EJERCICIO_3

El alcalde de un determinado pueblo dispone de un presupuesto para realizar las diversas actividades que estaban contempladas en su programa electoral. Con el fin de maximizar el número de habitantes que se vean beneficiados por esas medidas, los asesores pretenden diseñar un algoritmo de vuelta atrás para escoger aquellas actividades que hagan máximo el número de habitantes que se verían afectados por dicha actividad.

Ejemplo: La siguiente tabla muestra un ejemplo de datos de entrada. Para un presupuesto de 2500 unidades la mejor opción es realizar las actividades 1 y 4, que agruparían 60 habitantes y tendría un coste de 2500 unidades.

Actividad	Habitantes afectados	Coste de la actividad
1	15	1100
2	25	1300
3	25	1200
4	45	1400

Presupuesto: 2500

El objetivo es encontrar, a través de un algoritmo de backtracking, que actividades que deben realizarse para alcanzar al máximo número de habitantes para un presupuesto total concreto.

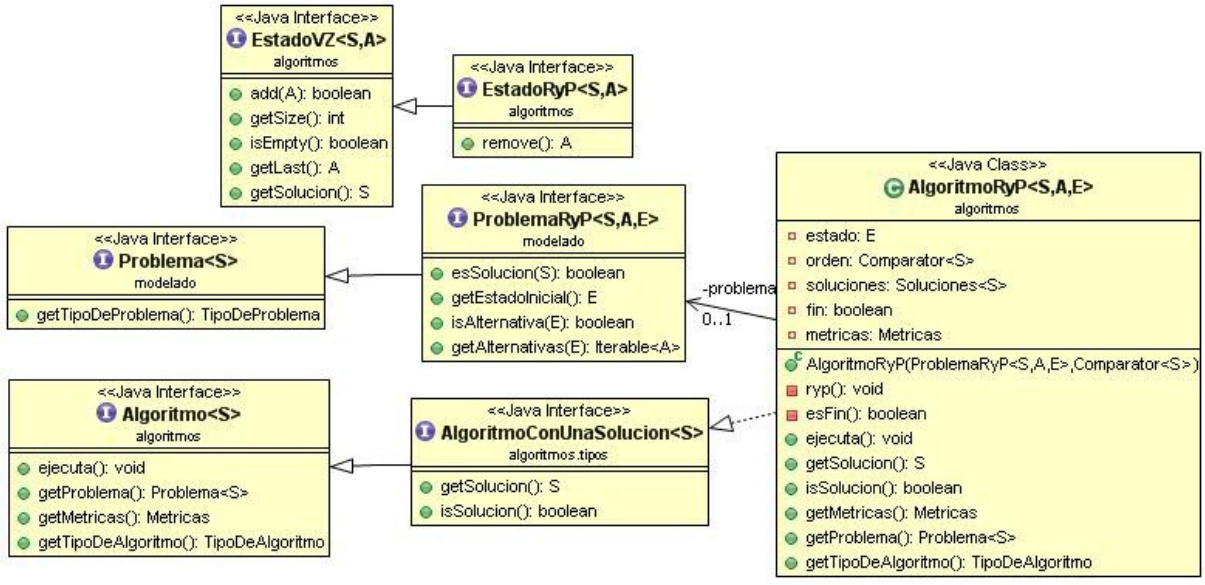
Para modelar el problema, se supondrá que existen *m* actividades (a_1, a_2,..., a_m). El tipo **Actividad** permite almacenar toda la información sobre una actividad. Los métodos más importantes son:

- Integer getHabitantes(): Devuelve el número de habitantes afectados por la actividad.
- Integer getCoste() : Devuelve el coste asociado a la actividad

Tenga en cuenta además que:

- Para calcular los habitantes afectados por las actividades simplemente se deben sumar los habitantes afectados por cada actividad, es decir, no hace falta contemplar si varias actividades afectan al mismo habitante.
- Las actividades no se pueden fraccionar.

