Ejercicio 2

La sucesión de **Jacobsthal-Lucas** es la secuencia de números enteros grandes P(n) definida por los siguientes valores iniciales

$$P(0) = 2,\ P(1) = 1$$

y la siguiente relación de recurrencia

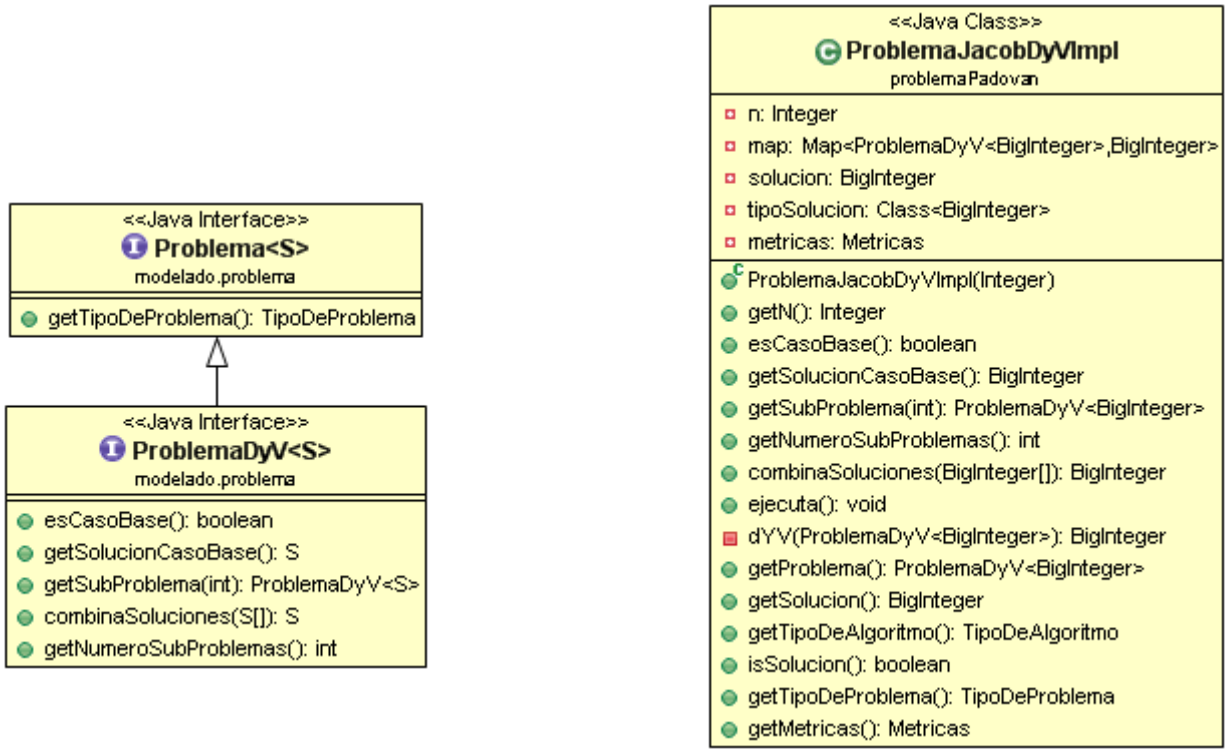
$$P(n) = P(n-1) + 2*P(n-2)$$

Los primeros valores de P(n) son: 2, 1, 5, 7, 17, 31, 65, 127, 257, 511, ...

Se pide:

a) Rellene la Ficha adjunta (que deberá entregar) siguiendo la información del problema de Jacobsthal-Lucas detallado anteriormente. Debe decidir cuál es la mejor opción para la técnica de Divide y Vencerás en este problema concreto: ¿con o sin memoria? Justifíquelo brevemente.

b) Implemente los métodos pertenecientes a la clase `ProblemaJacobDyVImpl`: para ello, ayúdese de los diagramas UML proporcionados a continuación:



a. `private BigInteger dYV(ProblemaDyV<BigInteger> p)`. Resuelve la técnica Divide y Vencerás.
b. `public boolean esCasoBase()`. Indica si el problema que se está resolviendo es tan simple que no necesita recursión para su resolución.
c. `public BigInteger getSolucionCasoBase()`. Devuelve la solución al problema, resuelto de forma directa.
d. `public int getNumeroSubProblemas()`. Indica el número de subproblemas.
e. `public ProblemaDyV<BigInteger> getSubproblema(int i)`. Devuelve el subproblema iésimo en el que se descompone el problema que se está resolviendo.
f. `public BigInteger combinaSoluciones (BigInteger[] soluciones)`. Combina cada solución de la lista de soluciones de manera que juntas resuelvan el problema completo. Puede utilizar las operaciones de BigInteger: *add(BigInteger a):BigInteger* y *multiply(BigInteger a):BigInteger*.

