EJEMPLOS DE PROBLEMAS RESUELTOS

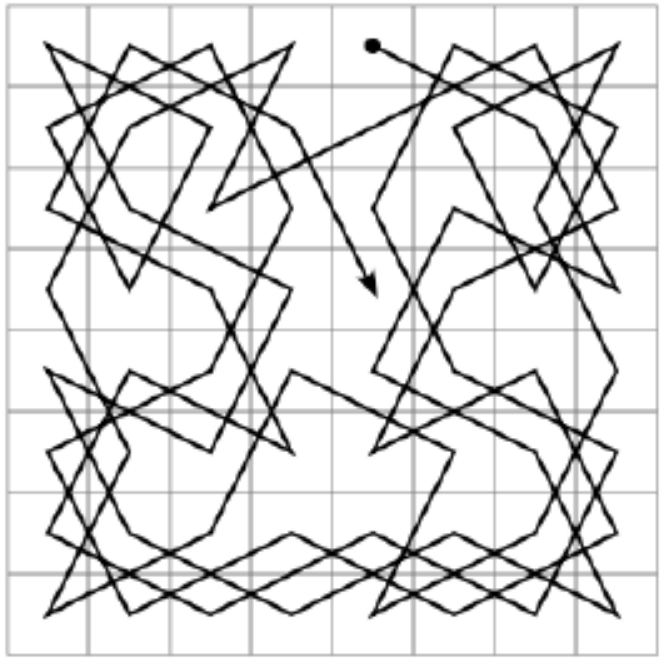
UNIVERSIDAD DE MÁLAGA
E.T.S. INGENIERÍA INFORMÁTICA

Fco Javier Moreno Barea

1. Considerar un tablero de ajedrez de N x N escaques y un caballo situado en la esquina inferior izquierda. Dado el movimiento de esta pieza en el juego, encontrar una secuencia de saltos del caballo que pase una única vez por cada casilla del tablero, o determinar que dicha secuencia no existe.

23	10	15	4	25
16	5	24	9	14
11	22	**1**	18	3
6	17	20	13	8
21	12	7	2	19

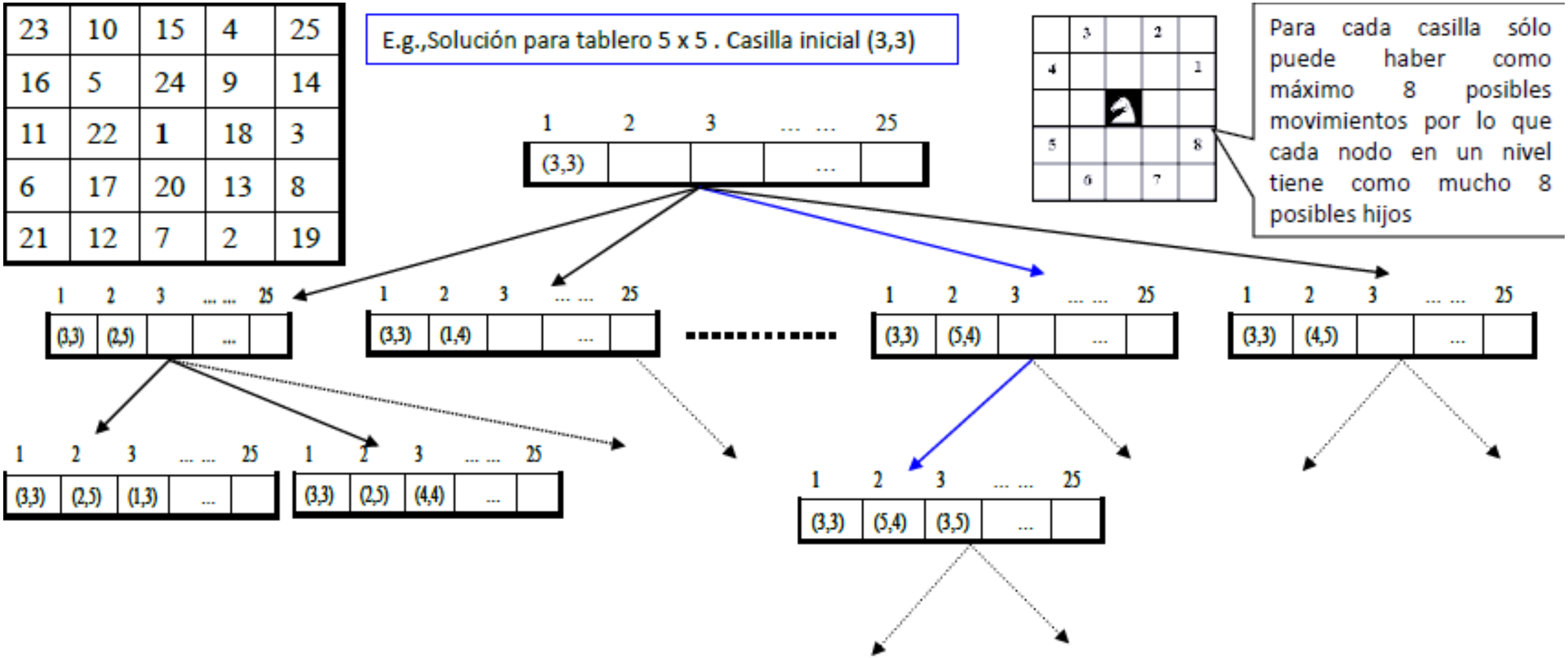
Una Solución para tablero 5 x 5 (son 24 saltos del caballo empezando desde la casilla 1)



