Programación III
Febrero de 2007

NOMBRE:
APELLIDOS:
DNI:

- **Cuestión 1 (2 puntos)**
  ¿ Cual es el tamaño del problema, n, que determina el orden de complejidad del siguiente procedimiento? (0.5 puntos). Calcula el orden de complejidad del algoritmo en función de dicho tamaño (1.5 puntos).

```
tipo vector = array de enteros;

procedimiento examen(var a:vector; prim,ult, x: entero): boolean;
var
  mitad: entero;

(1)  si (prim >= ult) entonces
(2)    devolver a[ult] = x
(3)  si no
(4)    mitad = (prim + ult) div 2;
(5)    si (x = a[mitad]) entonces
(6)      devolver cierto
(7)    si no si (x < a[mitad]) entonces
(8)      devolver examen(a, prim, mitad-1, x)
(9)    si no
(10)     devolver examen(a, mitad+1, ult, x)
(11)   fsi
(12) fsi
```

Solución
a) n = (ult - prim)+1

b) Contando las instrucciones tenemos que si se cumple la condición de la línea (1) el coste es cte.. Si se va por la línea (3), puede irse por la la linea (6) (coste cte), por la línea (8) (coste t(n/2) + cte) o por la línea (10) (coste t(n/2) + cte). Por tanto, T(n) = T(n/2) + cte.

Planteamos la recurrencia con reducción del problema mediante división:

n: tamaño del problema a: número de llamadas recursivas n/b: tamaño del subproblema cn^k: coste de las instrucciones que no son llamadas recursivas

$$T(n) = \begin{cases} cn^k & \text{si } 0 \leq n < b \\ aT(n/b) + cn^k & \text{si } n \geq b \end{cases}$$

entonces

$$T(n) \in \begin{cases} \Theta(n^k) & \text{si } a < b^k \\ \Theta(n^k log n) & \text{si } a = b^k \\ \Theta(n^{n\log_b a}) & \text{si } a > b^k \end{cases}$$

En nuestro caso, tenemos que los valores son a = 1, b = 2, cn^k = cte, k = 0. d De donde T(n) $\in \Theta(\log n)$.

- **Cuestión 2** (**2 puntos**)
  Escribe en Java el algoritmo principal del esquema de ramificación y poda utilizado en la práctica del presente curso (no es necesario implementar las funciones auxiliares).

  Solución:

  La implementación más sencilla es dejar los nodos solución dentro de la cola de prioridad. Cuando el primero de la cola sea solución entonces hemos encontrado la solución óptima porque el resto de nodos proporcionan una estimación que aún en el mejor de los casos sería peor:

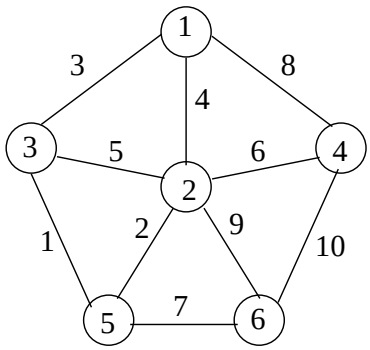
```
public class MejorPrimero implements SearchAlgorithm {
   public Solution SearchSolution(Node nodoInicial) {
Solution solucion;
        Queue<Node> cola = new PriorityQueue<Node>();
        cola.add(nodoInicial);
        while(!cola.isEmpty() ) {
            Node nodo = cola.poll();
            if( nodo.isSolution() ) {
                 return nodo.getSolution();
            } else {
                 for(Node nodoHijo:nodo.generatePossibleChildren()) {
                     cola.offer( nodoHijo );
                 }
            }
        }
        return null;
    }
}
```

  Sin embargo, la solución anterior tiene el problema de que todos los nodos (incluidas algunas soluciones) se guardan en memoria (en la cola de prioridad) aún sabiendo que muchos no pueden proporcionar la solución óptima. Por tanto, es posible refinar el algoritmo anterior para ir eliminando de la cola de prioridad todos los nodos que podamos ir descartando. Para descartar nodos, primero tenemos que encontrar una solución que se convierte en la solución de referencia. Todos los nodos ya existentes con una estimación peor se eliminan de la cola (método depurar) y todos los nuevos nodos con una estimación peor ya no se meten en la cola. Esta versión necesita menos memoria, pero puede tener mayor coste computacional puesto que hay que recorrer la cola cada vez que encontramos una nueva solución. Por otro lado, mantener la cola tiene menos coste porque en general contendrá menor número de nodos.

```
public class RamificacionYPoda implements SearchAlgorithm {
    public Solution searchSolution(Node nodoInicial) {
        Queue<Node> cola = new PriorityQueue<Node>();
        cola.add(nodoInicial);
        // Soluci'on inicial
        int costeMejor = Integer.MAX_VALUE;
        while(!cola.isEmpty() ) {
            Node nodo = cola.poll();
            if( nodo.isSolution() ) {
                int costeReal = nodo.getCost();
                if(  costeReal < costeMejor ) {
                    costeMejor = costeReal;
                    solucion = nodo.getSolution();
                    depurar(cola, costeMejor);
                }
            } else {
                for(Node nodoHijo:nodo.generatePossibleChildren()) {
                    int cota=nodoHijo.calculateEstimatedCost();
                    if(cota < costeMejor) {
                        cola.offer( nodoHijo );
                    }
                }
            }
        }
        return solucion;
    }

    private void depurar(Queue<Node> cola, int costeMejor) {
        Iterator it = cola.iterator();
        List<Node> depurationList = new ArrayList<Node>();
        while(it.hasNext()) {
            Node node = (Node) it.next();
            if(node.calculateEstimatedCost()>costeMejor) {
                depurationList.add(node);
            }
        }
        cola.removeAll(depurationList);
    }
}
```

