Examen final – 11 de mayo de 2017

Apellidos, nombre _____ NIF: _____

Pregunta 1 (1 p.)

Responde a las siguientes preguntas (Nota: Resuelve las operaciones intermedias):

a) (0,5 p.) Si la complejidad de un algoritmo es $O(3^n)$, y dicho algoritmo toma 4 segundos para n=2, calcula el tiempo que tardará para n=4

$n_1 = 2$ -- $t_1 = 4$ segundos

$n_2 = 4$ -- $t_2 = ?$

$t_2 = (3^{n2} / 3^{n1}) * t_1$

$t_2 = (3^4 / 3^2) * t_1$

$t_2 = (3^2) * 4 = 36$ segundos

b) (0,5 p.) Considere un algoritmo con complejidad O(logn). Si para t = 2 segundos el método pudiera resolver un problema con un tamaño de n = 8, ¿cuál podría ser el tamaño del problema si dispusiéramos de un tiempo de 4 segundos?

$t_1 = 2$ -- $n_1 = 8$

$t_2 = 4$ -- $n_2 = ?$

$$f(n_2) = \frac{t_2}{t_1} \cdot f(n_1) \Rightarrow n_2 = f^{-1}(\frac{t_2}{t_1} \cdot f(n_1))$$

¡Cuidado! Hay que recordar que la fórmula anterior no se puede simplificar a

$$f^{-1}\left(\frac{t_2}{t_1} \cdot f(n_1)\right) \neq f^{-1}\left(\frac{t_2}{t_1}\right) \cdot n_1$$

$log(n_2) = (t_2/t_1) * log(n_1)$

$log(n_2) = 2 * log(n_1)$

Da igual que tipo de logaritmos aplicamos: logaritmo decimal, logaritmo neperiano o logaritmo en base dos, para todos sale el mismo resultado si utilizamos el suficiente número de decimales. Evidentemente si aplicamos logaritmo base 2 se puede hacer sin calculadora.

$n_2 = 2^{2 * log(8)}$

$n_2 = 2^{2 * 3} = 2^6 = 64 = n_2$

Pregunta 2 (1 p.)

Teniendo en cuenta los siguientes métodos, indica sus complejidades y explica cómo las has calculado:

a) (0,5 p.)

```
public void algohara(int n) {
	if (n > 0) {
		algohara(n-3);
```

```
                for (int i = 0; i < n; i++)
                        for (int j = 0; j < n; j++)
                                System.out.println("i=" + i + "j=" + j); //O(1)
        }
}
```

Divide y vencerás por substracción a = 1; b = 3; k = 2. Así, con a = 1 la complejidad es $O(n^{k+1})$ => **$O(n^3)$**

b) (0,5 p.)

```
public void algoharaparecido(int n) {
        if (n > 0) {
                algohara(n-2);
                for (int i = 0; i < n; i++)
                        for (int j = 0; j < n; j++)
                                System.out.println("i=" + i + "j=" + j); //O(1)
        }
}
```

algohara tiene una complejidad de $O(n^3)$, como vimos antes, da igual que sea n-2 ya que estamos considerando n muy grande. Secuencialmente tenemos dos bucles anidados $O(n^2)$, como están en secuencia la complejidad será el máximo de las dos complejidades → $O(n^3)$

Pregunta 3 (2 p.)

Por favor, responde a las siguientes preguntas sobre el algoritmo de ordenación Quicksort:

a) (1 p.) Dada la siguiente secuencia de números: **6, 5, 4, 3, 9, 7, 1, 8, 2** ordénalos utilizando Quicksort con la estrategia de la mediana a tres para seleccionar el pivote en cada iteración. Indica claramente la traza del algoritmo.