Una Solución para tablero 8 x 8 (son 63 saltos del caballo)

¿ Cómo se representa una solución ?
Una forma es representarla como una tupla de $n_x n$ posibles coordenadas que representan los movimientos que va dando el caballo. Cada variable i de la tupla puede tomar un valor (x,y) que depende del valor de la variable i-1.

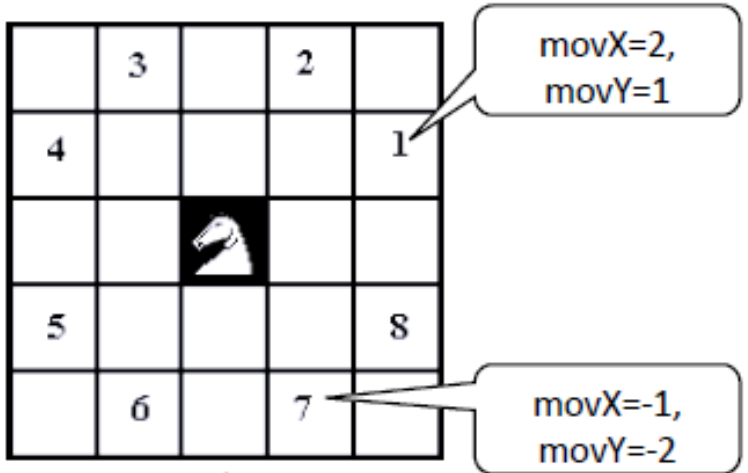
1. Considerar un tablero de ajedrez de N x N escaques y un caballo situado en la esquina inferior izquierda. Dado el movimiento de esta pieza en el juego, encontrar una secuencia de saltos del caballo que pase una única vez por cada casilla del tablero, o determinar que dicha secuencia no existe.



1. Considerar un tablero de ajedrez de N x N escaques y un caballo situado en la esquina inferior izquierda. Dado el movimiento de esta pieza en el juego, encontrar una secuencia de saltos del caballo que pase una única vez por cada casilla del tablero, o determinar que dicha secuencia no existe.

Inicialmente el tablero está vacío (sin saltos) donde cada celda vale 0. Se parte de una casilla inicial.
Recorremos todas las casillas menos las que no pueden ser visitadas.

1. Cada casilla sólo puede visitarse una vez.
2. Los pasos de una casilla a otra los marca las posibilidades determinadas por el movimiento del caballo ⇒ dos movimientos en línea recta y uno a la derecha o la izquierda (lo que es lo mismo uno en línea recta y otro diagonal).