- **Cuestión 3** (**2 puntos**)
  Aplica el algoritmo de Kruskal al siguiente grafo indicando claramente en cada paso qué arista se selecciona, la evolución de las componentes conexas y la evolución de la solución.



Se ordenan las aristas de menor a mayor coste:

Arista seleccionada	Evolución de las componentes conexas	Evolución de la solución
	{1},{2},{3},{4},{5},{6}	∅
(3,5)	{1},{2},{3,5},{4},{6}	{(3,5)}
(2,5)	{1},{2,3,5},{4},{6}	{(3,5), (2,5)}
(1,3)	{1,2,3,5},{4},{6}	{(3,5), (2,5), (1,3)}
(1,2)	Rechazada*	{(3,5), (2,5), (1,3)}
(2,3)	Rechazada*	{(3,5), (2,5), (1,3)}
(2,4)	{1,2,3,5,4},{6}	{(3,5), (2,5), (1,3), (2,4)}
(5,6)	{1,2,3,5,4,6}	{(3,5), (2,5), (1,3), (2,4), (5,6)}

(*: ambos nodos en una misma partición)

- **Problema** (**4 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 $m < n$ se desea resolver 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.

  La resolución de este problema debe incluir, por este orden:

  1. Elección del esquema más apropiado, el esquema general y explicación de su aplicación al problema (0.5 puntos).
  2. Descripción de las estructuras de datos necesarias (0.5 puntos).
  3. Algoritmo completo a partir del refinamiento del esquema general (2.5 puntos).
  4. Estudio del coste del algoritmo desarrollado (1 punto).

  Selección del esquema algorítmico Se trata de una búsqueda, no de una optimización, por lo que no son aplicables el esquema voraz ni el de ramificación y poda. Tampoco se puede dividir el problema en subproblemas que se resuelvan con exactamente con el mismo procedimiento.

  Tenemos que recorrer un árbol de soluciones candidatas, pero siguiendo aquellas ramas que tengan opción de convertirse en una solución (k-prometedoras) por lo que el esquema de vuelta atrás es adecuado. También es válido un recorrido del árbol de soluciones en profundidad, siempre que el recorrido se detenga en el nivel correspondiente a m.

Dependiendo de la representación del ensayo y de la estrategia de ramificación (generación de hijos) tenemos al menos dos aproximaciones para el esquema de vuelta atrás.

En una primera aproximación (que llamaremos A) podría representarse el ensayo como un vector de m enteros y los descendientes de un nodo de nivel k serían todas los valores no tomados anteriormente. Así del nodo raíz tendríamos n opciones para el nivel 1 del árbol, en el nivel 2 tendríamos $n(n-1)$ hijos, etc. hasta el nivel m. Como se verá más adelante, en el caso peor que es cuando m tiende a n, el número de nodos del árbol es $n!$.

La segunda aproximación (que llamaremos B) es más correcta y consistiría en representar el ensayo como un vector de n booleanos. De este modo los descendientes de un nodo de nivel k serían las opciones de tomar o no el valor $k+1$ del vector de entrada. Es decir siempre se tendrían dos opciones. Cuando m tiende a n, el número de nodos sería 2^n que es mejor que $n!$ Lo que está ocurriendo básicamente, es que la segunda alternativa evita considerar permutaciones, es decir, tomar los mismos números pero en orden diferente. Además esta segunda opción es más fácil de implementar.

A continuación mostraremos la implementación de las dos alternativas. Consideramos los siguientes apartados para cada una de las alternativas:

Alternativa A

1. **Esquema general**

```
fun vuelta-atras(e: ensayo) dev (booleano, ensayo)
    si valido(e) entonces
        dev (cierto,e)
    sino
        listaensayos <-- compleciones(e)

        mientras no vacia(listaensayos) y no es_solucion hacer
            hijo <--  primero(listaensayos)
            listaensayos <-- resto(listaensayos)

            si esprometedora(hijo) entonces
                (es_solucion,solucion) <-- vuelta-atras(hijo)
            fsi
        fmientras
        dev (es_solucion, solucion)
     fsi
ffun
```

   esprometedora comprueba si se cumplen las condiciones de poda, es decir será una función que compruebe que el número de sumandos ya seleccionados es menor que m.

   compleciones es una función que considera todos los numero que aún no han sido probados como siguiente elemento.

   valido comprueba que se haya alcanzado una solución.

