Respuestas del examen. 10 de Junio de 2.005

1. (2 puntos).

a)

Vamos a suponer que la llamada inicial es de la forma Pares(a, 1, n). El caso más favorable es cuando no hay que hacer ninguna llamada recursiva, lo que ocurre cuando a[(n+1)/2] es impar. Ojo, los casos n = 1 ó izq = der no son casos favorables, sino tamaños del problema pequeños.

El caso más desfavorable se da cuando se hacen todas las llamadas recursivas posibles. Esto ocurrirá cuando a[(n+1)/2] es par, a[(1+(1+n)/2)/2] = a[(3+n)/4] es par, a[(7+n)/8] es par, etc. Si n es potencia de 2, deberán ser pares los valores de a en las posiciones 2, 4, 8, ..., n/2.

b)

Como en el caso más favorable no se hacen llamadas recursivas, se ejecuta un número constante de instrucciones, y $t(n) = 4 \in \Omega(1)$.

Para el caso más desfavorable, podemos ver que el algoritmo tiene una forma parecida a una búsqueda binaria. Podemos definir el tiempo recurrentemente con: $t(n) = t(n/2) + 4$, contando el número de instrucciones. El caso base es $t(1) = 3$. Resolviéndola queda: $t(n)=4 \cdot log_2 n + 3$, con lo que $t(n) \in O(log\ n)$.

Como las dos notaciones, O y Ω, son distintas no podemos dar un orden exacto, Θ.

c)

En promedio, la mitad de las veces el elemento de a[m] será impar y la otra mitad par. Contando el número de instrucciones tenemos:

$t(n) = 3$, si $n = 1$

$t(n) = 3+1/2 \cdot 1+1/2 \cdot t(n/2)$

La ecuación característica es $(x-1/2)(x-1) = 0$, con lo que $t(n)= c_1/n + c_2$, y queda $t(n) \in \Theta(1)$.

Para la *o* pequeña sólo hace falta obtener las constantes. Imponiendo los casos base 1 y 2 (con valores $t(1) = 3$ y $t(2) = 5$) queda: $t(n) = 7 - 4/n$, con lo que $t(n) \in o(7)$.

d)

Si llamamos k al número de bytes por entero, en cada llamada recursiva se usarán $k(n+3)$ bytes. Se puede ver fácilmente que el número máximo de llamadas recursivas será de $log_2 n$, con lo que el máximo uso de memoria en el peor caso será: $k(n+3)log_2 n \in O(n \cdot log\ n)$.

2.a) (1 punto).

La ecuación se puede resolver con el cambio $n = 6k$ (aunque en este caso sólo será válido para los múltiplos de 6). Otra forma, válida para todos los casos, sería expandiendo la recurrencia:

$t(n)= 3(3t(n-2 \cdot 6)+2)+2= 3^2(3t(n-3 \cdot 6)+2)+2(1+3)= \ldots = 3^k t(n-k \cdot 6)+2(1+3+\ldots+3^{k-1}) = 3^k t(n-k \cdot 6)+3^k-1$

La expansión de la recurrencia acabará cuando $n-k \cdot 6 \le b$, es decir, para $k = \lceil (n-b)/6 \rceil$. Por lo tanto, la ecuación valdrá: $t(n)= 3^{\lceil (n-b)/6 \rceil}(b^2+1)-1$.

2.b) (1 punto).

Está claro que los candidatos aquí serán las tareas. Una vez seleccionadas en cierto orden, intentaremos introducirlas en la primera convocatoria en la que quepan (junio, septiembre o diciembre). Sólo nos queda decidir el orden de selección. Podemos probar dos criterios: seleccionar las tareas de mayor a menor (para ejecutar las más largas antes), y de menor a mayor (para poder empaquetar mejor las tareas).