Como mucho 8 posibilidades de avanzar en cada paso

```
public class Caballo {
    int n; // tamaño del tablero
    int [ ][ ] tablero;
    boolean exito;
    int[ ] movX = {2,1,-1,-2, -2, -1, 1, 2};
    int[ ] movY = {1,2, 2, 1, -1, -2, -2, -1};

    public Caballo (int tam) {
        n = tam;
        tablero = new int [n][n];
        for (int i=0;i<n;i++)
            for (int j=0;j<n;j++) tablero[i][j] = 0;
    }
```

	2	7		
				8
3	6	1		
		4	9	
5	10			

Cuando no hay posibilidad de continuar con el movimiento hay que dar marcha atrás y continuar con otra alternativa

	2	7		
		10		8
3	6	1		
		4	9	
5	10			

1. Considerar un tablero de ajedrez de N x N escaques y un caballo situado en la esquina inferior izquierda. Dado el movimiento de esta pieza en el juego, encontrar una secuencia de saltos del caballo que pase una única vez por cada casilla del tablero, o determinar que dicha secuencia no existe.

```
//@pre: (1<=k) && (k<=n*n)  // k es la etapa, (x,y) donde está el caballo.
boolean VA(int k, int x, int y) {
        int orden = 0; //Recorre las 8 posibilidades desde (x,y)
        boolean exito = false; int u,v; // Nuevas coordenadas

        while ( (!exito) && (orden<movX.length) ) {  //calculamos a donde salta
                u = x + movX[orden]; v = y + movY[orden]; // posible salto
                if ((0<=u)&&(u<n)&&(0<=v)&&(v<n)&& (tablero[u][v]==0)) {
                        // dentro o previamente visitado
                        tablero[u][v] = k; //anotar opción
                        if (k<n*n) { //hemos acabado?
                                exito = VA(k+1,u,v);
                                if (!exito)
                                        tablero[u][v] = 0; //cancela
                        } else
                                exito = true;
                }
                orden++;
        }
        return exito;
}
```

```
// (x,y) son las coordenadas desde donde
// empieza a moverse el caballo.
//@pre: (0<=x)&&(x<n)&&(0<=y)&&(y<n)
// devuelve el tablero solución.
public int [ ][ ] vueltaAtras(int x, int y) {
        tablero[x][y] = 1;
        VA(2,x,y);
        return tablero;
}
} // fin de class Caballo
```

4. Dado un grafo $G(V, E)$, un coloreado del mismo es una asignación de colores (representados como números naturales 1; 2; …) a vértices, tal que si dos vértices están unidos por una arista entonces tienen distinto color. Se dice que un grafo es k-coloreable si puede colorearse con los colores 1; 2; …; k.

Suponemos que la entrada es una matriz [V][V] representación de la matriz de adyacencia del grafo, siendo V el número de vértices. También tendremos como entrada el entero k que es el número máximo de colores que se pueden usar.

Y como salida, podemos suponer un array C[V] que debe tener números del 1 al k, de forma que C[i] representa el color asignado al i-ésimo vértice. El código también debe devolver falso si el gráfico no se puede colorear con k colores.

El enfoque es asignar colores uno por uno a diferentes vértices, comenzando desde el vértice 0. Antes de asignar un color, hay que verificar si los vértices adyacentes tienen el mismo color o no. Si hay alguna asignación de color que no infringe las condiciones de adyacencia, se puede marcar la asignación de color actual como parte de la solución. Si no es posible asignar un color, se retrocede y devuelve falso. De esto podemos sacar nuestro algoritmo:

1. Creamos una función recursiva que toma el grafo, el índice actual, el número de vértices y el array de colores.
2. Si el índice actual es igual al número de vértices, hemos terminado y la configuración de colores es válida.
3. En caso contrario, vamos a asignar color al siguiente vértice (1 a k), para lo cual, para cada color asignado hay que verificar si la configuración es segura (es decir, verificar si los vértices adyacentes no tienen el mismo color)
4. Llamamos de forma recursiva a la función con el siguiente índice y número de vértices.
5. Si alguna función recursiva devuelve verdadero, rompemos el ciclo y devolvemos verdadero, sino, devolvemos falso.