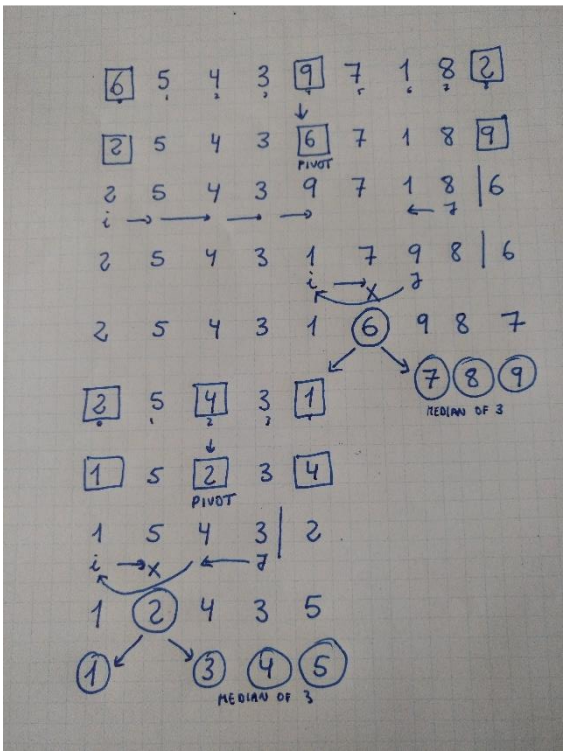
La estrategia mediana a tres para seleccionar el pivote, consiste en coger los elementos en la posición inicial, final y central de la partición y buscar la mediana de los tres (ojo, no confundir con la media). La mediana es el elemento de valor intermedio entre los tres. Esto se repite para todas las particiones que haya que procesar.

Una vez elegido el pivote, se actúa igual que para cualquier otra estrategia de elección del pivote: se "esconde" el pivote en el primer elemento de la partición (o en el último) y se van agrupando los elementos menores al principio, para después intercambiarlo con el último de estos. Ojo, siempre utilizamos intercambios para mover elementos, no desplazamientos.

A continuación, representamos la traza con los cambios del quicksort sobre el vector dado. Para elegir el pivote se utiliza un _ para señalar los elementos que se comparan y , para señalar el elemento que finalmente se elige como pivote. A partir de ahí se muestra el vector después de los intercambios, tanto del pivote si hay que moverlo, como del resto de los elementos para crear las particiones correctas y por último se muestra el pivote con las particiones marcadas por una línea inferior.

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

El pivote también se podría esconder al final de la partición.



b) (0,5 p.) Indica los casos mejor, peor y medio de las complejidades para el algoritmo de Quicksort. Explica claramente cuando se dan los casos mejor y peor.

Caso mejor: O(nlogn)

Caso peor: $O(n^2)$

Caso medio: O(nlogn)

El caso mejor se obtiene cuando en cada iteración seleccionamos como pivote aquel que coincide con la mediana del conjunto de números. De esa forma, siempre se crea un árbol perfecto que lleva a complejidades logarítmicas (la altura del árbol). Así, junto con la

complejidad lineal del particionado, obtenemos la complejidad O(nlogn) final. Evidentemente, esto no tiene que coincidir con elemento que está en la posición central de la partición, salvo que esta esté previamente ordenada, de forma ascendente o descendente.

El caso peor se obtiene cuando en cada iteración seleccionamos como pivote uno que coincida con los extremos en valor del conjunto de números (el primero o el último). Así, en lugar de un árbol perfecto obtendremos una sublista con un elemento menos en cada iteración, dando lugar a una altura del árbol lineal y una complejidad final cuadrática.

Hay que notar que en quicksort el que los elementos de entrada estén ordenados o no realmente no nos proporciona el caso peor o el mejor, sino que dependerá de cómo cogemos el pivote.

c) (0,5 p.) ¿Cuáles son las posibles complejidades (caso mejor, peor y medio) que se pueden obtener con los algoritmos de burbuja, selección e inserción?