2. **estructuras de datos**

```
Tipo ensayo=
    candidatos: vector de enteros //numeros que aun no se han probado
```

```
    solucion: vector de enteros   //numeros de la posible solucion
    suma: entero                  // valor alcanzado hasta ese momento
    num_sumados: entero           // cantidad de numeros que ya se han sumado
```

3. **algoritmo completo**

```
funcion compleciones(e: ensayo) dev lista_compleciones
    lista_compleciones <-- lista_vacia
    para cada i en e.candidatos hacer
      nuevo_ensayo <-- e
      eliminar(nuevo_ensayo.candidatos, i)
      anadir(nuevo_ensayo.solucion, i)
      nuevo_ensayo.suma <-- e.suma + i
      nuevo_ensayo.num_sumados <-- e.num_sumados + 1
      lista_compleciones <-- anadir(lista_compleciones, nuevo_ensayo)
    fpara
    dev lista_compleciones

funcion esprometedora(e:ensayo) dev booleano
  si (e.num_sumados < m) entonces
    dev cierto
  si no
    dev falso

funcion valido(e:ensayo) dev booleano
  si (e.suma = C) y (e.num_sumados = m) entonces
    dev cierto
  si no
    dev falso
```

4. **Complejidad del algoritmo** Una cota superior al tiempo de ejecución sería el número de nodos del árbol, es decir las combinaciones de n elementos tomados de m en m, multiplicado por el número de posibles permutaciones, ya que estas no se excluyen:

$$T(n,m) = \left(\begin{array}{c} n \\ m \end{array} \right) m! = \frac{n!}{(m-n)!m!} m! = \frac{n!}{(n-m)!} = n(n-1)\cdots(n-m+1)$$

Llegamos al mismo resultado calculando el número de hijos de cada nivel del árbol: el número de hijos de primer nivel es n, el numero de hijos del segundo nivel es $n(n-1)$, hasta llegar al nivel $(n-m+1)$, cuyo número de hijos es $n(n-1)\cdots(n-m+1)$.

Podemos observar que

$$\begin{array}{l} \text{si } \; m \to n \Longrightarrow T(n) \in O(n!) \\ \text{si } \; m \to 1 \Longrightarrow T(n) \in O(n) \end{array}$$

Alternativa B

1. **Esquema general**

```
fun vuelta-atras(e: ensayo, k ) dev (booleano, ensayo)
    // e es k-prometedor
    si K=limite entonces
        dev (cierto,e)
```

```
    sino
        listaensayos <-- compleciones(e,k+1)

        mientras no vacia(listaensayos) y no es_solucion hacer
            hijo <--  primero(listaensayos)
            listaensayos <-- resto(listaensayos)

            si esprometedora(hijo,k+1) entonces
                (es_solucion,solucion) <-- vuelta-atras(hijo,k+1)
            fsi
        fmientras
        dev (es_solucion, solucion)
     fsi
ffun
```

En la llamada inicial a *vuelta-atras*, todas las posiciones de *e.candidatos* se inicializan a *true* y k toma el valor 0.

2. **Estructuras de datos**

```
datos: vector de enteros //lista de numeros de entrada
Tipo ensayo=
    candidatos: vector de booleanos //indica si el numero de cada posicion
                            // se incluye en la solucion
    suma: entero            // valor alcanzado hasta ese momento
    num_sumados: entero     // cantidad de valores true de los candidatos
```

3. **algoritmo completo**

```
funcion compleciones(e: ensayo, k:entero) dev lista_compleciones
    lista_compleciones <-- lista_vacia
    nuevo_ensayo <-- e    // el valor asignado a la posicion K es true
    nuevo_ensayo.suma <-- e.suma + datos[k]
    nuevo_ensayo.num_sumados <-- e.num_sumados + 1
    lista_compleciones <-- anadir(lista_compleciones, nuevo_ensayo)

    nuevo_ensayo <-- e    // el valor asignado a la posicion K es false
    nuevo_ensayo.candidatos[k] <-- false
    lista_compleciones <-- anadir(lista_compleciones, nuevo_ensayo)

    dev lista_compleciones

funcion esprometedora(e:ensayo) dev booleano
  si (e.num_sumados < m) entonces
    dev cierto
  si no
    dev falso

funcion valido(e:ensayo) dev booleano
  si (e.suma = C) y (e.num_sumados = m) entonces
    dev cierto
  si no
    dev falso
```

4. **Complejidad del algoritmo** En cada nivel el árbol se divide como máximo en dos ramas (si no se ha llegado a tener m posicione a true). Luego una cota superior es 2^n.

Podemos observar que

$$\begin{array}{l} \text{si} \;\; m \to n \Longrightarrow T(n) \in O(2^n) \\ \text{si} \;\; m \to 1 \Longrightarrow T(n) \in O(n) \end{array}$$

Este algoritmo sería más eficiente que el anterior cuando $m \to n$.