4. Dado un grafo $G(V, E)$, un coloreado del mismo es una asignación de colores (representados como números naturales 1; 2; …) a vértices, tal que si dos vértices están unidos por una arista entonces tienen distinto color. Se dice que un grafo es k-coloreable si puede colorearse con los colores 1; 2; …; k.

```
public static void main(String args[]) {
        graphColoring(graph, k);
}

void graphColoring(int graph[][], int k) {
        color = new int[V];
        for (int i = 0; i < V; i++)  color[i] = 0;

        if (!graphColoringUtil(graph, k, color, 0)) {
            System.out.println("No existe solución");
        }

        printSolution(color); // Enseñamos la solución
}

boolean isSafe(int v, int graph[][], int color[], int c) {
        for (int i = 0; i < V; i++)
            if (graph[v][i] == 1 && c == color[i])
                return false;

        return true;
}
```

```
boolean graphColoringUtil(int graph[][], int k, int color[], int v) {
        if (v == V)  return true;

        for (int c = 1; c <= k; c++) {
            if (isSafe(v, graph, color, c)) {
                color[v] = c;

                if (graphColoringUtil(graph, k, color, v + 1))
                    return true;

                /* Si la asignación no ha llevado a una solución,
                   la eliminamos y probaremos un nuevo color*/
                color[v] = 0;
            }
        }

        /* Si no se ha asignado ningún color, no hay solución
           posible a encontrar en esta rama */
        return false;
}
```

5. Dado un grafo $G(V, E)$, un cliqué es un conjunto de vértices $S \subseteq V$ tal que para todo $v, w \in S$ hay una arista $(v, w) \in E$, i.e., todos los vértices de S están conectados entre si. Implementar un algoritmo que encuentre el mayor cliqué de un grafo.

Sea G = (V,E) un grafo, una función que calcula el cliqué mayor de G podría construirse como sigue.

Suponemos que existen unos tipos Vertice y Arista para representar los vértices y arcos del grafo.
La función MayorClique(V,E) itera por todos los vértices del grafo, y devuelve el mayor clique del grafo. Durante la búsqueda C contiene el mayor cliqué encontrado. Si se encuentra un cliqué mayor que el encontrado hasta ese momento, C se actualiza. La búsqueda la realiza el método MayorClique(C, W, E) que devuelve el mayor cliqué del grafo que contiene a C, extendiendo C con los nodos de W.

```
Set ⟨Vertice⟩ MayorClique (Set ⟨Vertice⟩ V, Set ⟨Arista⟩ E){
    Set ⟨Vertice⟩ C = ∅;

    for (v ∈ V) {
        Set ⟨Vertice⟩ W = V − {v};
        Set ⟨Vertice⟩ NClique = MayorClique ({v}, W, E);
        if (NClique.size() > C.size()) { C = Nclique; }
    }

    return C;
}
```

5. Dado un grafo $G(V, E)$, un cliqué es un conjunto de vértices $S \subseteq V$ tal que para todo $v, w \in S$ hay una arista $(v, w) \in E$, i.e., todos los vértices de S están conectados entre si. Implementar un algoritmo que encuentre el mayor cliqué de un grafo.

La función MayorClique(C,W,E) busca el mayor cliqué del grafo mediante backtracking. El conjunto C es el cliqué actual, W son los vértices del grafo que quedan por explorar, y E son los arcos del grafo inicial.

Esta función hace uso de la función booleana estaConectado(w, C, E) que comprueba si el vértice w está conectado a todos los vértices del cliqué actual C, mediante algún arco de E.

```
Set ⟨Vertice⟩ MayorClique (Set ⟨Vertice⟩ C, Set ⟨Vertice⟩ W, Set ⟨Arista⟩ E){
    if (W.size() == 0) { return C; }

    Set ⟨Vertice⟩ MClique = C;
    for (w ∈ W) {
        if (estaConectado (w, C, E)) {
            Set ⟨Vertice⟩ C' = C + {w};
            Set ⟨Vertice⟩ W' = W - {w};
            Set ⟨Vertice⟩ Clique = MayorClique (C', W', E);
            if (Clique.size() > MClique.size()) { MClique = Clique; }
        }
    }
    return Mclique;
}
```

6. Dada la matriz de adyacencia de un grafo no dirigido G, encontrar (o determinar que no existe) un camino hamiltoniano (un camino que pasa una única vez por cada vértice del grafo).

Sea Como en el ejercicio anterior, suponemos que existen dos tipos de datos Vertice y Arista. Además, suponemos que los vértices están numerados, es decir, el conjunto de vértices del grafo es V = {0, · · · , n − 1}.