Burbuja: $O(n^2)$ siempre

Inserción: $O(n^2)$ excepto en el caso mejor que es $O(n)$

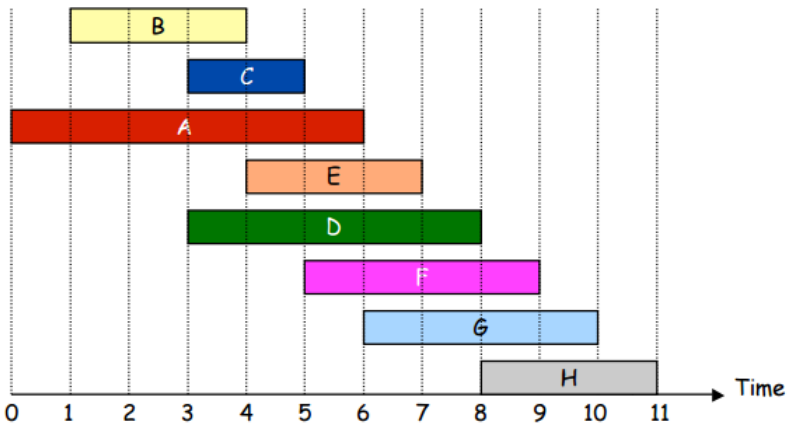
Selección: $O(n^2)$ siempre

Pregunta 4 (2 p.)

Supongamos lo siguiente:

- Tienes que realizar 8 tareas hoy
- Sólo puedes hacer una actividad al mismo tiempo
- Quieres hacer tantas actividades como te sea posible hoy

Así, necesitas crear una planificación que maximice el número de actividades. A continuación, puede verse un ejemplo de actividades y sus tiempos de realización:



Cada una de las actividades toma un tiempo t_i y dos actividades son compatibles si no se superponen. La idea es establecer un orden de acuerdo a un cierto criterio y elegir las actividades que no se superpongan con actividades que ya hayan sido elegidas.

a) (0,75 p.) Describe un heurístico que no ofrezca una solución óptima y pruébalo con un contraejemplo:

Tomar en orden ascendente de s_i (tiempo de inicio i). Tomaremos sólo aquellos que sean compatibles con la selección previa.

A → OK

B no es compatible con A

C no es compatible con A

D no es compatible con A

E no es compatible con A

F no es compatible con A

G → OK

H no es compatible con G

Así haríamos sólo dos actividades, pero según el punto siguiente, podríamos hacer tres actividades con otro heurístico.

b) (0,75 p.) Describe un heurístico que ofrezca una solución óptima (siempre, no solo para el ejemplo) y utilízalo con el ejemplo dado. ¿Cuántas actividades podrías hacer ese día?

Tomar en orden ascendente de f_i (tiempo de finalización i). Tomaremos sólo aquellos que sean compatibles con la selección previa.

B → OK

C no es compatible con B

A no es compatible con B

E → OK

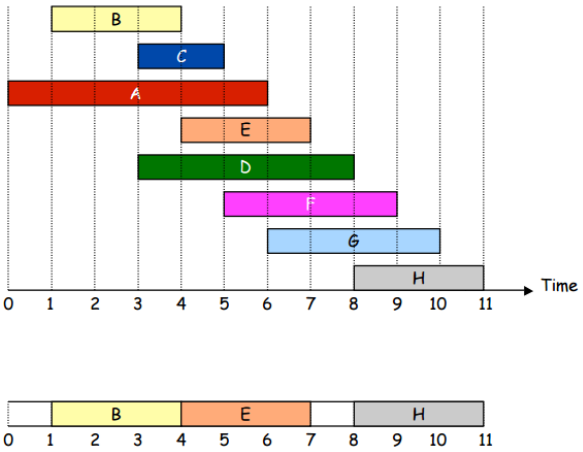
D no es compatible con E

F no es compatible con E

G no es compatible con E

H → OK

Notas: El heurístico debe estar completamente especificado, si está compuesto por varios criterios hay que indicar que función los unifica. Se debe especificar también si una vez calculado el heurístico los elementos se seleccionan en orden creciente o decreciente.



c) (0,5 p.) ¿Cuál sería la mejor complejidad temporal si quisiéramos implementar en Java el algoritmo que nos ofrece una solución óptima? Explícalo brevemente.

O(nlogn) para ello, tendríamos que ordenar previamente las tareas por el momento de finalización lo cual conlleva O(nlogn) y luego un simple bucle que selecciona actividades compatibles O(n), la complejidad final será → max(O(nlogn), O(n))= **O(nlogn)**

Pregunta 5 (2 p.)