Almacenaremos el resultado en un array **s**: array [1..n] de entero, indicando s[i] la convocatoria en la que se hace la tarea i (1= junio, 2= septiembre, 3= diciembre). Por otro lado, **conv**: array [1..3] de entero, almacenará el tiempo acumulado de cada convocatoria. El algoritmo podría ser como el siguiente.

```
TareasVoraz (t: array [1..n] de entero; var s: array [1..n] de entero)
    conv:= (0, 0, 0)
    Ordenar t, almacenando los índices en ord  // En orden creciente o decreciente
    para i:= 1, …, n hacer
            si conv[1]+t[ord[i]] ≤ M entonces s[ord[i]]:= 1
            sino si conv[2]+t[ord[i]] ≤ M entonces s[ord[i]]:= 2
            sino si conv[3]+t[ord[i]] ≤ M entonces s[ord[i]]:= 3
            sino devolver "No se puede encontrar solución"
            conv[s[ord[i]]]:= conv[s[ord[i]]] + t[ord[i]]
    finpara
```

Ninguno de los dos garantiza la solución óptima. Aplicando los dos criterios al ejemplo obtenemos:

- De mayor a menor (5, 4, 3, 3, 1): ord= {1, 5, 2, 4, 3} ⇒ s = [1, 2, 1, 2, 1]; conv = [10, 6, 0]
- De menor a mayor (1, 3, 3, 4, 5): ord= {3, 2, 4, 5, 1} ⇒ s = [2, 1, 1, 1, 2]; conv = [7, 9, 0]

3. (2,5 puntos).

Podemos interpretar el problema como una serie de toma de decisiones del tipo "dada una tarea **i**, ejecutarla en junio, en septiembre o en diciembre". Si la ejecutamos en junio, nos quedan **i**-1 tareas por decidir, y $\mathbf{t_i}$ unidades de tiempo menos en junio. Lo mismo para septiembre y diciembre. Por lo tanto, podemos plantear la ecuación: **Tareas (i, j, s, d)**, consistente en resolver el problema con las **i** primeras tareas y con **j**, **s** y **d** unidades de tiempo disponibles en junio, septiembre y diciembre, respectivamente.

La única "pequeña pega" es la función objetivo. Tenemos que minimizar el tiempo asignado a diciembre, pero en caso de empate debemos considerar el de septiembre. Si (j, s, d) son los tiempos asignados en cada convocatoria, la función objetivo podría ser: **Valor**(j, s, d) = d·M + s. Una vez con esto, es sencillo deducir la ecuación recurrente:

Tareas(i, j, s, d) = min {Tareas(i-1, j-t_i, s, d), t_i + Tareas(i-1, j, s-t_i, d), t_i·M + Tareas(i-1, j, s, d-t_i)}

Los casos base serían los siguientes:

Tareas(0, j, s, d) = 0 ; **Tareas(i, j, s, d)** = +∞ si j<0 ó s<0 ó d<0

La construcción del algoritmo a partir de esta ecuación es inmediata. La tabla será de la forma, **T**: array [0..n, 0..M, 0..M, 0..M] de entero, siendo T[i, j, s, d] = Tareas(i, j, s, d). El valor final de la función objetivo se encuentra en T[n, M, M, M]. En definitiva, el algoritmo sería como el siguiente:

```
para i:= 0, ..., n hacer
    para j:= 0, ..., M hacer
        para s:= 0, ..., M hacer
            para d:= 0, ..., M hacer
                si i==0 entonces T[i, j, s, d]:= 0
                sino T[i, j, s, d]:= min (T[i-1, j-t[i], s, d], t[i]+T[i-1, j, s-t[i], d], t[i]*M+T[i-1, j, s, d-t[i]])
escribir ("Tiempo en diciembre:", T[n, M, M, M] div M)
escribir ("Tiempo en septiembre:", T[n, M, M, M] mod M)
```

En la operación min anterior se supone que si un índice se sale del array tenemos un caso base +∞.

4. (2,5 puntos).