A diferencia de un ciclo hamiltoniano, en un camino el último y el primer nodo del camino no tienen por qué estar unidos en el grafo. Por lo tanto, en esta aplicación hay que probar con todos los nodos del grafo, hasta que se encuentre un camino, o bien hasta que se comprueba que no hay ningún camino hamiltoniano en el grafo. El método camHam(int[][] m) devuelve un camino hamiltoniano si existe en el grafo definido. El camino se devuelve como el conjunto de aristas (o arcos) que lo forman.

El método camHam(int[][] m) llama al método camHam(v,W,C,m) donde v es el nodo inicial del camino, W son los vértices que faltan por visitar, C es el camino construido hasta el momento (el camino vacío inicialmente), y m es la matriz de adyacencias. Este método es el que utiliza la técnica de backtracking para encontrar el camino.

Si en alguna llamada devuelve un camino no vacío, éste es un camino hamiltoniano, y la búsqueda se detiene. Si no se ha encontrado ningún camino hamiltoniano, se devuelve vacío.

```
Set ⟨Arista⟩ camHam (int[][] m){
    Set ⟨Vertice⟩ V = {0, · · · , n − 1};

    for (v ∈ V) {
        Set ⟨Vertice⟩ W = V − {v};
        Set ⟨Arista⟩ C = ∅;
        Set ⟨Arista⟩ Cam = camHam (v, W, C, m);
        if (Cam.size() > 0) { return Cam; }
    }

    return ∅;
}
```

6. Dada la matriz de adyacencia de un grafo no dirigido G, encontrar (o determinar que no existe) un camino hamiltoniano (un camino que pasa una única vez por cada vértice del grafo).

Como se ha comentado arriba, una llamada al método camHam(v,W,C,m) indica que v es el nodo actual del camino, que va a extenderse posiblemente con una nuevo arco a otro nodo no visitado. El conjunto W es el conjunto de los nodos no visitados aún, C es el camino que se ha construido hasta ahora, y m es la matriz de adyacencias original.

El método inicialmente comprueba si se han visitado todos los vértices, en cuyo caso se devuelve el camino encontrado C. En otro caso, se prueba con todos los vértices no visitados para ver si es posible construir un camino hamiltoniano a partir de ellos. En cuanto se encuentra un camino, la búsqueda termina. Si ningún vértice lleva a un camino hamiltoniano se devuelve el conjunto vacío.

```
Set ⟨Arista⟩ camHam (Vertice v, Set ⟨Vertice⟩ W, Set ⟨Arista⟩ C, int[][] m){
    if (W.size() == 0) { return C; }
    else {
        for (w ∈ W) {
            if (estaConectado (w, v, m)) {
                Set ⟨Arista⟩ C' = C + {w};
                Set ⟨Vertice⟩ W' = W − {w};
                Set ⟨Arista⟩ H = camHam (w, W', C', m);
                if (H.size() > 0) { return H; }
            }
        }
        return ∅;
    }
}
```

7. Un constructor dispone de 3 solares (que llamaremos A, B y C) y desea construir 3 edificios distintos: un banco, un hotel y un colegio. El coste de construir cada edificio depende del solar en donde se realice, y viene dado por la siguiente tabla (en millones de euros):