El problema se modelará a través de la clase **ProblemaActividadesImpl**, que implementa la interfaz `ProblemaRyP<Integer, Boolean, EstadoActividades>`. Esta clase almacena el ***presupuesto inicial*** (atributo `presupuesto` de tipo `Integer`) que es el tope máximo de gasto total.



La clase **EstadoActividadesImpl** (implementación de la interfaz EstadoActividades) permitirá almacenar toda la información del estado del problema (ver ficha adjunta). Los atributos serán:

- Integer **actividadesProcesadas**: Contador de actividades para las que ya se ha decidido si recibirán o no subvención.
- List<Actividad> **actividadesAtendidas**: Actividades para las que ya se ha decidido realizar una subvención.
- Integer **presupuestoDisponible**: Parte del presupuesto que aún no ha sido utilizado.
- Integer **habitantesTotal**: Habitantes cubiertos por las actividades subvencionadas.
- List<Actividad> **listaActividades**: Lista que almacena todas las actividades que existen en el problema.

SE PIDE:

a) Escribir el código completo del método *ryp*() de la clase AlgoritmoRyP<Integer, Boolean, EstadoActividades>.

b) Escribir el método *isAlternativa*(EstadoActividad e) de la clase ProblemaActividadesImpl.

c) Escribir el método *getAlternativas*() de la clase ProblemaActividadesImpl. Tenga en cuenta que tal cómo se puede ver en la ficha proporcionada, las alternativas serán valores lógicos que indicarán si la actividad recibirá subvención o no.

d) Escribir el código completo de los métodos *add*(Boolean) y *remove*() de la clase EstadoActividadesImpl, teniendo en cuenta la ficha proporcionada.

BT	
Problema Ayuntamiento	
Técnica: Backtracking	
Tamaño: n = listaActividades.size() – actividadesProcesadas	
Propiedades Compartidas	***presupuesto**: Integer*
Propiedades del Estado	***actividadesProcesadas:** Integer* ***actividadesAtendidas**: List<Actividad>* ***presupuestoDisponible**: Integer* ***habitantesTotal**: Integer* ***listaActividades**: List<Actividad>*
Solución: Integer	
Estado inicial: *actividadesProcesadas: 0* *actividadesAtendidas: { }* *presupuestoDisponible: PresupuestoTotal* *habitantesTotal: 0* *listaActividades = ActividadesPrevistas*	

```
	private void ryp() {
		S solucion = estado.getSolucion();
		if (problema.esSolucion(solucion)) {
		   soluciones.add(solucion);
		    fin = esFin();
		}
		if (!fin && problema.isAlternativa(estado)) {
		   Iterable<A> it = problema.getAlternativas(estado);
		   for (A alternativa: it){
				estado.add(alternativa);
				ryp();
				estado.remove();
			}
		}
	}

	public boolean add(Boolean e) {
		if (e){
			Actividad a =
listaActividades.get(actividadesProcesadas);
			actividadesAtendidas.add(a);
			presupuestoDisponible -= a.getCoste();
			habitantesTotal += a.getHabitantes();
		}
		actividadesProcesadas++;
		return true;
	}

	public Boolean remove() {
		actividadesProcesadas--;
		Actividad a = listaActividades.get(actividadesProcesadas);
		Boolean ret = actividadesAtendidas.contains(a);
		if (ret){
			actividadesAtendidas.remove(a);
			presupuestoDisponible += a.getCoste();
			habitantesTotal -= a.getHabitantes();
		}
		return ret;
	}

	public Iterable<Boolean> getAlternativas(EstadoActividades e) {
		List<Boolean> it = Lists.newLinkedList();
		it.add(false);
		if (e.getPresupuestoDisponible() >=
	e.getListaActividades().get(e.getActividadesProcesadas()).getCoste())
			it.add(true);
		return it;
	}

	public boolean isAlternativa(EstadoActividades e) {
		return e.getActividadesProcesadas() <
		e.getListaActividades().size() &&
		e.getPresupuestoDisponible() > 0;
	}
```