Vamos a resolver el problema con backtracking. La solución se representará mediante una tupla **s** = ($\mathbf{s_1}$, $\mathbf{s_2}$, ..., $\mathbf{s_n}$), siendo cada $\mathbf{s_i}$ = 1, 2 ó 3, según la tarea **i** se asigne a la convocatoria de junio, septiembre o diciembre, respectivamente. Obviamente, esto da lugar a un árbol de soluciones 3-ario. La inicialización será **s** = (0, 0, ..., 0). Para controlar el tiempo asignado a cada convocatoria, usaremos un array **conv**: array [0..3] de entero. La función objetivo será la misma del ejercicio anterior. Las funciones básicas del esquema serían las siguientes:

```
operación Inicializar
    voa:= +∞
    nivel:= 1
    s:= (0, 0, ..., 0)
    conv:= (0, 0, 0, 0)

operación Generar (nivel, s)
    conv[s[nivel]]:= conv[s[nivel]] - t[nivel]
    s[nivel]:= s[nivel] + 1
    conv[s[nivel]]:= conv[s[nivel]] + t[nivel]

operación Solución (nivel, s)
    devolver (nivel==n) AND (conv[s[nivel]]<=M)

operación Criterio (nivel, s)
    devolver (nivel<n) AND (conv[s[nivel]]<=M)

operación MasHermanos (nivel, s)
    devolver s[nivel] < 3

operación Retroceder (nivel, s)
    conv[s[nivel]]:= conv[s[nivel]] – t[nivel]
    s[nivel]:= 0
    nivel:= nivel - 1

operación Valor (s)
    devolver conv[3]*M + conv[2]
```

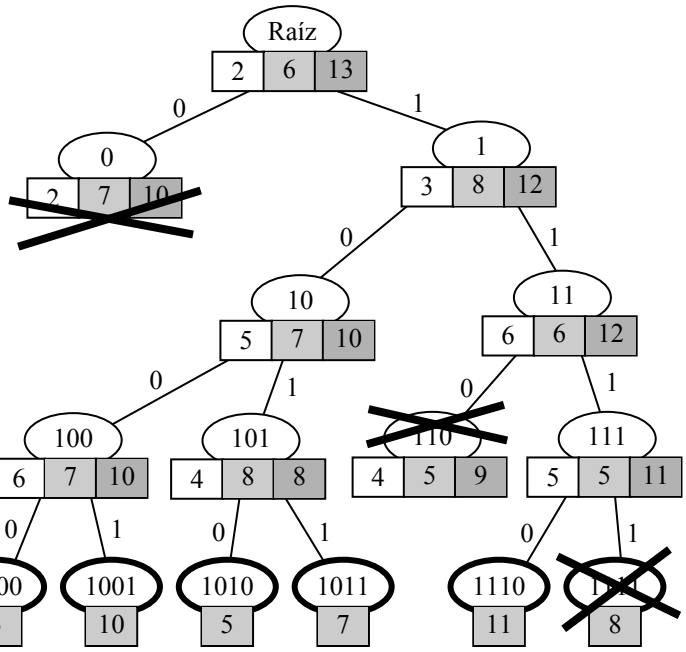
Con estas funciones, y suponiendo el esquema de maximización visto en clase, el algoritmo está ya completamente definido. Podríamos mejorar un poco la eficiencia, haciendo una poda más exhaustiva, por ejemplo, basada en la función objetivo: si el valor actual es mayor que el óptimo actual, podar el nodo. Esto se puede conseguir con una simple modificación de la función criterio:

```
operación Criterio (nivel, s)
    devolver (nivel<n) AND (conv[s[nivel]]<=M) AND (Valor(s) < voa)
```

5. (2 puntos).

Puesto que el problema es de maximización, la variable de poda será: **C** = max {**CI(j)**, **Valor(s)**}, y podamos un nodo **i** cuando **CS(i)** ≤ **C**. La ejecución del algoritmo se muestra abajo. Denotamos los nodos por los valores de las tuplas indicados en el enunciado.

x	C	soa/voa	LNV	Nodos podados
	2		Raíz	
Raíz	3		0, 1	
1	6		0, 10, 11	
10	6		0, 11, 100, 101	
101	7	1011/7	0, 11, 100	
100	10	1001/10	0, 11	
0	10	"	11	0 (al sacarlo)
11	10	"	111	110 (al meterlo)
111	11	1110/11	-	



Si hubiéramos usado la estrategia LC-FIFO la solución final sería la misma (la tupla 1110), pero el recorrido habría sido ligeramente distinto. En concreto, en el 4º paso el nodo escogido, **x**, habría sido el 0 (en lugar del 10).