	A	B	C
Banco	1	2	8
Hotel	4	5	3
Colegio	1	7	9

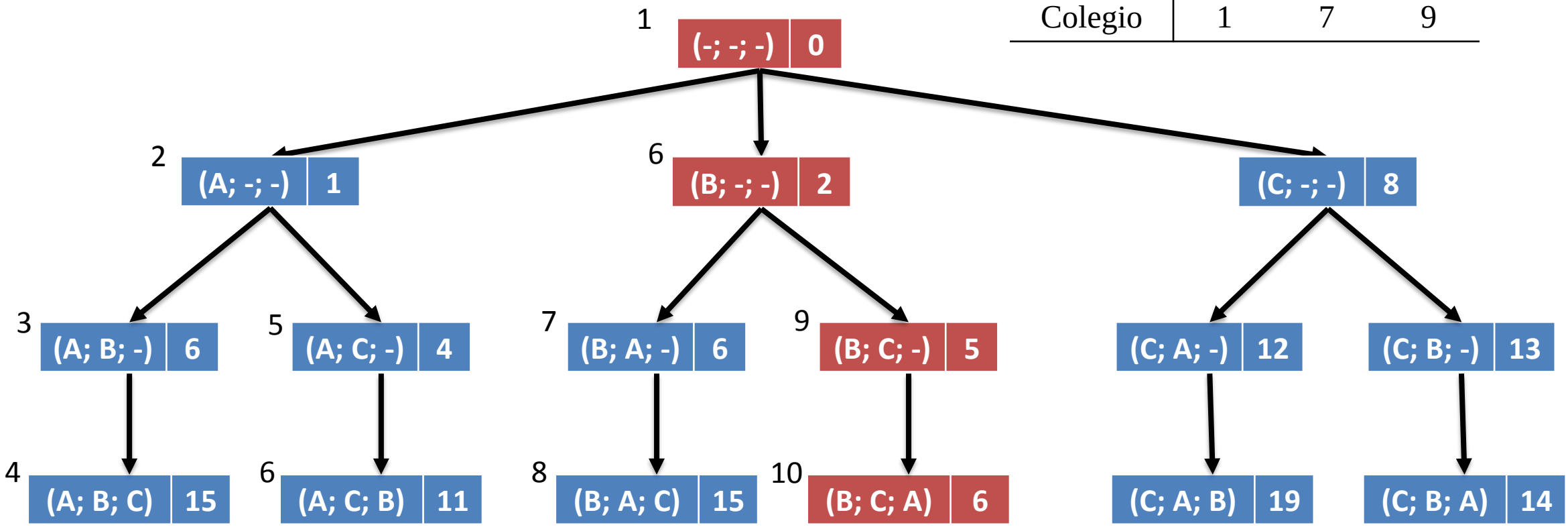
El constructor desea construir los tres edificios, cada uno en un solar, pero con el mínimo desembolso económico.

a) Implementar un algoritmo de backtracking para resolver este problema.

b) Dibujar el árbol de búsqueda para la instancia anterior. ¿Cuál es el número de nodos que se visitan para encontrar la solución al problema?

b) Dibujar el árbol de búsqueda para la instancia anterior. ¿Cuál es el número de nodos que se visitan para encontrar la solución al problema?

	A	B	C
Banco	1	2	8
Hotel	4	5	3
Colegio	1	7	9



a) Implementar un algoritmo de backtracking para resolver este problema.

```
int[] alg (int[][] tabla) {
	int[] sol = new int[3];
	int[] msol = new int[3];
	int mcal = Integer.MAX_VALUE;

	msol = alg_back(tabla, 0, sol, msol, mcal);
	return msol;
}

int suma (int[][] tabla, int[] sol) {
	int res = 0;
	for (int i = 0; i < sol.length; i++) {
		res = res + tabla[i][sol[i]];
	}
	return res;
}

List<Integer> posibilidades (int[] sol, int i) {
	List<String> list = Arrays.asList(0, 1, 2);
	for (int j = 0; j < i; j++) {   list.remove(sol[j]);  }
	return list;
}
```

