$). La casilla a la que nos desplacemos tiene que ser una casilla libre. Además, hemos de tener cuidado de no salirnos del tablero y, para evitar estar dando vueltas sin fin en el laberinto, no está permitido pasar más de un vez por una misma casilla (en el camino finalmente elegido desde la posición inicial a la final).

En el ejemplo de la figura (a), los datos de entrada del problema serían:

```
n=5; pos_ini=(1,0); pos_fin=(3,4)
lab=[[True,False,True,True,True],
     [True,True,True,False,True],
     [False,True,False,False,True],
     [True,True,False,True,True],
     [True,False,True,False,True]]
```

A continuación, se presenta el esquema de un posible algoritmo de búsqueda con retroceso que resuelve este problema:

```
def laberinto(n,lab,pos_ini,pos_fin):
  sol = [pos_ini]
  def backtracking(ult_pos):
    if is_complete(ult_pos):
      return sol
    else:
      (i,j)=ult_pos
      for pos in [(i+1,j),(i,j+1),(i-1,j),(i,j-1)]:
        if is_promising(pos):
          sol.append(pos)
          found = backtracking(pos)
          if found != None: return found
          sol.pop()
      return None
  return backtracking(pos_ini)
```

Si llamamos a la función `laberinto` con los datos del ejemplo anterior: `laberinto(5,lab,(1,0),(3,4))`, el resultado será:

```
[(1, 0), (1, 1), (1, 2), (0, 2), (0, 3), (0, 4), (1, 4), (2, 4), (3, 4)]
```

que se corresponde con la solución mostrada en la figura (a). Se pide que:

a) Escribas el código correspondiente a la función `is_complete(ult_pos)`, es decir, que definas cuál es el criterio que permite determinar si un estado es una *solución completa* (y, en este caso, también factible).

b) Escribas el código correspondiente a la función `is_promising(pos)`, es decir, que definas cuál es el criterio que permite determinar si un *estado parcial* puede conducir a una solución factible o no.

c) A la vista del código desarrollado en los apartados anteriores, ¿cuál sería la complejidad temporal en el *mejor de los casos* del algoritmo desarrollado? Justifícala. ¿Crees posible reducir dicho coste? Indíca cómo.

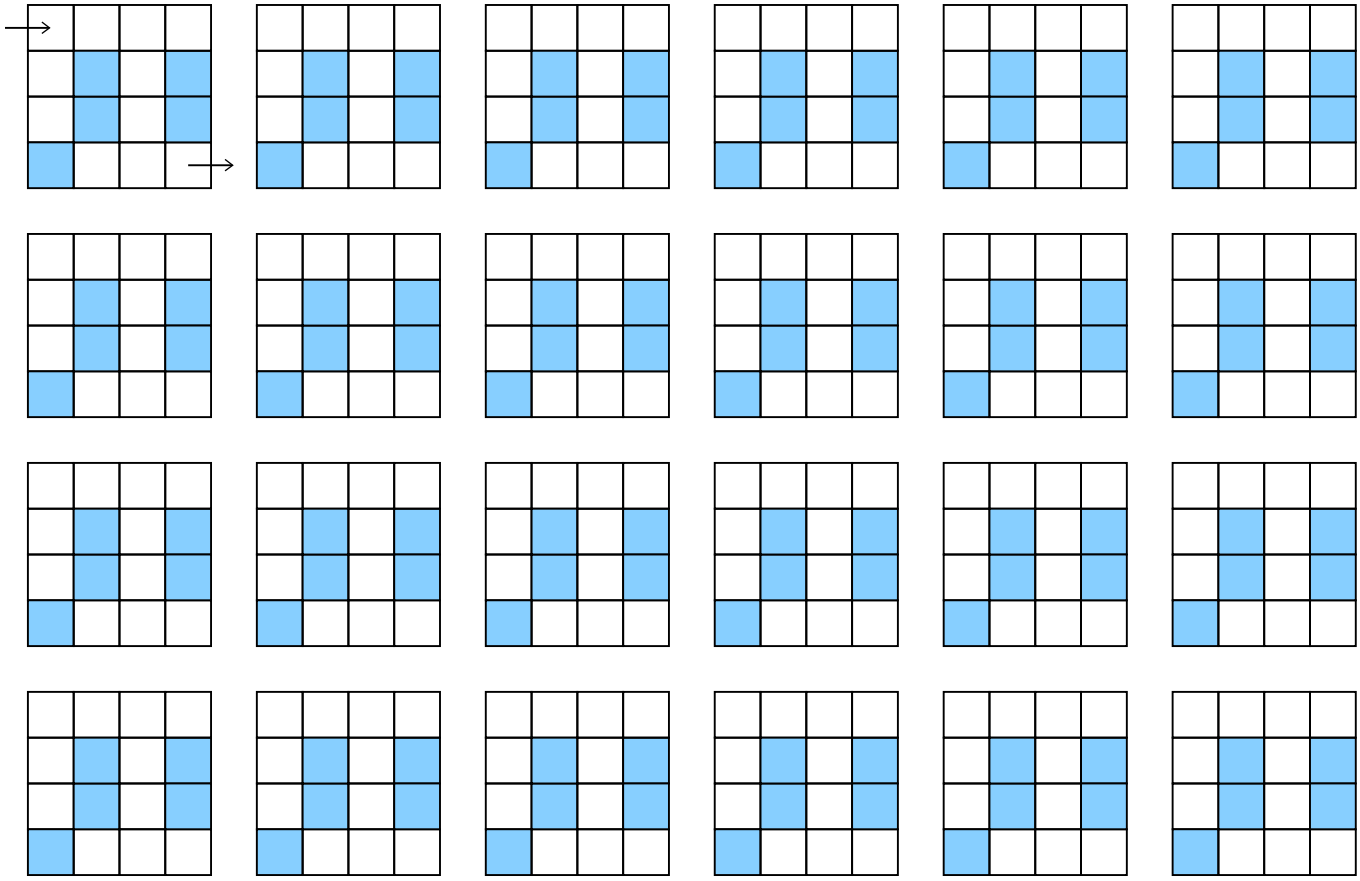
d) Realiza una traza de ejecución de este algoritmo para la llamada

```
laberinto(4,lab,(0,0),(3,3))
```

donde

```
lab=[[True,True,True,True],
     [True,False,True, False],
     [True,False,True, False],
     [False,True,True,True]]
```

Para ello, haz uso de las figuras que aparecen a continuación (sólo de aquellas que necesites). En cada una de ellas debes indicar con claridad la casilla que está estudiando el algoritmo, incluyendo solamente aquellos estados que se generan durante el proceso de búsqueda. Marca de forma especial los estados que se estudian pero se descartan por no superar la condición de ser prometedores. Indica cuál es el resultado devuelto por el algoritmo.



e) Por último, modifica el algoritmo para que, en lugar de devolver un camino a través del laberinto devuelva el número de caminos distintos que permiten atravesar el laberinto desde la casilla de salida a la de llegada.

Examen segunda convocatoria de Esquemas Algorítmicos (IG24)

5 de septiembre de 2006

Marca con una X (una única opción) la parte o partes de la asignatura a la que te presentas (sólo debes rellenarlo si has firmado el contrato del método de evaluación alternativo) :
Parte 1 (Preg. 1 y 2) ( ) Parte 2 (Preg. 3 y 4) ( ) Todo ( ) Renuncio a la evaluación alternativa ( )
(Si lo dejas en blanco o no lo rellenas claramente se asumirá que renuncias a la evaluación alternativa y te presentas a la convocatoria ordinaria.)

1. Voraces — 2,5 puntos

En el almacén de una cafetería hay n cajas con helados de m diversos sabores. Cada caja contiene helados de un único sabor (por supuesto, puede haber varias cajas con helados del mismo sabor). Los propietarios han decidido utilizar para los postres del menú diario de la cafetería una única caja de helados completa (una al día) a partir del 1 de junio y hasta que se acaben todas las cajas (bien porque se hayan consumido, bien porque hayan caducado). Tenemos dos vectores indexados por el número de caja c ($1 \leq c \leq n$): $D[c]$ indica el número de días que faltan para que caduquen los helados de la caja c a partir del 1 de junio, mientras que $S[c]$ indica el sabor de los helados que hay en dicha caja. Además, la cafetería maneja un estudio estadístico que muestra la aceptación de los diversos sabores de los helados; así, tenemos otro vector, A, tal que para cada sabor i ($1 \leq i \leq m$), $A[i]$ indica el número de clientes a los que les gustan los helados de dicho sabor.

Un ejemplo con $n = 10$ cajas y $m = 3$ diferentes sabores podría tener como días de caducidad para las cajas de la 1 a la 10 $D = [10, 5, 1, 20, 8, 1, 5, 5, 3, 10]$, cuyos números de sabores serían $S = [1, 2, 1, 3, 3, 1, 2, 3, 2, 2]$ y donde las respectivas aceptaciones de los tres sabores fueran $A = [50, 30, 40]$.

Por último, cabe tener en cuenta la siguiente restricción adicional: existe una función, `compatible(i,j)` (*no simétrica*), que devuelve `True` si podemos poner de postre un helado de sabor j dado que el día anterior pusimos uno de sabor i y `False` en caso contrario ($1 \leq i \leq m, 1 \leq j \leq m$). Los propietarios de la cafetería nos piden que desarrollemos una aplicación informática que determine la secuencia de cajas de helados que hay que servir en el menú diario, de manera que se maximice el número de clientes a los que les gustan los helados servidos en los diversos menús. Hemos pensado desarrollar una solución mediante un algoritmo *voraz*, pero necesitamos tu ayuda. Se pide que:

a) Ya que se trata de un problema de optimización, indiques las características que ha de poseer una *solución factible*, cuál es la función objetivo que se pretende optimizar y qué propiedad debe cumplir una solución factible para ser considerada óptima.

b) Considera la siguiente estrategia voraz:

   "Ordenamos las cajas de menor a mayor caducidad en una lista C y mientras queden cajas en C:

   - Extraemos la primera caja de la lista,
   - si la caja ha caducado, se descarta,
   - si no ha caducado pero no es compatible con la anterior, se añade al final de una segunda lista C',
   - si la caja no ha caducado y es compatible con la anterior, se añade la caja a la solución y se incrementa en 1 el número de días transcurridos (inicialmente fijado a 0).

   A continuación, y si hay cajas en la segunda lista C', esta se recorre igual que la primera, sólo que en este caso, si la caja extraida ha caducado *o no es compatible* se descarta y, si no, se añade a la solución incrementando también en 1 el número de días."

   Con respecto a esta estrategia:

   1. ¿Garantiza encontrar una solución factible, si es que ésta existe? Justifica tu respuesta (si es negativa, debes dar un contraejemplo).
   2. ¿Encuentra siempre una solución óptima? Justifica la respuesta (si es negativa, debes dar un contraejemplo).
   3. ¿Cuál sería la complejidad temporal y espacial de esta estrategia? Justifícalas.
   4. ¿Es posible que se puedan desechar cajas que no están caducadas y podrían aprovecharse a la hora de servir el postre en algún menú? Si es así, pon un ejemplo.

c) Planteamos otra estrategia voraz similar a la anterior, excepto en que inicialmente lo que hacemos es ordenar las n cajas de mayor a menor aceptación de su sabor, teniendo en cuenta que las cajas de idéntico sabor se ordenan entre sí de menor a mayor caducidad. Responde a las mismas preguntas (1, 2, 3 y 4) del anterior apartado para esta nueva estrategia.

2. Divide y vencerás — 2,5 puntos

La *mediana* M de un conjunto de valores es aquel elemento que tiene tantos elementos menores como mayores en el conjunto (es el valor que, si el conjunto estuviera ordenado, ocuparía la posición central). Sean v y w dos vectores *ordenados* de manera *no decreciente* con n números enteros cada uno ($n > 0$). Se desea implementar un algoritmo que obtenga eficientemente la mediana M de los $2n$ elementos que totalizan ambos vectores.