Dado un vector de enteros positivos *v* y un valor *sumaDada*, realizar un programa que devuelva un subconjunto del vector que sumen ese valor dado. Si no existe un subconjunto que devuelva la suma exacta, devolver el que más se aproxime, por arriba o por abajo.

a) Para resolverlo se ha utilizado la técnica de backtracking. Completa el código rellenando los huecos. Se dispone de un método *int abs(int num)* que devuelve el valor absoluto del número pasado como parámetro. Ten cuenta que se ha optimizado para que no haga más llamadas recursivas si se encuentra la solución exacta.

```
public class SubconjuntosSumaDada
{
	static int n;		// número de elementos en el conjunto de partida
	static int[]v;			// vector de números positivos diferentes
	static int sumaDada;		// valor de la suma

	static boolean[]marca;		// marca True los elementos usados
	static int sumaParcial;		// suma parcial acumulada hasta un estado

	private static int mejorSuma;
	private static boolean[] mejorMarca;
	private static boolean solExacta;

	static void subconjuntoCercaSumaDada(int nivel) {
		if (nivel == n) // hay ya un conjunto que analizar
		{
			if (abs(sumaParcial- sumaDada) < abs(mejorSuma- sumaDada))
			{
				mejorSuma = sumaParcial;
				mejorMarca = marca.clone();
				if (abs(sumaParcial - sumaDada) == 0)
					solExacta = true;
			}
		} else
```

```
                for (int j = 1; j >= 0; j--) {
                    if (!solExacta)
                    {
                        if (j == 1) {
                            sumaParcial = sumaParcial + v[nivel];
                            marca[nivel] = true;
                        }
                        subconjuntoCercaSumaDada(nivel + 1);
                        if (j == 1) {
                            sumaParcial = sumaParcial - v[nivel];
                            marca[nivel] = false;
                        }
                    }
                }
        }
```

Este backtracking que corresponde a un árbol binario, por la izquierda decidimos usar el número del conjunto que coincide en orden con nivel y por la derecha no usarlo. Por tanto, siempre bajamos hasta el nivel n para que hayamos decidido usar o no usar sobre todos los números del conjunto.

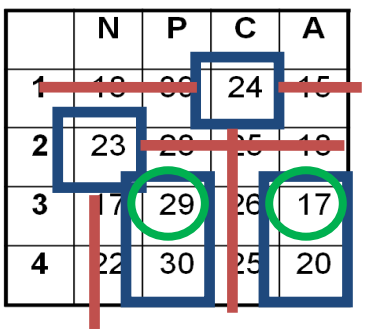
Pregunta 6 (2 p.)

El *cuadriatlón* es una variante de triatlón en la que se combinan Natación(N)-Piragua(P)-Ciclismo(C)-Carrera a pie(A). Ahora que están en auge los deportes de resistencia *César Acero* y tres amigos han decidido participar en una prueba que se realiza por relevos, es decir, cada participante realizará un deporte y le dará el relevo a su compañero. César que es muy competitivo, dispone sus tiempos suyos y de sus compañeros para cada una de los deportes. Quiere diseñar el algoritmo que mediante la técnica de ramificación y poda permita decidir a César qué deportista realizará cada prueba para obtener el mejor tiempo posible.
Deportista 1 (César) (18-39-24-15). Deportista 2 (23-28-25-18). Deportista 3 (17-29-26-17). Deportista 4 (22-30-25-20). Los tiempos corresponden respectivamente a cada uno de los deportes.

a) (0,25 p.) Explicar cuál sería un heurístico de ramificación bueno y cómo se calcula. Y aplicarlo al estado en el que asignamos al deportista 1 a ciclismo y al deportista 2 a natación (y quedan dos deportistas por asignar).

El cálculo del heurístico de ramificación consiste en: La suma de lo ya asignado en cada estado más el caso mejor (mínimo de columnas) de lo que queda por asignar.

Al estado 1→C, 2→N, le corresponde un coste de 24+23+29+17=93.