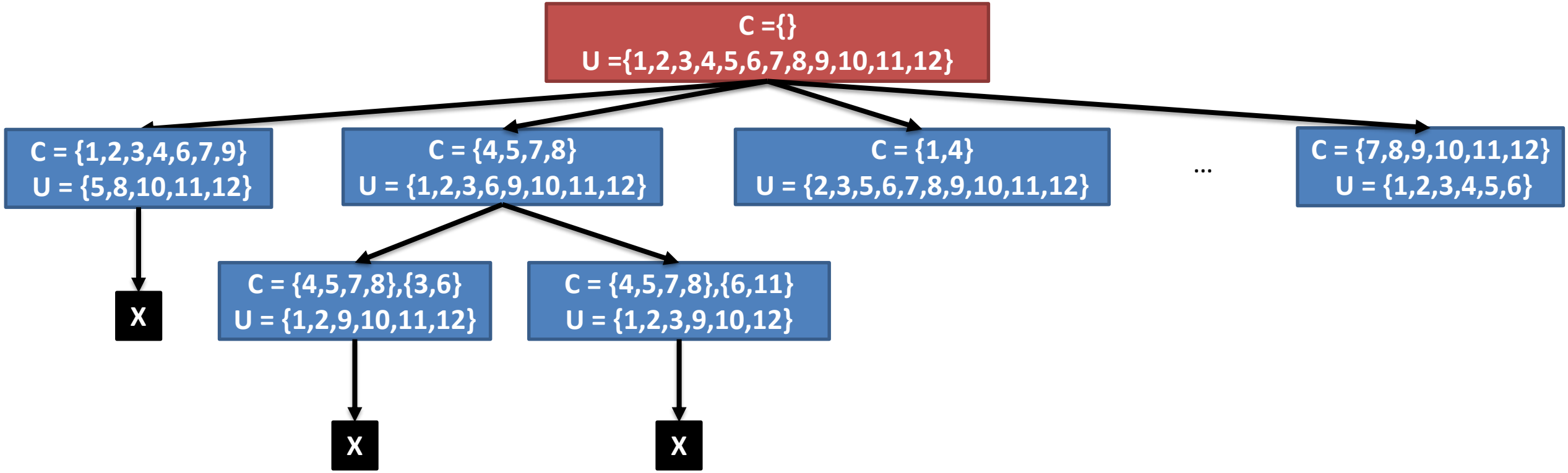
```
int[] alg_back (int[][] tabla, int i, int[] sol, int[] msol, int mcal) {

	if (i == sol.length) {
		int cal = suma(sol, tabla);
		if (cal < mcal) return sol;
		else return msol;
	}
	else {
		List<Integer> opciones = posibilidades(sol, i);

		for (int j = 0; j < opciones.length; j++) {
			int[] sol_copy = sol.clone();
			sol_copy[i] = opciones.get(j);

			int[] otra = alg_back(tabla, i+1, sol_copy, msol, mcal);
			if (suma(otra, tabla) < mcal) {
				msol = otra;
				mcal = suma(otra, tabla)
			}
		}
		return msol;
	}
}
```

1. Sea U=$\{1 \ldots n\}$ y $S = \{S_1, \ldots, S_m\}$ una colección de subconjuntos de U (i.e. $S_i \subseteq U$ para $1 \leq i \leq m$). Una partición de U es una colección $C \subseteq S$ tal que todos los subconjuntos en C son mutuamente disjuntos y su unión es U.

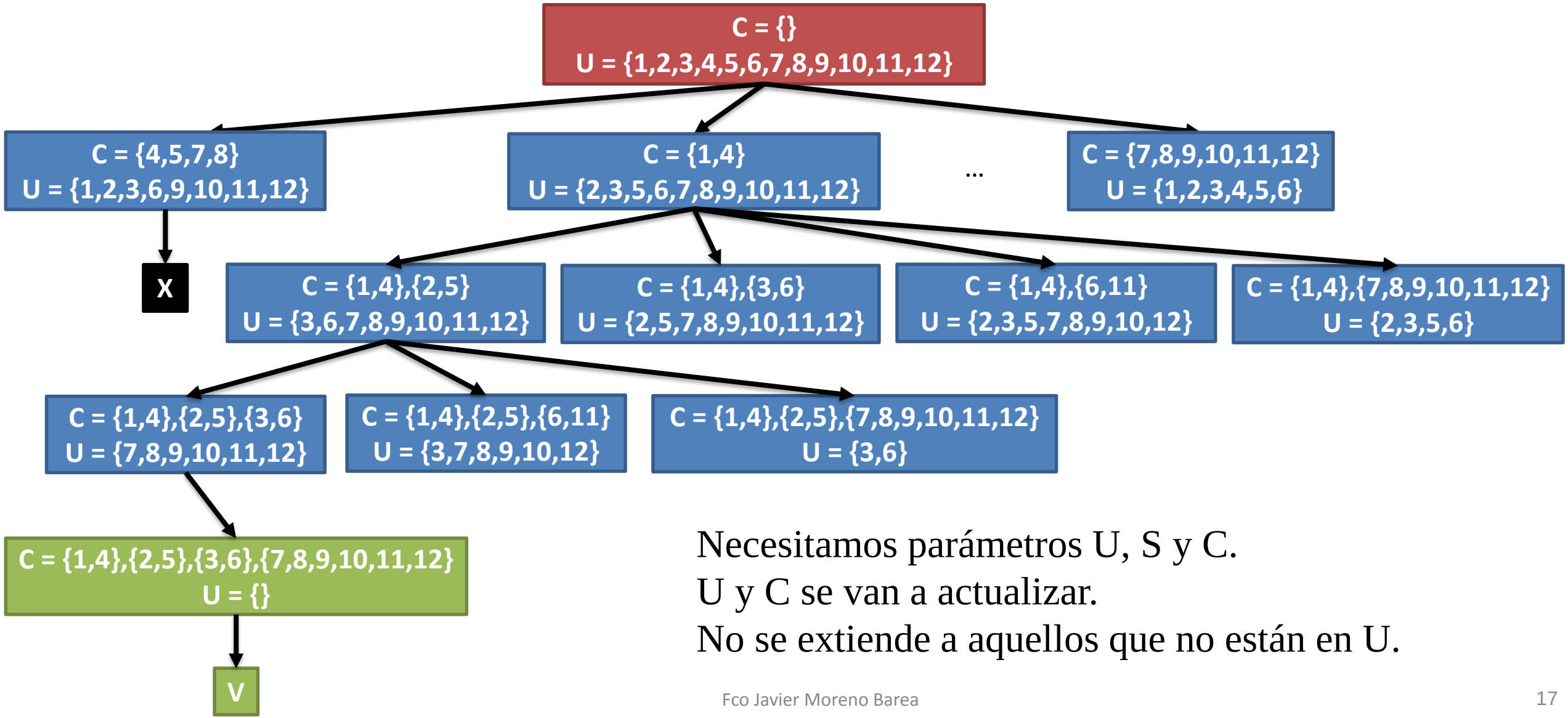
   a) Diseñar un algoritmo de Vuelta Atrás tal que, dados U y S, encuentre una partición o determine si esta no existe. Indica los parámetros de la primera llamada al algoritmo diseñado.