Para ello, date cuenta que podemos calcular de forma inmediata las medianas de cada vector, M_v y M_w, ya que ambos se encuentran ordenados. Si ocurre que $M_v = M_w$ entonces M coincide con ellas. Si $M_v \neq M_w$, entonces puede ocurrir que $M_v < M_w$ o bien que $M_v > M_w$. En el primer caso, podemos afirmar que M va a ser mayor que M_v pero menor que M_w, mientras que en el segundo sabemos que M va a ser mayor que M_w pero menor que M_v. Fíjate también en que si $n = 1$, entonces la mediana sería el *mínimo* entre los dos números posibles, el de v y el de w.

a) Conociendo este hecho, has de diseñar un algoritmo eficiente basado en la técnica de *divide y vencerás* para obtener M. Recuerda que debes especificar la función que recibe los parámetros y realiza la llamada inicial que desencadena el procedimiento recursivo que efectúa los cálculos necesarios (no olvides indicar igualmente esta llamada).

b) Analiza y justifica la *complejidad temporal* y *espacial* del algoritmo desarrollado en el apartado anterior *en función de* n, indicando si presenta un comportamiento uniforme o existe un mejor y un peor caso. Si el comportamiento es uniforme, modifícalo para que tenga un mejor y un peor caso. Finalmente, señala en qué condiciones (ante qué tipo de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor.

c) Propón un algoritmo iterativo que resuelva este mismo problema *sin* utilizar la técnica de divide y vencerás. Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el resultante tras la realización del apartado b).

3. Programación dinámica — 3,5 puntos

Una cadena de televisión dispone de una serie de n anuncios publicitarios, cada uno con una duración concreta (en segundos) $(d_1, d_2, \ldots, d_{n-1}, d_n)$, que deben emitirse *exactamente en ese orden* en el horario de *prime time*. Para emitir los anuncios, los responsables de la cadena pueden asignar tantas franjas como sean necesarias de D segundos (el tope máximo establecido por ley, obviamente se asume $\forall i$ que $d_i \leq D$). Así pues, en cada franja se emitirá una tanda de anuncios[1], cuya duración nunca puede exceder de D segundos; aunque puede sobrar tiempo. Los responsables de la cadena estiman que si en una franja se pierden p segundos, las perdidas económicas asociadas a la franja son de p^2 €.

La cadena desea averiguar la cantidad de franjas necesarias y cuáles serían los bloques de anuncios que debe emitir en cada una de ellas de tal manera que las *pérdidas sufridas fuesen mínimas*.

Veamos un ejemplo: tenemos 8 anuncios con duraciones, respectivamente, $d_1 = 20$ s., $d_2 = 30$ s., $d_3 = 10$ s., $d_4 = 15$ s., $d_5 = 50$ s., $d_6 = 45$ s., $d_7 = 35$ s. y $d_8 = 20$ s; siendo $D = 75$ s. Una posible solución consistiría en programar 4 franjas: la primera emitiría los anuncios del 1 al 4 ($\texttt{segs}(1,4) = 75 \rightarrow p = 0$), la segunda el 5 ($\texttt{segs}(5,5) = 50 \rightarrow p = 25$), la tercera el 6 ($\texttt{segs}(6,6) = 45 \rightarrow p = 30$) y la cuarta los anuncios 7 y 8 ($\texttt{segs}(7,8) = 55 \rightarrow p = 20$). Las pérdidas serían $0^2 + 25^2 + 30^2 + 20^2 = 1925$ €. Fíjate que no sería posible emitir los anuncios 5 y 6 en una misma franja horaria ya que ambos totalizan 95 segundos; cantidad superior a los 75 que constituye el límite legal.

Sin embargo, existe otra solución mejor, también usando 4 franjas: en la primera se emiten los anuncios del 1 al 3 ($\texttt{segs}(1,3) = 60 \rightarrow p = 15$), en la segunda se emiten los anuncios 4 y 5 ($\texttt{segs}(4,5) = 65 \rightarrow p = 10$), en la tercera se emite el 6 ($\texttt{segs}(6,6) = 45 \rightarrow p = 30$) y en la cuarta el 7 y 8 ($\texttt{segs}(7,8) = 55 \rightarrow p = 20$). En este caso, las pérdidas serían de $15^2 + 10^2 + 30^2 + 20^2 = 1625$ €. Tienes que resolver este problema de la forma más eficiente posible utilizando la técnica de *programación dinámica*, para lo que se te pide:

a) Formaliza el problema en términos de optimización.

b) Plantea la ecuación recursiva que calcula la mínima pérdida posible. Tampoco olvides indicar cómo se calcula la solución óptima (es decir, qué llamada o llamadas iniciales deben efectuarse).

c) Diseña un algoritmo un algoritmo recursivo *con memorización* que resuelva eficientemente la recurrencia planteada en el apartado anterior.

d) Representa el grafo de dependencias entre estados a partir de la recurrencia desarrollada en el apartado b) para el ejemplo propuesto.

e) A partir del grafo anterior, indica y justifica la complejidad temporal y espacial del algoritmo *iterativo* de programación dinámica que resuelve el problema de calcular *la asignación de anuncios a franjas* cuya emisión produce las pérdidas mínimas.

4. Búsqueda con retroceso — 1,5 puntos

Una empresa de desarrollo de software para terceros tiene en perspectiva la realización de n aplicaciones diferentes, cada una solicitada por un cliente distinto. En la empresa trabajan un total de m analistas ($m > n$). La empresa desea asignar un analista *distinto* a cada uno de los proyectos. Para ello, se debe tener en cuenta que no todos los analistas tienen los conocimientos adecuados para desarrollar satisfactoriamente la aplicación. En este sentido, puedes asumir que se dispone de una función *booleana*, `domina(a,p)`, que devuelve `True` si el analista a tiene los conocimientos requeridos para llevar a cabo el proyecto p y `False` en caso contrario.

Además, puesto que ya se han tenido anteriores experiencias profesionales con los clientes y se han producido ciertos "roces" con el personal de la empresa de software, no se desea asignar un analista que haya tenido problemas con el cliente al proyecto que ha solicitado. Nuevamente, asume que existe una función *booleana*, `adecuado(a,p)`, que devuelve `True` si resulta apropiado asignar el analista a al proyecto p y `False` en caso contrario. Hay que desarrollar una solución mediante la técnica de *búsqueda con retroceso* que permita encontrar, si es que existe, la asignación de analistas a proyectos de manera que los analistas asignados conozcan las técnicas necesarias y no hayan tenido problemas anteriormente con el cliente solicitante. Para ello:

[1] Para facilitar los cálculos, se dispone de la función `segs` que, dados los números del primer y último anuncio de una franja, calcula los segundos invertidos en la franja: `segs`(i,j)$= d_i + d_{i+1} + \ldots + d_j = \sum_{i \leq k \leq j} d_k$.

a) Describe cómo vas a representar los estados en la resolución del problema planteado (en qué van a consistir las tuplas, cuantos elementos van a contener, valor posible de esos elementos).

b) Atendiendo a la representación escogida en el apartado anterior, ¿cuál sería la función de *ramificación* (`branch`)? Debes indicar el código o bien una descripción lo *más precisa* posible.

c) Describe la implementación de la función que permite determinar si una solución que se acaba de ramificar es prometedora o no (`is_promising`): igualmente se pide el código o bien una descripción lo *más precisa* posible.

d) Teniendo en cuenta las decisiones tomadas en los apartados anteriores, hemos de depurar el código haciendo *una traza* con los siguientes datos: se dispone de 6 analistas, que deseamos asignar a 4 proyectos distintos, y las funciones `domina(a,p)` y `adecuado(a,p)` devuelven `False` en los proyectos indicados en la siguiente tabla:

analista	`domina`	`adecuado`
1	{1,3}	-
2	{2}	{1}
3	{4}	{2}
4	{1,4}	{4}
5	{3}	{4}
6	{4}	-

Recuerda que debes dibujar el árbol de llamadas incluyendo sólo aquellos estados que se generan durante el proceso de búsqueda y marcando de forma distinta aquellas soluciones que no se ramifican por no ser prometedoras. Indica cuál es el resultado devuelto por el algoritmo.

Control parcial de Esquemas Algorítmicos (IG24)

16 de abril de 2005

1. Estructuras de datos — 1,5 puntos

Deseamos añadir una nueva operación a las disponibles sobre los diccionarios de prioridad: la eliminación de un elemento dada su clave.

Describe brevemente como se podría realizar dicha operación de la forma más eficiente posible, indicando y justificando el coste temporal que presentaría la operación. Acompaña la descripción con un ejemplo gráfico de la estructura completa que, como mínimo, la muestre antes y después de efectuar la operación.

2. Voraces — 4,25 puntos

Un grupo de n amigos se reúnen de tanto en tanto para correrse una juerga. Cuando a uno le apetece, convoca la reunión enviando un mensaje de móvil a todos los demás, en el que indica lugar y hora (sin esperar respuesta, al que le viene bien acude y al que no, no lo hace). Todos los amigos tienen teléfono móvil y saben el número de cada uno de los otros. Se conoce el coste $c(i, j)$ de que un amigo i envie un mensaje a otro j, para todo par de amigos, $i, j \in \{1, 2, \ldots, n\}$. Todos los amigos han suscrito un nuevo tipo de plan intercompañías, "amistad dos a dos", que permite que, aunque el precio que paga cada uno por enviar un mensaje a cualquier otro sea dependiente de la compañia y su contrato particular, cueste lo mismo enviar un mensaje a otra persona suscrita que que ella te lo envie a tí ($c(i, j) = c(j, i)$). Un ejemplo de coste de mensajes para $n = 5$ amigos puede verse en la tabla inferior izquierda que, como se puede comprobar y debido al plan, es equivalente a la que se muestra a su derecha:

c(i,j)	1	2	3	4	5
1	-	15	20	10	25
2	15	-	25	5	30
3	20	25	-	15	20
4	10	5	15	-	20
5	25	30	20	20	-

$\Leftrightarrow$

c(i,j)	1	2	3	4	5
1	-	15	20	10	25
2		-	25	5	30
3			-	15	20
4				-	20
5					-

Los amigos desean modificar el sistema utilizado para quedar (el que convoca manda un mensaje a todos los demás) estableciendo un sistema de reenvío del mensaje de convocatoria con el fin de que el coste total del envío de mensajes sea el menor posible (el coste total es la suma de los costes de todos los mensajes enviados). Para ello, van a establecer una *red de amigos conectados*, que será un conjunto de $n - 1$ pares de valores (i, j) que permita conectar a todos los amigos, es decir, que si un amigo recibe un mensaje de otro que está conectado con él a través de la red, lo reenvíe a los restantes que también aparezcan unidos a él en la red. El problema consiste en encontrar, de todas las posibles combinaciones de amigos conectados, aquella que tiene un mínimo coste.

En el ejemplo anterior, la combinación $\{(1, 3), (3, 2), (4, 3), (5, 4)\}$ es una *red de amigos conectados*: si, p.ej., el amigo 1 decide enviar una convocatoria, se la enviará al 3. Al recibir éste el mensaje, se lo reenviará al 2 y al 4. De éstos, tan sólo el 4 lo reenviará, en este caso al 5. El coste de enviar un mensaje con esta red es $20 + 25 + 15 + 20 = 80$.

El problema, pues, es encontrar la mejor red. Para comenzar, se pide:

a) Se trata de un problema de optimización. Indica las características que debe tener una solución factible, cuál es la función objetivo y que debe cumplir una solución factible para ser la óptima.

b) ¿Puede haber más de una solución óptima para este problema? Sobre el ejemplo presentado, ¿cuál sería una solución óptima?