Sobre esto se pueden aplicar algunas variantes, pero han de estar claramente especificadas, por ejemplo, podemos decidir que los mínimos no pueden coincidir en la misma fila. Entonces, para el último deporte habría que utilizar el valor de 20.

b) (0,25 p.) Explicar cómo se calcula la cota inicial de poda y razonar cuándo se produce el cambio de esta cota.

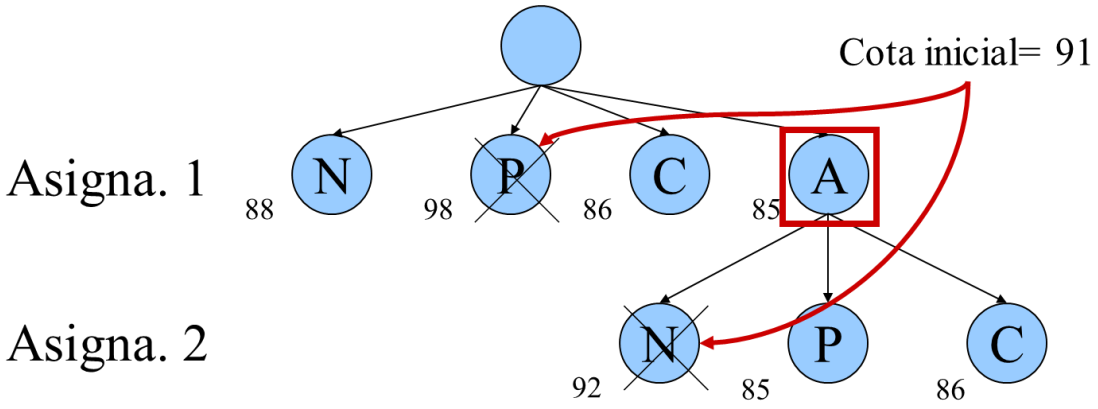
Inicialmente, es la menor de las sumas de las dos diagonales de la matriz de costes.

Valor cota inicial: min(91,92)=91.

Cuando en el desarrollo de los estados del árbol, lleguemos a una solución válida cuyo valor sea menor que el de la cota actual, se cambiará la cota a este nuevo valor.

	N	P	C	A
1	18	39	24	15
2	28	28	25	18
3	17	29	26	17
4	22	30	25	20

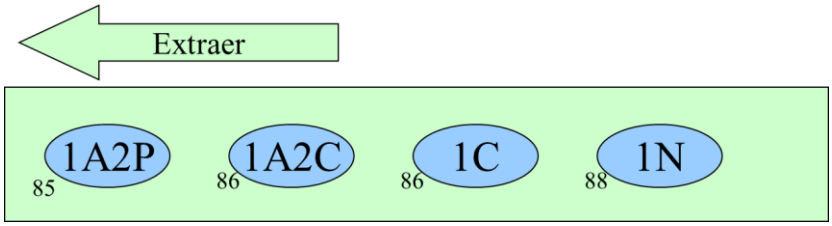
c) (1 p.) Representar el árbol de estados después de haber expandido dos estados del árbol.



Estamos aplicando los heurísticos indicados en los apartados anteriores. Si aplicamos un heurístico de ramificación que consista en sumar sólo las asignaciones realizadas, este heurístico no funcionará correctamente, ya que tendrá el efecto contrario al que necesitamos: cuanto más abajo en el árbol -> peor heurístico tendrá y además, no se pueden aplicar las podas.

Se indica “después de haber expandido dos estados”, aquí hemos expandido el estado inicial y el hijo con el mejor heurístico.

d) (0,5 p.) Representar de forma ordenada los estados que quedan en la cola de prioridad en la situación descrita en el punto c).



En la cola de prioridad no pueden aparecer ni los estados que hemos expandido para obtener sus hijos, ni los podados, ni los estados solución. El resto de los estados aparecerá de menor a mayor ordenados por su heurístico (en realidad debería seguir la estructura de un montículo de mínimos, pero por simplicidad lo pongo así).