Ayuda: algunas operaciones de BigInteger son: BigInteger add(BigInteger val) – BigInteger multiply(BigInteger val) – BigInteger div(BigInteger val)

Tiempo estimado: 45 min Puntuación: 4 puntos

<table>
<tr><th colspan="2">Ficha</th></tr>
<tr><th colspan="2">Sucesión de Jacobsthal-Lucas</th></tr>
<tr><td colspan="2">Técnica: // Rellenar

Breve justificación (con o sin memoria): // Rellenar</td></tr>
<tr><td colspan="2">Representación: Un objeto por problema</td></tr>
<tr><td>Tamaño: N</td><td></td></tr>
<tr><td>Propiedades Compartidas</td><td>// Rellenar</td></tr>
<tr><td>Propiedades Individuales</td><td>// Rellenar</td></tr>
<tr><td colspan="2">Solución: BigInteger</td></tr>
<tr><td colspan="2">// Rellenar</td></tr>
<tr><td>Complejidad</td><td>// Rellenar</td></tr>
</table>

Solución:

a) **(2 puntos)**

Ficha	
Sucesión de Jacobsthal-Lucas	
Técnica: // ***Rellenar*** **Divide y Vencerás con Memoria** *Breve justificación (con o sin memoria):* // ***Rellenar*** **Al dividir el problema en subproblemas ((n-2) y (n-3)), aparecen repetidos los problemas más de una vez. Guardando en memoria las operaciones ya realizadas disminuye el tiempo computacional y el número de operaciones a realizar.**	
Representación: ***Un objeto por problema***	
Tamaño: ***N***	
Propiedades Compartidas	// ***Rellenar*** **M, Memoria**
Propiedades Individuales	// ***Rellenar*** **N, Integer no negativo**
Solución: ***BigInteger***	
// ***Rellenar*** $P\ n\ = \begin{cases} M\ n & ,si\ n \in M \\ 2 & ,si\ n = 0 \\ 1 & ,si\ n = 1 \\ P\ n-1\ +\ 2 * P\ n-2\ & ,si\ n > 1 \end{cases}$	
Complejidad	// ***Rellenar*** Si g(n) = k Θ(|I|) // también es válido Θ(n)

b)

a. `private BigInteger dYV(ProblemaDyV<BigInteger> p)`. Resuelve la técnica Divide y Vencerás. **(2 puntos)**

```
private S dYV(ProblemaDyV<S> p) {
	S s;
	if (map.containsKey(p)) {
		s = map.get(p);
	} else if (p.esCasoBase()) {
		s = p.getSolucionCasoBase();
		map.put(p, s);
	} else {
		int numeroDeSubProblemas = p.getNumeroSubProblemas();
		S[] soluciones = Utiles.creaArray(numeroDeSubProblemas);
		for(int i = 0; i < numeroDeSubProblemas; i++){
			ProblemaDyV<S> pr = p.getSubProblema(i);
			s = dYV(pr);
			soluciones[i]=s;
			map.put(pr, s);
		}
		s = p.combinaSoluciones(soluciones);
		map.put(p, s);
	}
	return s;
}
```

b. `public boolean esCasoBase()`. Indica si el problema que se está resolviendo es tan simple que no necesita recursión para su resolución. **(1 puntos)**

```
public boolean esCasoBase() {
		return getN() <= 1;
}
```

c. `public BigInteger getSolucionCasoBase()`. Devuelve la solución al problema, resuelto de forma directa. **(1 puntos)**

```
public BigInteger getSolucionCasoBase() {
	BigInteger s;
	if(getN().equals(0))
		s = new BigInteger("2");
	else
		s = BigInteger.ONE;
		return s;
}
```

d. `public int getNumeroSubProblemas()`. Indica el número de subproblemas. **(0,5 puntos)**

```
public int getNumeroSubProblemas() {
		return 2;
}
```

e. `public ProblemaDyV<BigInteger> getSubproblema(int i)`. Devuelve el subproblema iésimo en el que se descompone el problema que se está resolviendo. **(1,5 puntos)**

```
public ProblemaDyV<BigInteger> getSubProblema(int i) {
		return new ProblemaJacobDyVImpl(getN()-i-1);
}
```

f. `public BigInteger combinaSoluciones (BigInteger[] soluciones)`. Combina cada solución de la lista de soluciones de manera que juntas resuelvan el problema completo. **(2 puntos)**

```
public BigInteger combinaSoluciones(BigInteger[] soluciones) {
            BigInteger sol =
soluciones[0].add(soluciones[1].multiply(new BigInteger("2")));
            return sol;
}
```