Para resolver el problema hemos diseñado las siguientes estrategias voraces:

- "La mejor conexión dos a dos": partimos de un amigo cualquiera, x, y seleccionamos al amigo al que le puede enviar un mensaje con el menor coste posible, y, e incorporamos ese par, (x, y), a la solución. A continuación, estudiamos la mejor conexión de y con cualquier otro amigo (excluyendo a x), (y, z), y la añadimos a la solución. Para el nuevo amigo z, buscamos la conexión de menor coste (excluyendo a los dos ya considerados, x e y), y así sucesivamente hasta llegar al último, que no deberá establecer ninguna conexión adicional.
- "La mejor conexión dos a dos, a partir del mejor primero": identica a la anterior, pero el amigo por el que se comienza debe ser uno de los que puede enviar un mensaje con el menor coste posible.
- "Las mejores conexiones aisladas": en este caso, seleccionamos directamente las conexiones (en primer lugar la de menor coste, luego la siguiente, y así sucesivamente), quedándonos con las n-1 de menor coste.

Se pide también:

c) Indica y justifica para cada una de las estrategias propuestas:

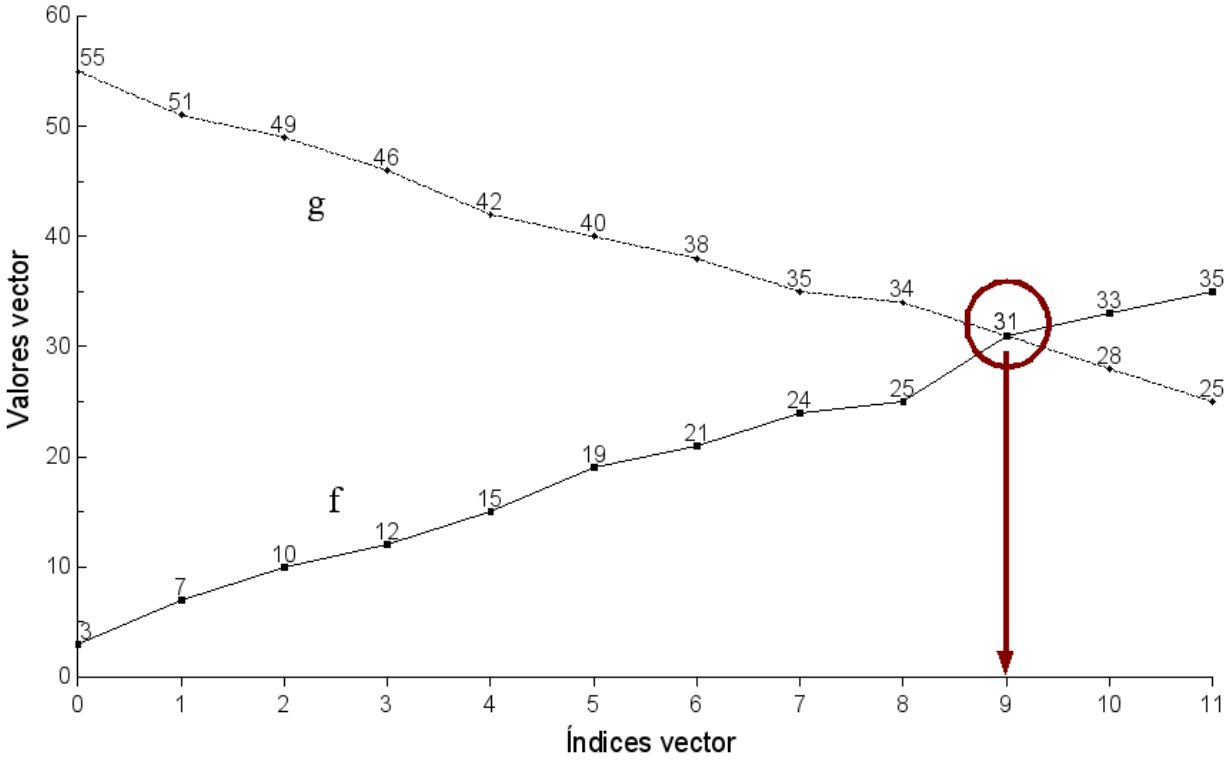
1. Si encuentra siempre una solución factible y, en caso de hacerlo, si siempre es la óptima. Debes demostrar con un contraejemplo los casos en que no esté garantizada la obtención de una solución factible y/u óptima.
2. El coste temporal de la estrategia (indica para ello las estructuras de datos concretas que utilizarías en la implentación de un algoritmo para la estrategia, describiendo los detalles del mismo que den lugar al coste señalado).

d) Si de entre las anteriores has detectado más de una estrategia que garantice la obtención de la solución óptima, dí cuál emplearías.

e) Si crees que puede haber alguna estrategia correcta alternativa a las propuestas, coméntala, señalando también su coste y las estructuras de datos empleadas (en el caso en que haya más de una estrategia posible, cita únicamente la mejor).

3. Divide y vencerás **4,25 puntos**

Sean f y g dos vectores de n elementos que representan los valores que toman dos funciones en el intervalo entero $[0 \ldots n-1]$. Los dos vectores están ordenados, aunque el primero, f, es un vector estrictamente creciente ($f[0] < f[1] < \ldots < f[n-1]$) y g es un vector estrictamente decreciente ($g[0] > g[1] > \ldots > g[n-1]$). Las curvas que representan dichos vectores se cruzan en un punto concreto, y nos interesa saber si dicho punto está contenido entre los que se nos proporcionan en ambos vectores, es decir, si existe un valor i tal que $f[i] = g[i]$ para $0 \leq i \leq n-1$.



a) Diseña un algoritmo recursivo mediante la técnica de divide y vencerás que devuelva el índice en el que coinciden los valores de las funciones o un valor especial en caso de que no exista dicho índice. El algoritmo debe ser lo más eficiente posible.

b) Indica si el algoritmo presenta un comportamiento uniforme o existe un mejor y un peor caso. Si el comportamiento es uniforme, trata de modificarlo para que ante determinadas instancias pueda finalizar antes su ejecución. Señala en qué condiciones (ante que tipos de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor (da algún ejemplo de cada una de esas entradas).

c) Realiza un análisis de la complejidad temporal y espacial del algoritmo presentado, justificando los costes en el mejor y en el peor de los casos.

d) Si el algoritmo propuesto presenta recursividad por cola, elimínala e indica si alguno de los costes calculados en el apartado anterior varía.

e) ¿Se te ocurre algún algoritmo iterativo sencillo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás (no es necesario que lo implementes, tan sólo descríbelo)? Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el que has diseñado.

Segundo control de Esquemas Algorítmicos (IG24)

28 de mayo de 2005

1. Programación dinámica — 4 puntos

Un joven emprendedor que acaba de ser contratado como becario en una sucursal de una gran multinacional se ha trazado como objetivo llegar a ser el presidente de la compañía en el menor tiempo posible. En la empresa hay C categorías de empleados, numeradas jerárquicamente, de manera que a los becarios les corresponde el nivel 1 y al presidente el C. El sistema de promoción interna en la empresa es muy exigente, en concreto:

- No es posible ascender a una categoría c si no se está ocupando una plaza, bien en la categoría inmediatamente inferior (la $c-1$), bien en alguna de las dos inferiores (la $c-1$ y la $c-2$), o bien en alguna de las tres inferiores ($c-1$, $c-2$ y $c-3$). Cada categoría c ($1 < c \leq C$) tiene asociado un valor a_c qué nos indica cuál es la categoría más baja desde la que se puede llegar a ella.

  Por ejemplo, si $C = 7$ y $a = [None, 1, 1, 2, 4, 3, 5]$, podemos ver que el valor de a_7 es 5, lo que indica que a la categoría 7 se permite ascender desde la 5 o la 6, que $a_6 = 3$ (a la categoría 6 se puede ascender desde la 3, la 4 o la 5), o que $a_5 = 4$ (a 5 sólo se puede ascender desde 4). El valor para la categoría 1 es $None$ puesto que no se accede desde ningún otro puesto de la empresa.
- Además de la restricción anterior, únicamente se puede ascender a una determinada categoría c desde una categoría de nivel inferior válida, $c' \in [a_c..c-1]$, si se reúnen una serie de méritos. Los méritos que hay que reunir para ascender a la categoría c dependen tanto de c como de la categoría de procedencia c'.

El becario dispone de una estimación del tiempo que le costará reunir los méritos necesarios para la transición entre cualquier par de categorias posibles $t(c', c)$.

Por tanto, en su objetivo de ascender desde becario a presidente en el menor tiempo posible, se pide:

1. Con las condiciones anteriormente expuestas:
   a) Formaliza el problema en términos de optimización,
   b) plantea la ecuación recursiva de programación dinámica que calcula el mínimo tiempo necesario, indicando cuál debe ser la llamada que se efectué a dicha función para obtener el valor óptimo, y
   c) representa el grafo de dependencias entre las llamadas del algoritmo recursivo para el ejemplo anteriormente citado, *razonando* los costes espacial y temporal con los que se puede efectuar el cálculo sobre dicho grafo (indica explícitamente, justificando tu respuesta, si es posible efectuar una reducción de complejidad espacial al realizar el cálculo iterativo).
2. La compañía está pensando en establecer un requisito adicional a los anteriores, que es que la persona que ocupe la presidencia debe contar con la experiencia de haber pasado al menos por N categorías inferiores. Realiza las modificaciones oportunas en el planteamiento del problema para responder nuevamente a las cuestiones a) y b) planteadas en el apartado 1.

2. Programación dinámica — 3,25 puntos

En una cadena de montaje, la construcción de un artefacto requiere la realización de E ensamblajes de piezas que hay que efectuar de forma secuencial. Para ello, la empresa dispone de P puestos ocupados por operarios que se disponen a lo largo de una cinta en la que se van completando los ensamblajes. En cada puesto se realizan siempre las mismas tareas. Los ensamblajes están numerados entre 1 y E y los puestos también se identifican de 1 a P, siendo el puesto 1 donde se comienza a construir el aparato y el P en el que se realiza el ensamblaje final.

El número de ensamblajes es bastante mayor que el de puestos, por lo que en cada puesto puede que tengan que realizarse varios ensamblajes. Si al puesto de índice p le asignamos los ensamblajes $i, i+1, \ldots, j$, el *disgusto* que siente el operario encargado del puesto puede estimarse mediante una función $d(i, j, p)$ (al tener que realizarse los ensamblajes de forma secuencial, los correspondientes a un puesto tendrán que ser consecutivos; el mismo razonamiento sirve para indicar que si el último ensamblaje asignado a un puesto p es el de índice j, el primero del puesto $p+1$ debe ser el $j+1$).

La empresa quiere saber cuál es la asignación de ensamblajes que debe realizar entre puestos respetando las anteriores restricciones de forma que el disgusto total de los trabajadores sea el mínimo posible. Para ello, ha aplicado la técnica de programación dinámica, llegando a la siguiente ecuación recursiva:

$$D(j,p) = \begin{cases} 0, & \text{si } j = 0 \text{ y } p = 0; \\ +\infty, & \text{si } j = 0 \text{ y } p > 0; \\ +\infty, & \text{si } j > 0 \text{ y } p = 0; \\ \min\limits_{0 \leq i < j} D(i, p-1) + d(i+1, j, p), & \text{en otro caso.} \end{cases}$$

La llamada $D(E, P)$ proporciona el valor del mínimo disgusto que se puede obtener con una asignación válida.
Se pide:

a) Representa el grafo de dependencias para la llamada $D(6,3)$. Indica qué representa un estado (j,p) y un arco entre dos estados (j',p') y (j,p) dentro del grafo.

b) Diseña un algoritmo iterativo que devuelva la *asignación óptima* de ensamblajes que le corresponde a cada puesto.

c) Indica cuál es el coste temporal y espacial del algoritmo diseñado justificando brevemente tu respuesta.

d) ¿Crees que sería posible reducir el coste espacial si únicamente nos interesará obtener el valor del mínimo disgusto? Si es así, índica cuál sería dicho coste y justifícalo.

