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).

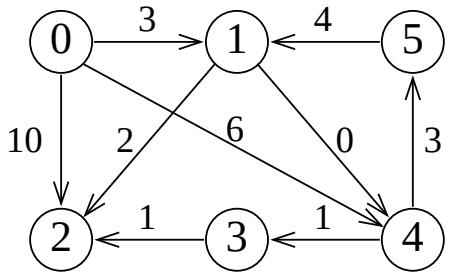
Algoritmos voraces
25 de abril de 2007 (Mañana)

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)

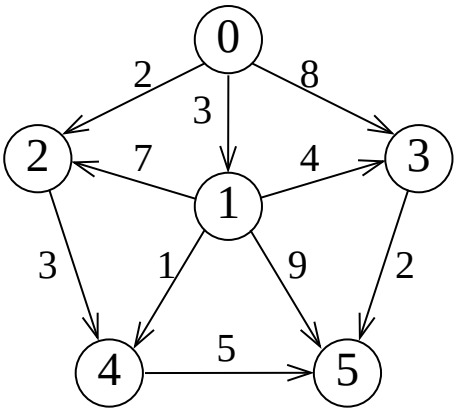
Prueba individual

Nombre: **Grupo:**

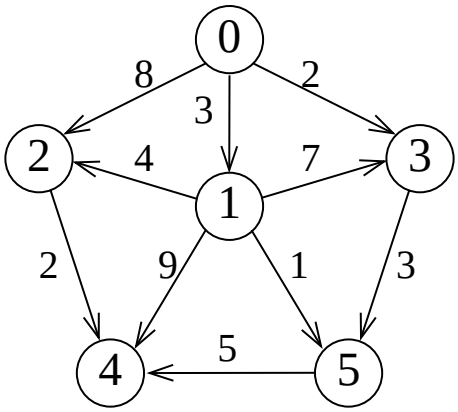
Algoritmo de Dijkstra **35 puntos**

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.

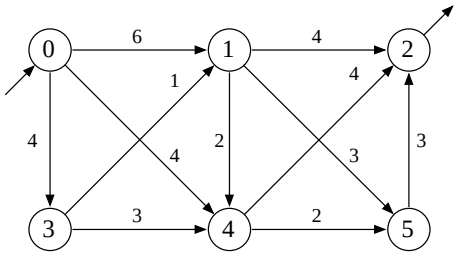
Programación dinámica
10 de mayo de 2007 (Mañana)

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.