   b) Escribe el árbol implícito de expansión para la siguiente instancia del problema: $U = \{1 \ldots 12\}$ y $S = \{\{1,2,3,4,6,7,9\}, \{4,5,7,8\}, \{1,4\}, \{2,5\}, \{3,6\}, \{6,11\},$

b) Escribe el árbol implícito de expansión para la siguiente instancia del problema:
$U = \{1 \dots 12\}$ y $S = \{\{1,2,3,4,6,7,9\},\{4,5,7,8\},\{1,4\},\{2,5\},\{3,6\},\{6,11\},$



b) Escribe el árbol implícito de expansión para la siguiente instancia del problema:
$U = \{1 \dots 12\}$ y $S = \{\{1,2,3,4,6,7,9\},\{4,5,7,8\},\{1,4\},\{2,5\},\{3,6\},\{6,11\},$



Necesitamos parámetros U, S y C.
U y C se van a actualizar.
No se extiende a aquellos que no están en U.

a) Diseñar un algoritmo de Vuelta Atrás tal que, dados U y S, encuentre una partición o determine si esta no existe. Indica los parámetros de la primera llamada al algoritmo diseñado.

```
boolean alg_back (Set<Set<Integer>> S, Set<Set<Integer>> C,
                  Set<Integer> U ) {
    if (U.isEmpty()) {
        return true;
    } else {
        Set<Set<Integer>> setCont = contSet(S, U);
        for (cont ∈ setCont) {
            C.add(cont);
            Set<Integer> aux = deleteRes(cont, U);
            if (alg_back(S, C, aux)) {
                return true;
            } else {
                C.remove(cont);
            }
        }
        return false;
    }
}
```

```
public Set<Set<Integer>> C = new Set<>();

void alg (Set<Set<Integer>> S, Set<Integer> U ) {
    boolean res = alg_back(S, C, U);

    if (res) { print("He encontrado conjunto " + C); }
    else {   print("No se ha encontrado solución"); }
}

Set<Set<Integer>> contSet (Set<Set<Integer>> S,
                           Set<Integer> U) {
    Set<Set<Integer>> C = new Set<>();
    for (s ∈ S) {  if (element(s) ∈ U)) {  C.add(s);  } }
    return C;
}

Set<Integer> deleteRes (Set<Integer> cont, Set<Integer> U) {
    Set<Integer> aux = new Set<>(U);
    for (r ∈ U) {  if (cont.contains(r)) {  aux.remove(r);  } }
    return aux;
}
```