3. Búsqueda con retroceso — 2.75 puntos

Disponemos de un conjunto A de n números enteros (tanto positivos como negativos) sin repeticiones almacenados en una lista. Dados dos valores enteros, m y C, siendo $m < n$, queremos encontrar un algoritmo de *búsqueda con retroceso* que resuelva el problema de encontrar un subconjunto de A compuesto por *exactamente* m elementos y tal que la suma de los valores de esos m elementos sea C. A continuación, se presenta el esquema de un posible algoritmo de *backtracking* que resuelve este problema.

```
def subconj_m_C(A, C, m):
  s = [None] * m

  def backtrack(i):
    if es_completa(i):
      if es_factible(): return s
    else:
      for s[i] in xrange(len(A)):
        if es_prometedora(i):
          found = backtrack(i+1)
          if found != None: return found
    return None

  return backtrack(0)
```

Si ejecutamos este programa Python con los siguientes datos de entrada: $A = [1, 3, -4, 10, 11, -3]$, $C = 10$ y $m = 3$, el resultado será:

```
Solución: [1, 2, 4]
```

que se corresponde con el siguiente desglose:

```
Elemento de L en pos. 1 es 3
Elemento de L en pos. 2 es -4
Elemento de L en pos. 4 es 11
Total: 3 + -4 + 11 = 10
```

Se pide que:

a) Escribas el código correspondiente a la función `es_completa(i)`, es decir, que definas cuál es el criterio que permite determinar si un estado es una *solución completa*.

b) Escribas el código correspondiente a la función `es_factible()`, es decir, que definas cuál es el criterio que permite determinar si una *solución completa* es una *solución factible* o no.

c) Escribas el código correspondiente a la función `es_prometedora(i)`, es decir, que definas cuál es el criterio que permite determinar si una *solución parcial* puede conducir a una solución factible (en cuyo caso hay que ramificarla) o no.

d) A la vista del código desarrollado en los apartados anteriores, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*?

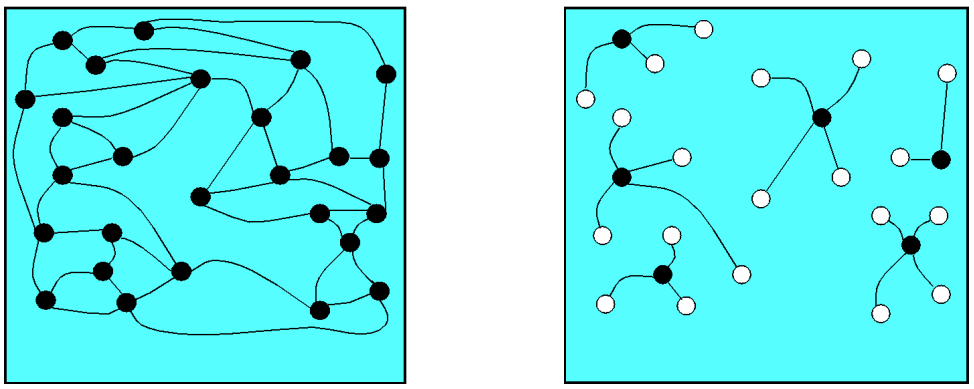
e) Realiza una traza de ejecución de este algoritmo para los valores de entrada: $A = [3, 5, -2, 4, -1]$, $C = 3$ y $m = 2$. Para ello, tendrás que dibujar el árbol de llamadas efectuadas, incluyendo solamente aquellos estados que se generan durante el proceso de búsqueda. Marca de forma especial los estados que se estudian pero se descartan por no superar la condición de ser prometedores.

Examen convocatoria ordinaria de Esquemas Algorítmicos (IG24)

17 de junio de 2005

1. Voraces **2,5 puntos**

Un *conjunto dominante* de vértices en un grafo no dirigido es un subconjunto de vértices del grafo que cumplen que, considerándolos a ellos y a todos los vértices que son adyacentes a ellos, se tienen todos los vértices del grafo. La Figura 1 ilustra este concepto:



(a) (b)

Figura 1: (a) Ejemplo de grafo no dirigido. (b) Un conjunto dominante sobre dicho grafo formado por seis vértices en el que los vértices sombreados son los que forman el conjunto dominante.

Se plantea encontrar, dado un grafo no dirigido, el conjunto dominante compuesto por *el menor número posible de vértices*. En la práctica existen *algoritmos voraces* que intentan solucionar este problema. Se pide que:

a) Ya que se trata de un problema de optimización, debes indicar las características que posee una *solución factible*, cuál es la función objetivo que se pretende optimizar y qué propiedad debe cumplir una solución factible para ser considerada óptima.

b) Para resolver este problema, se plantea la siguiente estrategia voraz: "Ordenar los vértices de mayor a menor grado en una lista L. Mientras queden elementos en L, se selecciona el primero de la lista, se añade al conjunto solución S (inicialmente vacío) y se borra dicho vértice y todos sus sucesores de L. El conjunto de vértices S resultante al final del proceso es la solución." Con respecto a esta estrategia:

   1. ¿Garantiza encontrar una solución factible? Justifica tu respuesta.
   2. ¿Encuentra siempre una solución óptima? Justifica la respuesta (si la respuesta es no, debes incluir un contraejemplo).
   3. ¿Cuál sería el resultado que obtendría si la aplicamos al grafo que aparece en la Figura 2? Justifica la respuesta indicando los diferentes pasos que efectuaría un algoritmo basado en esta estrategia.

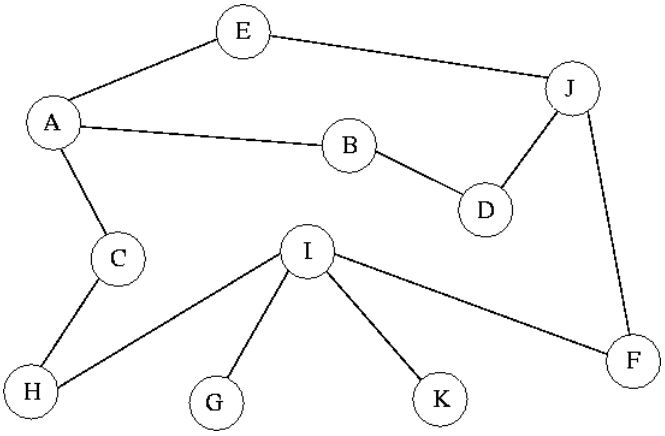


Figura 2: Ejemplo de grafo no dirigido para efectuar las trazas.

c) Planteamos ahora esta otra estrategia voraz: "Ordenar los vértices de menor a mayor grado en una lista L. Mientras queden elementos en L, se selecciona el primer vértice de la lista, v. Para todos los sucesores de este vértice v que estén en L, se coge el de mayor grado u (si no hay ningún sucesor de v en L, u es v). Se añade el vértice u al conjunto solución S (inicialmente vacío) y se borra dicho vértice y todos sus sucesores de L. El conjunto de vértices S resultante al final del proceso es la solución." Responde a las mismas preguntas (1, 2 y 3) del anterior apartado para esta nueva estrategia.

2. Divide y vencerás — 2,5 puntos

Como parte de un programa de procesamiento de textos, la empresa para la que trabajas te pide que hagas una rutina que, dada una cadena alfanumérica c y una secuencia s formada por *exactamente* dos caracteres, devuelva el valor booleano `True` si la secuencia de dos caracteres es una subcadena de la cadena alfanumérica y `False` en caso contrario. Fíjate que no importa el número de veces que la secuencia pueda encontrarse en la cadena, ni las posiciones en las que pueda estar, lo único que interesa es devolver el valor booleano correspondiente.

Por ejemplo, dada la cadena $c =$ `h72uitetei`, ante las secuencias $s =$ `ui`, $s =$ `h7`, $s =$ `ei` y $s =$ `te` devolverá `True` y ante las secuencias $s =$ `2i` y $s =$ `ee` devolverá `False`.

a) Diseña un algoritmo *recursivo* que resuelva este problema empleando la técnica de *divide y vencerás*. El algoritmo desarrollado debe presentar una complejidad espacial *no mayor* de $O(\log n)$. Recuerda que debes indicar explícitamente cuál es la llamada inicial a la función recursiva que desarrolles.

b) El algoritmo que has desarrollado en el apartado anterior, ¿tiene mejor y peor caso o se comporta de manera uniforme? Describe claramente cuál es el mejor y el peor caso. Si el algoritmo presenta un comportamiento uniforme, modifícalo para que tenga un mejor y un peor caso. Finalmente, realiza un análisis de la complejidad temporal y espacial del algoritmo justificando tu respuesta.

c) Existe un sencillo algoritmo iterativo (no desarrollado mediante divide y vencerás) que resuelve igualmente el problema planteado: descríbelo brevemente. Desde el punto de vista asintótico, ¿este algoritmo es mejor o peor que el recursivo realizado mediante divide y vencerás? Responde analizando tanto la complejidad temporal como la espacial.

3. Programación dinámica — 3,5 puntos

En un tablero de ajedrez de tamaño $n \times n$ (donde n es un número par y $n \geq 2$) se disponen una serie de *pesos* (números enteros) en las casillas de color blanco (está prohibido circular por las casillas negras). Disponemos de una ficha que podemos ubicar en cualquiera de las casillas blancas de la columna situada más a la izquierda (las casillas iniciales no tienen asignado ningún peso, es decir, se puede considerar que el peso de todas es el mismo e igual a 0). La ficha sólo puede desplazarse una casilla en cada movimiento y no puede volver atrás, es decir, sólo hay dos movimientos posibles: las dos diagonales de izquierda a derecha (hacia arriba o hacia abajo). Cada vez que la ficha pasa por una casilla obtiene como "recompensa" el peso asignado a la misma. Un *camino* sobre dicho tablero consiste en una *secuencia de movimientos que permiten a la ficha cruzarlo de izquierda a derecha*. Cada camino tiene asociado un *valor*, que es igual a la suma de los pesos asignados a las casillas recorridas por la ficha.

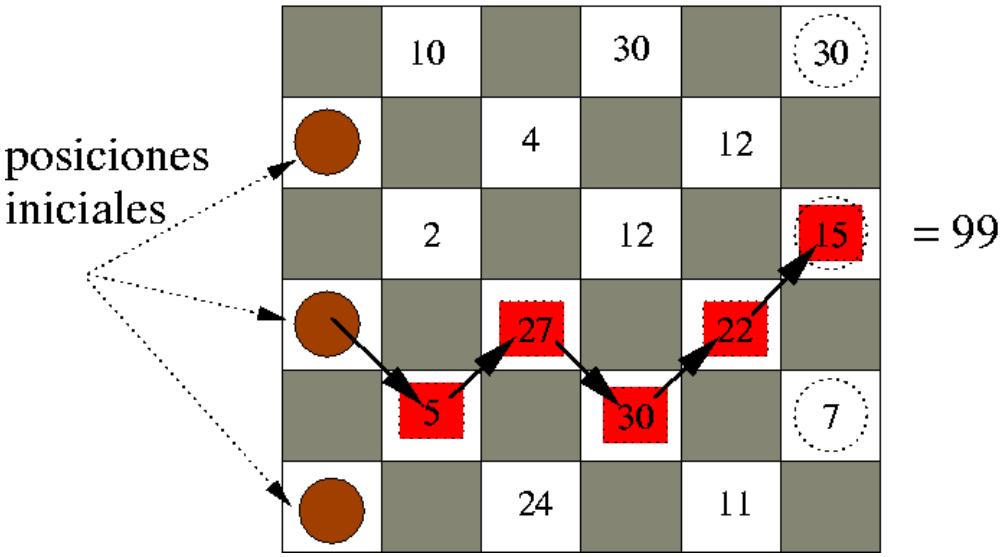


Figura 3: Camino de máximo valor en un tablero de 6×6 ($n = 6$). Las flechas indican el camino efectuado por la ficha. La posición de partida puede ser cualquiera de las casillas blancas de la columna situada más a la izquierda. La posición final puede ser cualquiera de las casillas blancas de la columna situada más a la derecha. Los cuadrados en las casillas indican la serie de recompensas obtenidas. Al final se indica la recompensa conseguida usando el camino de máximo valor.

Los datos de entrada para este problema son la dimensión del tablero n, y una función $p(c, f)$ que, dadas las coordenadas de una casilla (una columna c y una fila f), devuelve el peso almacenado en dicha casilla. Por ejemplo, para el tablero de la Figura 3, algunos valores de f son $f(2, 2) = 5$, $f(2, 6) = 10$, $f(3, 1) = 24$ o $f(6, 4) = 15$. Las casillas negras no tienen definido un valor de la función. Se pide que desarrolles un algoritmo mediante *programación dinámica* que resuelva el problema de encontrar el *valor* asociado al camino de máximo valor sobre un tablero de estas características. Para ello:

a) Presenta una formalización del problema en términos de optimización (el conjunto de soluciones factibles y la función objetivo),

b) plantea la ecuación recursiva que calcula el máximo valor posible de un camino a través del tablero en las condiciones previamente descritas, indicando cuál debe ser la llamada que se efectúe a dicha función para obtener el valor óptimo, y

c) diseña un algoritmo iterativo que devuelva dicho valor, justificando su coste temporal y espacial.

d) Suponiendo que las casillas de partida (las de la primera columna) también tuvieran pesos, indica como afectaría eso a la ecuación recursiva que has planteado y modifícala en consecuencia. Comenta si este hecho tiene incidencia en los costes asintóticos del algoritmo.

4. Búsqueda con retroceso **1,5 puntos**

Un camión cisterna cargado de gasoil sale de un almacén origen X y debe de llegar a un almacén destino Y. Por el camino, debe realizar visitas para llenar el depósito de la caldera del sistema de calefacción de una serie de chalés. El conductor de la cisterna conoce todas las carreteras que hay entre X, Y y los chalés con caldera, así como su distancia en kilómetros. Todas las carreteras son de doble sentido. Por otro lado, el camión cisterna tiene un tacógrafo electrónico que le impide efectuar recorridos de más de T km. consecutivos sin efectuar parada alguna. El programador de la empresa de suministros ha recibido el encargo de desarrollar un programa informático que permita (si es posible) efectuar el recorrido de X a Y *visitando todos los chalés una única vez* y de tal manera que las paradas que debe efectuar el camión para reiniciar la cuenta del tacógrafo coincidan con el llenado de la caldera de los chalés y así se pierda el menor tiempo posible.

El programa recibe como datos: un conjunto, V, con el número que identifica cada chalé (de 1 a n) más el número 0 que identifica el origen X y el $n+1$ que identifica el destino Y; otro conjunto, E, que indica las carreteras existentes (mediante pares) y su distancia en km.; y el número de km. consecutivos que permite recorrer el tacógrafo, T.

Por ejemplo, se han de rellenar las calderas de 5 chalés, $V = \{0, 1, 2, 3, 4, 5, 6\}$, con esta lista de carreteras: $E = \{(0, 1) : 45, (0, 3) : 80, (0, 5) : 40, (1, 3) : 100, (1, 2) : 30, (2, 4) : 50, (2, 5) : 60, (2, 6) : 90, (3, 5) : 40, (3, 6) : 55, (4, 5) : 35, (5, 6) : 70\}$ y $T = 75$. Una posible solución sería: $(0, 1, 2, 4, 5, 3, 6)$ ya que por ninguna de las carreteras empleadas se recorren más de 75 km. Sin embargo, la solución $(0, 3, 1, 2, 4, 5, 6)$ no sería factible ya que la carretera que une el almacén X con el chalé 3 tiene 80 km. y la carretera que une el chalé 3 con el 1 tiene 100 km.

El informático de la empresa está diseñando un algoritmo de *búsqueda con retroceso* para resolver este problema. Sin embargo, tiene unas cuantas dudas a la hora de desarrollarlo y como sabe que estás cursando la asignatura de *Esquemas Algorítmicos* te pide que le ayudes a resolverlas:

a) ¿Qué estructura de datos usarías para representar *de forma compacta* los *datos de entrada* del algoritmo, es decir, los chalés, los almacenes (origen y destino) y las carreteras que los unen (incluyendo su distancia en km.)? Debes indicar una única estructura de datos.

b) ¿Qué estructura o estructuras de datos utilizarías en el algoritmo para *resolver* de la manera más eficiente posible el problema?

c) ¿Cuál sería la función de *ramificación* (*branch*)? Debes indicar el (pseudo)código o bien una descripción lo más precisa posible.

d) ¿Cuál tendría que ser la implementación de la función que permite determinar si una solución es prometedora o no (*is_promising*)? Igualmente se pide el (pseudo)código o bien una descripción lo más precisa posible.

e) Teniendo en cuenta las decisiones tomadas en los apartados anteriores, el informático de la empresa quiere que le ayudes a depurar el código haciendo una traza con estos datos de entrada: $V = \{0, 1, 2, 3, 4\}$, $E = \{(0, 1) : 100, (0, 2) : 80, (0, 3) : 40, (1, 2) : 30, (1, 4) : 70, (2, 3) : 20, (3, 4) : 50\}$, $T = 70$. Debes dibujar el árbol de llamadas marcando de forma distinta aquellas soluciones que no se ramifican por no ser prometedoras.

Examen segunda convocatoria de Esquemas Algorítmicos (IG24)

6 de septiembre de 2005

1. Voraces — 2,5 puntos

Conocemos un algoritmo voraz que resuelve eficientemente el problema de la selección de actividades. Nos interesa saber si existen también soluciones voraces para la *siguiente variante* de dicho problema: disponemos de una gran sala que alquilamos para la realización de actividades y que podemos dividir mediante paneles para la realización simultánea de varios actos. La sala tiene una capacidad máxima para M personas (independientemente de que se divida o no). Por otra parte, tenemos una propuesta de todas las actividades que podemos realizar en la sala durante un día. Cada actividad ocupa un intervalo de tiempo (con una hora de inicio s y una hora de finalización t) y tiene un número máximo de participantes m.

El problema consiste en averiguar cuál es la combinación con mayor número de actividades que podemos realizar en la sala en ese día. No importa que se realicen varias actividades simultáneamente, siempre y cuando no se supere en ningún momento el aforo máximo de la sala. Asumimos que la sala se puede reconfigurar en cualquier momento de forma inmediata y que los tiempos de comienzo y finalización de las actividades coinciden con las horas en punto.

Para resolver el problema hemos diseñado las siguientes estrategias voraces:

a) Ordenar las actividades *de menor a mayor instante de finalización*. Siguiendo dicho orden, considerar cada actividad y *seleccionarla* siempre que con ello no se supere la capacidad máxima de la sala en alguna de sus horas de realización.

b) Ordenar las actividades *de menor a mayor número de participantes*. Siguiendo dicho orden, considerar cada actividad y *seleccionarla* siempre que con ello no se supere la capacidad máxima de la sala en alguna de sus horas de realización.

c) Considerar inicialmente todas las actividades como seleccionadas. A continuación, para cada hora (desde las 00h hasta las 24h) comprobar si se supera el aforo máximo de la sala. Si se supera el aforo en dicha hora, eliminar de forma repetida la actividad con más participantes de las seleccionadas hasta que quepan en la sala los participantes de todas las actividades que aún permanecen seleccionadas.

Indica y justifica, *para cada una de las estrategias propuestas*:

1) Si encuentra o no la solución óptima siempre. En este segundo caso, debes justificarlo con un contraejemplo.

2) El coste temporal de la estrategia. Indica para ello las estructuras de datos que utilizarías en la implementación de un algoritmo para la estrategia, describiendo los detalles del mismo que den lugar al coste señalado.

2. Divide y vencerás — 2,5 puntos

Sea v un vector *ordenado* de n elementos que pueden aparecer repetidos dispuestos en orden decreciente ($v[0] \geq v[1] \geq \ldots \geq v[n-1]$). La función `cuenta` calcula mediante divide y vencerás el número de veces que aparece el elemento x en v:

```
def _cuenta(v,i,k,x):
  if k-i==0: return 0
  elif k-i==1:
    if x==v[i]: return 1
    else:       return 0
  else:
    j=(k+i)/2
    if x > v[j]:    return _cuenta(v,i,j,x)
    elif x < v[j]:  return _cuenta(v,j+1,k,x)
    else:           return _cuenta(v,i,j,x)+_cuenta(v,j,k,x)
def cuenta(v,x):
  return _cuenta(v,0,len(v),x)
```

a) ¿Es correcto el algoritmo? Si crees que no, corrígelo para que, aplicando divide y vencerás, funcione adecuadamente.

b) Indica si el algoritmo presenta un comportamiento uniforme o existe un mejor y un peor caso. Si ocurre esto último, señala en qué condiciones (ante qué tipos de entradas) puede el algoritmo comportarse en el mejor de los casos y en qué condiciones en el peor y da algún ejemplo de cada una de esas entradas.

c) Realiza un análisis de complejidad temporal y espacial del algoritmo presentado, indicando los costes en el mejor y peor de los casos si los hubiera. Justifica los costes alcanzados.

d) Es posible refinar el algoritmo original introduciendo algunas modificaciones que permiten obtener una versión del mismo que presenta un menor coste temporal. Hazlo y di cuál es ese coste. (Por supuesto, el algoritmo resultante debe seguir la estrategia de divide y vencerás).

   Pistas: Una posible vía de ataque es detectar cuál es la parte del algoritmo que contribuye en mayor medida al coste e intentar que no sea así. Otra opción es estudiar aquellos casos en los que el fragmento de vector analizado presenta una solución directa al problema.

e) Propón un algoritmo iterativo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás (no es necesario que lo implementes, tan sólo descríbelo). Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el anterior.

3. Programación dinámica — 3,5 puntos

El problema del *trayecto más probable en el río Congo* puede resolverse mediante un algoritmo de programación dinámica. Recordemos su planteamiento inicial: a lo largo del río Congo hay E embarcaderos (numerados del 1 al E). Es posible ir directamente en canoa desde un embarcadero a cualquiera de los dos siguientes en la dirección de la corriente, pero no más allá del segundo sin hacer escala previa. Para cualquier embarcadero $i = 1 \dots E-1$ conocemos la probabilidad de que un viajero que se encuentra en i decida ir directamente al embarcadero j, y es $p(j|i)$ (por la definición del problema, j sólo puede ser $i+1$ o $i+2$, excepto cuando $i = E-1$, en que j es $i+1$). Se nos pedía averiguar qué trayecto entre el embarcadero 1 y el E es el que sigue con más probabilidad un viajero.

Considera la siguiente variante del problema: nos interesa conocer el trayecto más probable entre el primer y el último embarcadero que *realice exactamente N escalas* (incluidos los embarcaderos origen y destino). Por tanto, desarrolla un algoritmo mediante *programación dinámica* que calcule y devuelva el trayecto más probable que atraviesa N embarcaderos con las restricciones indicadas. Para ello:

a) Presenta una formalización del problema en términos de optimización (conjunto de soluciones factibles y función objetivo),

b) plantea la ecuación recursiva que calcula el valor del trayecto más probable según las condiciones previamente descritas, indicando cuál debe ser la llamada que se efectúe a dicha función para obtener el valor óptimo, y

c) diseña un algoritmo iterativo que devuelva *el trayecto óptimo* (no sólo su valor), justificando su coste temporal y espacial.

d) ¿Podría haber instancias del problema para las que no existiera una solución factible? Razona la respuesta y, si es así, pon algún ejemplo de instancia que no tenga solución factible.

e) Otra versión del problema es aquella en la que nos interesa conocer el trayecto más probable en el que se realizan *al menos N* escalas. Indica como modificarías la solución recursiva que has planteado en el apartado b) para adaptarla a esta variante. Comenta si este hecho tiene incidencia en los costes asintóticos del algoritmo.

4. Búsqueda con retroceso — 1,5 puntos

Dados n números enteros sin signo almacenados en una lista D y un valor T, se nos pide determinar el signo (positivo o negativo) que debe tener cada uno de los n números de D para que, al sumarlos todos ellos, obtengamos el valor T.

Como solución al problema debemos devolver una lista de n componentes, cada uno de los cuales será 1 o -1 en función de si el número que esté en la correspondiente posición de D es positivo o negativo, respectivamente. En caso de no encontrar solución, deberemos devolver el valor especial `None`.

A continuación, se presenta un posible algoritmo de *búsqueda con retroceso* que resuelve este problema:

```
def signos(D,T):
  s = [None] * len(D)
  def backtracking(i):
    if es_completa(i):
      if es_factible(): return s
    else:
      for s[i] in [1, -1]:
        if es_prometedora(i):
          found = backtracking(i+1)
          if found != None: return found
    return None
 return backtracking(0)
```

Ejecutando este programa Python con los datos de entrada $D = [3, 12, 6, 3, 6, 5, 5]$ y $T = 24$, se obtiene como solución la lista $[1, 1, 1, -1, 1, 1, -1]$, que se corresponde con las operaciones: $3 + 12 + 6 + -3 + 6 + 5 + -5 = 24$. Se pide que:

a) Escribas el (pseudo)código correspondiente a la función `es_completa(i)`, es decir, que definas cuál es el criterio que permite determinar cuando un estado es una *solución completa*.

b) Escribas el (pseudo)código correspondiente a la función `es_factible()`, es decir, que definas cuál es el criterio que permite determinar si una *solución completa* es una *solución factible* o no.

c) Escribas el (pseudo)código correspondiente a la función `es_prometedora(i)`, es decir, que definas algún criterio que permita determinar si una *solución parcial* puede conducir a una solución factible (en cuyo caso se ramifica) o no.

d) A la vista del código desarrollado en los apartados anteriores, ¿cuál sería la complejidad espacial del algoritmo desarrollado? ¿Y la complejidad temporal en el *mejor de los casos*?

e) ¿Crees que es posible mejorar el coste temporal introduciendo alguna variable o estructura de datos adicional o haciendo algún tipo de preproceso? Si es así, describe con claridad la mejora o mejoras que introducirías en el algoritmo e indica como afectaría a los costes previamente calculados.

f) Realiza una traza de ejecución de este algoritmo para los datos de entrada: $D = [12, 5, 3, 2, 10]$ y $T = 6$. Debes dibujar el árbol de llamadas efectuadas, incluyendo solamente aquellos estados que se generan durante el proceso de búsqueda. Marca de forma especial los estados que se estudian pero se descartan por no superar la condición de ser prometedores.

Examen parcial de Esquemas Algorítmicos

3 de abril de 2004

1. Estructuras de datos **3,5 puntos**

Hemos de diseñar soluciones eficientes para una serie de problemas seleccionando (o combinando) las estructuras de datos más adecuadas.

Debes indicar con claridad qué estructura de datos (simple u obtenida como combinación de estructuras más sencillas) te permite dar solución a cada problema con la mayor eficiencia posible. Acompaña tu respuesta con un breve análisis de complejidad espacial para la estructura de datos y temporal para las operaciones que debemos efectuar sobre ella.

(Recurre a gráficos ilustrativos de las estructuras si consideras que pueden resultar de ayuda.)

a) Las personas que iniciaron la colonización de una isla provienen todas de diferentes familias. Conforme transcurre el tiempo, los habitantes de la isla se casan entre sí y tienen descendencia. En los juzgados de la isla se registran todas la bodas y nacimientos.

Hay una relación de parentesco natural entre padres, hijos y hermanos. Cuando dos personas se casan, establecen un vínculo de parentesco político y pasan a formar parte de una misma familia. La nueva relación de parentesco hace que todos los integrantes de las respectivas familias de los cónyuges pasen a considerarse miembros de una sola familia. O sea, dos personas son familia si existe una «cadena» de personas que son familia natural o política dos a dos.

La estructura de datos más adecuada para representar este problema es el MF-set. Nos interesa modificarla para poder atender de la manera más eficiente posible a las dos siguientes situaciones:

- Las leyes de la isla permiten que un individuo sea expulsado de una familia. Cuando eso ocurre, dicha persona pasa a constituir una nueva familia de la que es el único miembro. Describe cómo se podría realizar esa operación sobre la estructura.
- Las reglas de cortesía de la isla indican que cuando un individuo se encuentra con otro de diferente familia, lo primero que hace es presentarse mediante la fórmula "Encantado de conocerle, los m miembros de mi familia estaríamos orgullosos si nos honrara con su amistad", donde m es el número total de miembros de su familia. Indica cómo puede disponer lo más rapidamente posible de dicha información y, en el caso en que sea necesario, detalla cómo afectaría la solución propuesta a las demás operaciones básicas de la estructura.

b) Una peña de apostantes juega todas las semanas a la lotería. Entre todos han nombrado a un responsable de hacer el seguimiento del resultado del sorteo. El acuerdo al que han llegado es que cada socio de la peña compra un décimo de lotería y le comunica al responsable el número (es posible que varios socios compren distintos décimos del mismo número). El responsable comprueba semanalmente el resultado del sorteo. El acuerdo que tienen es que, si toca algún premio, la mitad de su importe se lo queda la persona o personas que han comprado los décimos y la mitad restante se reparte a partes iguales entre todos los socios.

El responsable debe poder introducir los números comprados por los socios durante la semana, comprobar si alguno de los números premiados está en poder de la peña y obtener los nombres de todos los socios que tienen cada uno de los décimos premiados lo más rápidamente posible.

c) En la universidad, tras finalizar el periodo de matrícula, para cada titulación se crea una lista de espera única para poder cubrir las posibles bajas, en la que están todos los estudiantes que deseándolo no han podido acceder a la titulación. Cada vez que se produce una baja en una titulación se da paso al estudiante con mejor nota de los que están en la lista de espera de esa titulación. Asimismo, es habitual que en diferentes periodos se incorporen nuevos estudiantes a las listas de espera, provinientes de la selectividad de septiembre, de segundos ciclos de formación profesional o de otras universidades. En cualquier caso, podemos asegurar que nunca habrá más de n estudiantes en cada lista de espera.

Deseamos gestionar las listas de espera de forma que la selección de los estudiantes con mejores puntuaciones cuando se produzcan bajas y la incorporación de nuevos estudiantes a ellas sea lo más eficiente posible.

d) En un taller de la ITV para camiones, éstos deben pasar por una serie de controles (deben pasar necesariamente por todos ellos, aunque no es preciso que lo hagan en un orden concreto). Pese a estar en un mismo edificio, el acceso y salida de cada control se realiza de forma independiente de los demás controles a través de un largo y estrecho corredor en el que se suelen formar colas y que hace que, una vez el camión se incorpora al mismo, únicamente pueda salir tras superar el control o, si es el último de la cola, haciendo marcha atrás.

En la empresa encargada de gestionar ese taller se han dado cuenta de que hay muchos momentos en que algunos controles quedan vacios, mientras en otros hay varios camiones esperando.

Quisieran, por un lado, poder informar a la entrada del edificio a los nuevos camiones que llegan qué control es el que está más despejado. Por otra parte, si un control queda libre, les gustaría poder avisar por radio a un camión de los que están al final de una de las colas de acceso a los otros controles, para que haga marcha atras y se incorpore al control libre. ¿A qué camión? A aquel que (no habiendo pasado aún por el control que queda libre) está en la cola más larga.

e) Desarrolla un nuevo método para la clase *IndexedMinHeap*, llamado *increase*, que reciba un elemento que ya está en el heap y una nueva puntuación para el mismo e incremente su valor en el min-heap. El método debe ejecutarse en $O(\lg n)$, siendo n la talla del heap. Tras su ejecución los distintos elementos del heap indexado deben mantener sus propiedades.

En el caso en que optes por una descripción verbal, deberás acompañar a la misma necesariamente de un ejemplo gráfico que incluya la estructura completa.

2. Divide y vencerás **3,25 puntos**

La siguiente función calcula mediante la técnica de divide y vencerás el número de veces que aparece el elemento x en una lista desordenada L:

```
def veces(x,L):
  if len(L)==0:
    return 0
  elif len(L)==1:
    return 1
  else:
    return veces(x,L[0:len(L)/2])+veces(x,L[len(L)/2:len(L)])
```

a) ¿Es correcto el algoritmo? Si consideras que lo es, demuestra su correción. Si consideras que no lo es, corrígelo para que, aplicando la técnica de divide y vencerás, funcione adecuadamente y demuestra su corrección.

b) Presenta un análisis de complejidad temporal y espacial, indicando los costes en el mejor y peor de los casos si los hubiera. En el análisis del coste temporal utiliza el método del desplegado. Demuestra por inducción que la expresión cerrada a la que llegas es correcta.

c) Obtén una versión recursiva del algoritmo que no haga uso de la operación de corte. Analiza su complejidad computacional.

d) ¿Puedes proponer algún algoritmo iterativo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás y que presente como mucho el mismo coste asintótico? (No es necesario que lo implementes, tan sólo descríbelo).

3. Divide y vencerás **3,25 puntos**

Sea a un vector de enteros diferentes (negativos y/o positivos) dispuestos en orden creciente ($a[0] < a[1] < \ldots < a[n-1]$) y sea i un número entero. El siguiente algoritmo responde a la pregunta de si el elemento que ocupa la posición i en la lista es precisamente i ($a[i] = i$):

```
def _buscar(i,a,p,r):
  if r-p==0:
    return False
  elif r-p==1:
    if i==a[p]:
      return True
    else:
      return False
  else:
    q=(r+p)/2
    if i<a[q]:
      return _buscar(i,a,p,q)
    else:
      return _buscar(i,a,q,r)
def buscar(i,a):
  return _buscar(i,a,0,len(a))
```

a) ¿Es correcto el algoritmo? Si consideras que no lo es, corrígelo para que, aplicando la técnica de divide y vencerás, funcione adecuadamente.

b) Presenta un análisis de complejidad temporal y espacial, indicando los costes en el mejor y peor de los casos si los hubiera.

c) Razona si hay o no hay recursión por cola en la versión original del algoritmo. En cualquier caso, elimina la recursión y obtén una versión iterativa. Analiza a continuación su complejidad espacial y temporal.

d) ¿Puedes proponer una versión del algoritmo original que mejore su comportamiento para el mejor caso? Sí es así, hazlo y analiza su complejidad computacional, indicando para que instancias del problema se comporta mejor el algoritmo.

e) Modifica el algoritmo original para que reciba un valor u que representa un umbral. Cuando se efectue una llamada al algoritmo en la que el subvector que se vaya a analizar tenga un tamaño menor o igual que dicho umbral, el algoritmo deberá sustituir las llamadas recursivas por una búsqueda secuencial del elemento en el vector.

f) ¿Se te ocurre algún método alternativo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás y que iguale o mejore el coste asintótico del algoritmo original? (No es necesario que lo implementes, tan sólo descríbelo).

Examen parcial de Esquemas Algorítmicos

29 de mayo de 2004

1. Voraces — 3,5 puntos

Un alumno debe realizar a lo largo de un semestre N trabajos para otras tantas asignaturas. Cada trabajo le puede proporcionar hasta p_i puntos, para $1 \leq i \leq N$. El alumno conoce el tiempo máximo t_i, para $1 \leq i \leq N$, que tendría que invertir en cada trabajo para obtener la máxima puntuación. Si invierte menos tiempo, la puntuación del trabajo se verá reducida de forma proporcional (p.ej.: si en el trabajo i invierte $t_i/3$, la puntuación total será $p_i/3$). El tiempo máximo total que está dispuesto a invertir en todos los trabajos es T. ¿Cómo debe distribuir el tiempo T entre los trabajos de forma que obtenga la máxima cantidad global de puntos?

Se pide:

a) Formaliza el problema en términos de optimización,

b) propón una estrategia voraz que resuelva el problema y diseña un algoritmo que siga dicha estrategia devolviendo los tiempos que se deben dedicar a cada trabajo y

c) realiza un análisis de su complejidad computacional.

d) Suponiendo que al estudiante le exijan obtener una nota mínima en cada trabajo, $p'_i < p_i, 1 \leq i \leq N$, indica como afectaría esto a la formulación del problema y a la estrategia planteada para poder seguir resolviendo el problema de forma voraz. Señala si con esta restricción hay algún caso para el que no exista una solución factible para el problema.

2. Programación dinámica — 4,5 puntos

El problema de la asignación óptima de recursos puede resolverse mediante un algoritmo de programación dinámica. Recordemos su planteamiento inicial: disponemos de R unidades de un recurso y deseamos asignar cierta cantidad del mismo a cada una de D actividades distintas. Una función $v : \mathbb{N} \times \mathbb{N} \to \mathbb{R}$ nos indica el beneficio que obtenemos al asignar r unidades del recurso a la actividad d con $v(d, r)$. El número de unidades del recurso que podemos asignar a una actividad d debe ser inferior o igual a m_d. Deseamos obtener la asignación de recursos a actividades que maximiza el beneficio total obtenido, es decir, la suma de beneficios que proporciona cada asignación individual.

Considera la siguiente variante del problema: para cada actividad d no sólo hay un límite superior al número de unidades que se le pueden asignar (m_d), sino que también hay un límite inferior (un número mínimo de unidades de recurso) que se le deben asignar a la actividad (m'_d). Por otra parte, tenemos la obligación de consumir todas las unidades disponibles del recurso R.

a) Desarrolla un algoritmo de programación dinámica que determine cuál es la distribución de las unidades de recurso entre las actividades atendiendo a las condiciones de la variante planteada. Para ello:

   1) Formaliza el problema en términos de optimización,
   2) plantea la ecuación recursiva que calcule el valor de la solución óptima y
   3) diseña un algoritmo iterativo que devuelva dicha solución, estudiando su complejidad computacional.

b) ¿Es posible reducir la complejidad computacional si sólo estamos interesados en conocer el valor de la solución óptima? Razona la respuesta y, en caso afirmativo, di cuál es la complejidad espacial resultante.

c) Si sabemos que la máxima diferencia entre el límite inferior y el superior de las actividades que se pueden asignar a cualquier recurso es M (es decir, $M = \text{máx}_{1 \leq d \leq D}(m_d - m'_d)$), siendo $M < R$, ¿podrías expresar de forma más ajustada el coste temporal del algoritmo?

d) ¿Podría haber instancias del problema para las que no existiera una solución factible? Razona la respuesta y, si es así, indica las condiciones que se deberían cumplir para que pueda haber una solución factible y pon algún ejemplo de instancia que no tenga solución factible.

3. Búsqueda con retroceso — 2 puntos

En el problema de la suma de subconjuntos disponemos de N objetos con pesos $w_0, w_1, \ldots, w_{N-1}$ y de una mochila con capacidad para soportar una carga W y deseamos cargar la mochila con una selección arbitraria de objetos cuyo peso sea exactamente W. El siguiente algoritmo resuelve el problema aplicando la técnica de búsqueda con retroceso.

```
def subsets_sum(w, W):
  w.sort()
  return _subsets_sum(0, w, W, [0] * len(w))

def _subsets_sum(i, w, W, x):
  for x[i] in (1,0):
    total_weight = sum([w[j]*x[j] for j in range(i+1)])
    if total_weight == W:
      return x
    elif i < len(w)-1 and total_weight <= W:
      if i < len(w)-2 and total_weight + w[i+1] <= W:
        found = _subsets_sum(i+1, w, W, x)
        if found != None:
          return found
  return None
```

Dibuja el árbol de estados visitados por el algoritmo para una mochila con capacidad para $W = 15$ kilogramos y 5 objetos de pesos $w = [7, 3, 5, 2, 10]$. Al lado de cada estado, indica el valor del peso total ocupado por los objetos considerados. Señala con claridad cuál es la solución factible seleccionada y los pesos de los objetos que la forman.

Examen final de Esquemas Algorítmicos

29 de junio de 2004

1. Estructuras de datos — 1,25 puntos

Indica que estructura o estructuras de datos utilizarías para implementar algoritmos que resuelvan estos problemas y los costes temporales con los que se realizarían las correspondientes operaciones:

a) En un campamento con n niños se han formado m clanes cerrados, siendo m una cantidad desconocida. Cuando un niño quiere ingresar en un clan, debe solicitarlo a uno de sus miembros. Si éste accede, el niño es sometido a un ritual iniciático e ingresa en el clan. Cada clan tiene un cabecilla que es el fundador del mismo y al que no inició nadie. Un clan está formado por al menos dos personas. Deseamos acabar con esa práctica, para lo que necesitamos obtener un listado de clanes (de los nombres de los miembros de cada uno ellos, identificando al cabecilla), pero lo primero es conocer con detalle la situación actual.

   Para ello a cada niño se le pregunta si forma parte de un clan (puede responder sí o no) y, en caso de respuesta afirmativa, a qué persona solicitó su ingreso. Indica la estructura o estructuras de datos con que implementar eficientemente esta operación, el modo de identificar los cabecillas y cómo generar el listado. (Recuerda que un solo niño no forma un clan).

b) A los pacientes que llegan a urgencias en un hospital se les asigna un número entero de 0 a n atendiendo a la gravedad de su afección (n es la máxima gravedad y 0 es la mínima). Se pensó en usar un max-heap para implementar la lista de espera priorizada por gravedad, pero se observó una anomalía: cuando a dos o más pacientes se les asignaba una misma gravedad, podían «salir» del max-heap en cualquier orden. Es imperativo que, a igual gravedad, salgan en el mismo orden en que entraron, es decir, priorizados por el valor de la hora de entrada (que se conoce). Teniendo en cuenta esta restricción, ¿cómo podemos garantizar que el ingreso y la extracción de un paciente de una lista con m pacientes sean operaciones $O(\lg m)$?

2. Divide y vencerás — 2,75 puntos

Sea v un vector *ordenado* de elementos que pueden aparecer repetidos dispuestos en orden creciente ($v[0] \leq v[1] \leq \ldots \leq v[n-1]$). El siguiente algoritmo calcula mediante la técnica de divide y vencerás el número de veces que aparece repetido en el vector su *primer elemento*:

```
def veces(v,x):
    if len(v)==0:
        return 0
    elif len(v)==1:
        if x==v[0]:
            return 1
        else:
            return 0
    else:
        if x<v[len(v)/2]:
            if x==v[len(v)/2-1]:
                return len(v)/2
            else:
                return veces(v[:len(v)/2],x)
        else: #es decir, x==v[len(v)/2]
            return len(v)/2+veces(v[len(v)/2:],x)

print 'El elemento que ocupa la primera posición del vector aparece %d veces' % veces(v,v[0])
```

a) ¿Es correcto el algoritmo? Si consideras que lo es, demuestra su correción. Si consideras que no lo es, corrígelo para que, aplicando la técnica de divide y vencerás, funcione adecuadamente y demuestra su corrección.

b) Presenta un análisis de complejidad temporal y espacial, indicando los costes en el mejor y peor de los casos si los hubiera. En el análisis del coste temporal utiliza el método del desplegado.

c) Obtén una versión recursiva del algoritmo que no haga uso de la operación de corte. Analiza su complejidad computacional.

d) Propón un algoritmo iterativo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás (no es necesario que lo implementes, tan sólo descríbelo). Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el anterior.

3. Voraces — 2 puntos

El problema de la selección de actividades puede resolverse eficientemente con una estrategia voraz. Te recuerdo su planteamiento básico: dada la propuesta de realizar en una sala una serie de n actividades, de las que conocemos la hora de comienzo y la de finalización, $C = \{(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\}$, queremos seleccionar el mayor subconjunto de actividades que no se

solapen, ya que en la sala sólo se puede llevar a cabo una actividad en cada instante. Si una tarea acaba en un instante y otra empieza en ese mismo instante no hay conflicto alguno.

El algoritmo voraz se ejecuta en tiempo $O(n \lg n)$. ¿Es posible efectuar alguna modificación en cada uno de estos supuestos para obtener algoritmos más eficientes? Si es así, indica cómo y acompaña la explicación con el coste de los algoritmos resultantes.

a) Todas las actividades empiezan a la vez.

b) Ninguna de las actividades propuestas se solapa con las otras.

c) Todas las actividades duran lo mismo.

d) Las actividades se nos proporcionan ya ordenadas de menor a mayor instante de comienzo.

e) Las actividades se nos proporcionan ya ordenadas de menor a mayor instante de finalización.

f) Las actividades se nos proporcionan ya ordenadas de mayor a menor instante de finalización.

g) La hora de comienzo y de finalización de las actividades puede modificarse si ello facilita el encontrar una mejor combinación de las mismas, lo único que debería mantenerse es su duración (que se obtiene como $t_i - s_i, 1 \leq i \leq n$).

4. Programación dinámica — 2,75 puntos

Un vendedor ambulante de productos de artesanía realiza una ruta a través de una serie de P pueblos dispuestos a lo largo de una carretera. Parte de la ciudad en la que habita (situada al comienzo de la carretera) y debe llegar a una ciudad que se encuentra en el otro extremo de la carretera sin retroceder nunca. Su sistema de ventas es siempre el mismo: llega a un pueblo, vende toda la mercancía, compra la artesanía típica del pueblo y continúa hasta otro pueblo, en el que repetirá la operación. Al principio del viaje partirá con productos típicos de su ciudad y al acabar deberá vender la mercancía que le quede en la ciudad final de trayecto. El comerciante, una vez ha parado en un pueblo, no vuelve a detenerse hasta haber atravesado un mínimo de dos pueblos más (al ser pueblos vecinos no podría vender a buen precio los objetos).

El comerciante tiene una estimación del beneficio que puede obtener en un determinado pueblo en función de la mercancía que venda en él. Es decir, si numeramos los pueblos entre 1 y P según el orden en que aparecen en la carretera, el beneficio que obtiene al vender la mercancia de un pueblo i en otro j sería, $b(i, j)$, para $1 \leq i < j \leq P$.

Se pretende averiguar cual sería el máximo beneficio (en función del recorrido que haga) que podrá conseguir en su viaje. Se pide:

a) Formaliza el problema en términos de optimización,

b) plantea la ecuación recursiva que calcula el máximo beneficio,

c) representa esquemáticamente el grafo de dependencias entre las llamadas recursivas, calculando los costes espacial y temporal con los que se puede realizar el cálculo sobre dicho grafo (indica explicitamente justificando tu respuesta si es posible efectuar una reducción de complejidad espacial al efectuar el cálculo iterativo), y

d) diseña un algoritmo iterativo que recorra el anterior grafo y devuelva el *valor del máximo beneficio* que puede obtener el vendedor.

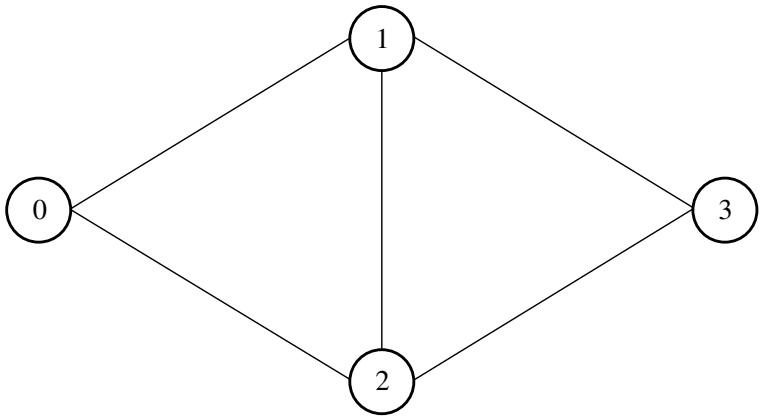
5. Búsqueda con retroceso — 1,25 puntos

En el problema de la existencia de un ciclo Hamiltoniano, dado un grafo $G = (V, E)$, deseamos encontrar un camino que parta de un vértice, termine en el mismo vértice y visite todos los vértices del grafo una sola vez (excepto en el caso del vértice de partida, que deberá coincidir con el de llegada). El siguiente algoritmo resuelve el problema aplicando la técnica de búsqueda con retroceso.

```
from sets import Set
def hamiltonian_cycle(G):
  return _hamiltonian_cycle(G, [G.V[0]]+[None]*(len(G.V)-1), 0, Set([G.V[0]]))

def _hamiltonian_cycle(G, path, m, visited_vertices):
  for v in G.succs(path[m]):
    if v not in visited_vertices:
      path[m+1] = v
      visited_vertices.add(v)
      if len(visited_vertices) == len(G.V):
        if (v,path[0]) in G.E:
          return path + [path[0]]
      else:
        found = _hamiltonian_cycle(G, path, m+1, visited_vertices)
        if found:
          return found
      visited_vertices.remove(v)
  return None
```

Dibuja el árbol de estados visitados por el algoritmo para el grafo mostrado a continuación (considera para ello que el vértice de partida es el 0 y que la función $G.succ(v), \forall v \in G.V$, devuelve *siempre* una lista con los vértices sucesores de v ordenados de menor a mayor). Incluye también, marcándolos de forma especial, aquellos estados que no llegan a visitarse por no cumplir la condición `if v not in visited_vertices`. Señala con claridad cuál es la solución factible seleccionada.



Examen segunda convocatoria de Esquemas Algorítmicos (IG24)

6 de septiembre de 2004

1. Estructuras de datos — 1,25 puntos

Indica qué estructura o estructuras de datos utilizarías para realizar operaciones que resuelvan estos problemas, el coste espacial de dichas estructuras y los costes temporales con los que se realizarían las operaciones indicadas:

a) En el parlamento valenciano los diputados autonómicos se adscriben obligatoriamente a un grupo parlamentario. Inicialmente los grupos se dan de alta y en la sesión inaugural de la legislatura cada diputado debe indicar, cuando es citado, el nombre del grupo parlamentario al que desea pertenecer. Una vez efectuada la adscripción inicial de todos los diputados querremos realizar dos tipos de operaciones: dado un grupo, obtener un listado de todos sus diputados, y dado un diputado, modificar su adscripción de un grupo a otro.

   Nos interesa poder realizar de la manera más eficiente la construcción de la estructura o estructuras que almacenen la información sobre diputados y grupos y las dos operaciones señaladas.

b) La UJI ha decidido crear la oficina de ayuda a la búsqueda de empleo (OFABE) de sus titulados recientes. Para ello, por cada titulación se mantendrá información sobre los alumnos egresados que lo soliciten, teniendo en cuenta tanto su nota media en la carrera como la antigüedad de la solicitud. Cuando llega una petición de personal desde una empresa, la OFABE avisa al titulado de la carrera requerida que presenta un mejor expediente (a igualdad de expediente, al de mayor antigüedad), eliminándolo de su lista tanto si acepta el empleo como si no lo hace (en este último caso, se avisaría al siguiente titulado, mientras que el titulado que rechazó el empleo no podrá solicitar de nuevo su inclusión en la lista hasta pasada una semana, perdiendo además la antigüedad acumulada). El procedimiento descrito es dinámico, ya que en cualquier momento deben poder añadirse nuevos demandantes de empleo (a los que se incorporará con los mismos derechos que los ya existentes, aunque con menor antigüedad), y la gente debe poder darse de baja voluntariamente de las listas de solicitantes.

   Nos interesa poder gestionar lo más eficientemente posible la información de las distintas titulaciones, en concreto la incorporación de nuevos titulados, la obtención de los candidatos a los empleos que se creen y las bajas voluntarias de las personas.

No olvides describir con claridad, cuando sea necesario, cómo las estructuras permiten modelar los problemas y, cuando plantees los costes, qué representan las variables con las que los indicas.

2. Divide y vencerás — 2,75 puntos

Sea v un vector de n elementos *distintos* que cumple que los elementos aparecen dispuestos en orden creciente desde el principio hasta una determinada posición p, y en orden decreciente desde esa posición p hasta el final del vector ($v[0] < v[1] < \ldots < v[p-1] < v[p] > v[p+1] > \ldots > v[n-2] > v[n-1]$), donde la posición p puede variar entre 0 y $n-1$ (es decir, puede estar en uno de los extremos del vector). El algoritmo `pos_mayor` calcula mediante la técnica de divide y vencerás la posición p que ocupa el elemento de mayor valor:

```
def pos_mayor(v):
    return _pos_mayor(v,0,len(v))

def _pos_mayor(v,i,k):
    if k-i==1:
        return i
    else:
        j=(i+k)/2
        if k-i>=3 and (v[j-1]<v[j] and v[j]>v[j+1]):
            return j
        else:
            if v[j-1]>v[j]:
                return _pos_mayor(v,j,k)
            else:
                return _pos_mayor(v,i,j)
```

a) ¿Es correcto el algoritmo? Si consideras que lo es, demuestra su correción. Si consideras que no lo es, corrígelo para que, aplicando la técnica de divide y vencerás, funcione adecuadamente y demuestra su corrección.

b) Realiza un análisis de complejidad temporal y espacial del algoritmo presentado, indicando los costes en el mejor y peor de los casos si los hubiera. En el análisis del coste temporal utiliza el método del desplegado.

c) Razona si hay o no hay recursión por cola en la versión original del algoritmo. En cualquier caso, elimina la recursión y obtén una versión iterativa. Analiza a continuación su complejidad espacial y temporal.

d) Propón un algoritmo iterativo que resuelva el mismo problema sin utilizar la técnica de divide y vencerás (no es necesario que lo implementes, tan sólo descríbelo). Indica el coste asintótico que presenta y si, atendiendo a dicho coste, puede considerarse mejor o peor que el anterior.

3. Voraces **2 puntos**

Con el fin de poder emitir certificados, la UJI dispone en su archivo de una fotocopia cotejada por cada uno de los títulos que ha expedido a sus alumnos desde su creación. Cada título tiene un número de registro único que lo identifica. En la actualidad, las fotocopias de los títulos se encuentran repartidas en varios montones, uno por cada titulación, y ordenadas, dentro de la titulación, en función de ese número. El nuevo archivero cree que resultaría más util disponer de una única clasificación para todas las copias de los títulos de la universidad, para lo cual ha decidido fusionar todos los montones en uno sólo, manteniendo la ordenación según el número de registro. Al estar ya ordenados dentro de cada titulación, ha decidido seguir el siguiente procedimiento: selecciona los montones de dos titulaciones y los fusiona tal y como haría el algoritmo `merge`, con lo que reduce en uno el número de montones; a continuación vuelve a hacer lo mismo con otro par de montones, y así sucesivamente hasta quedarse con ún único montón completamente ordenado.

Al archivero le gustaría poder *minimizar la cantidad total de comparaciones y fusiones entre pares de fotocopias* y cree que dicha cantidad puede depender tanto del número de fotocopias que hay cada montón (que se conoce) como del orden en que realice las fusiones. Por ello ha pensado en varias *estrategias voraces que le permitan determinar cómo seleccionar los montones que se deben fusionar en cada instante* (¡Ojo!, no te confundas, la estrategía voraz NO realiza las fusiones, sino sólo debe indicar cuál es el mejor orden para efectuarlas):

a) Fusionar en una primera etapa la titulación con mayor número de copias con la que tiene menos copias, la segunda con más copias con la segunda con menos copias y así sucesivamente. En una segunda etapa, seguir el mismo procedimiento con los montones resultantes. Repetir esta estrategia hasta quedarse con un único montón.

b) Fusionar en primer lugar las dos titulaciones que tienen un mayor número de copias. A continuación, fusionar el montón resultante con la siguiente titulación en (mayor) número de copías y aplicar este método sucesivamente hasta que quede solamente un montón.

c) Fusionar en primer lugar las dos titulaciones que tienen un menor número de copias. A continuación, fusionar el montón resultante con la siguiente titulación en (menor) número de copías y aplicar este método sucesivamente hasta que quede solamente un montón.

d) Seleccionar indistintamente los montones que se van a fusionar, ya que el número de comparaciones es independiente del orden en que se realizan las fusiones.

Indica para cada estrategia si encuentra o no la solución óptima. En este segundo caso, debes demostrarlo con un contraejemplo. Si crees que puede haber alguna otra estrategia correcta alternativa a las señaladas, descríbela. En todos los casos, indica y justifica el coste temporal de las estrategías voraces propuestas (el coste deberás expresarlo en función del número inicial de titulaciones/montones).

4. Programación dinámica **2,75 puntos**

El problema de la segmentación de un texto en palabras puede resolverse mediante un algoritmo de programación dinámica. Recordemos su planteamiento inicial: dada una secuencia t de caracteres sin espacios en blanco y dadas N palabras con sus probabilidades de aparición en el texto, deseamos segmentar la secuencia de caracteres en la secuencia de palabras más probable, para lo cual disponemos de una función `Pr` que nos indica, dada una secuencia de caracteres, la probabilidad de que dicha secuencia sea una palabra que aparezca en el texto. Las secuencias de caracteres que no pueden aparecer en el texto presentan probabilidad 0. La probabilidad de una secuencia de palabras se define como el producto de las probabilidades individuales de cada palabra.

Considera la siguiente variante del problema: conocemos el número exacto M de palabras que contiene el texto que se nos proporciona, por lo que cualquier segmentación que no contenga tal cantidad de palabras no podrá considerarse válida (¡aunque sea más probable!). Por otra parte, sabemos el tamaño W de la palabra más larga que puede aparecer en la frase.

Desarrolla un algoritmo de programación dinámica que determine cuál es la probabilidad de la mejor segmentación en M palabras del texto facilitado atendiendo a las condiciones de la variante planteada. Para ello:

a) Formaliza el problema en términos de optimización.

b) Plantea la ecuación recursiva que calcule la probabilidad de la segmentación óptima.

c) Diseña un algoritmo iterativo que devuelva el valor de la probabilidad de la segmentación óptima de la frase, estudiando su coste espacial y temporal.

d) Indica explícitamente, justificando tu respuesta, si es posible efectuar una reducción de la complejidad espacial del algoritmo. En tal caso, presenta una nueva versión del anterior algoritmo que incorpore esta mejora.

5. Búsqueda con retroceso **1,25 puntos**

En el problema de la suma de subconjuntos disponemos de N objetos con pesos $w_0, w_1, \ldots, w_{N-1}$ y de una mochila con capacidad para soportar una carga W y deseamos cargar la mochila con una selección arbitraria de objetos cuyo peso sea exactamente W. El algoritmo `subsets_sum` resuelve el problema aplicando la técnica de búsqueda con retroceso.

```
def subsets_sum(w, W):
  w.sort()
  return _subsets_sum(0, w, W, [0] * len(w))

def _subsets_sum(i, w, W, x):
  for x[i] in (1,0):
    total_weight = sum([w[j]*x[j] for j in range(i+1)])
    if total_weight == W:
      return x
    elif i < len(w)-1 and total_weight + w[i+1] <= W:
      found = _subsets_sum(i+1, w, W, x)
      if found != None:
        return found
  return None
```

Dibuja el árbol de estados visitados por el algoritmo para una mochila con capacidad para $W = 8$ kilogramos y 5 objetos de pesos $w = [1, 8, 5, 3, 6]$. Además,

a) Al lado de cada estado, indica de qué tipo es (prometedor, no prometedor, factible o no factible) y el valor del peso total ocupado por los objetos considerados.

b) Incluye también, marcándolos de forma especial, aquellos estados que no llegan a visitarse por no cumplir la condición `elif i <len(w)-1 and total_weight + w[i+1] <= W`.

c) Señala con claridad cuál es la solución factible seleccionada y los pesos de los objetos que la forman. Indica si para el conjunto de objetos propuestos hay otras soluciones factibles además de la devuelta por el algoritmo.