Ejercicio 3 – Algoritmos Genéticos

Se pretende renovar el PC de los N miembros de un departamento. Para ello, se dispone de M configuraciones distintas de PCs que pueden usarse, cada una con un coste concreto. Hay varios PCs disponibles para cada configuración. Además, cada miembro tiene unas necesidades mínimas que su PC (procesador, memoria, etc.) ha de satisfacer.
El objetivo será asignar un PC a cada miembro de manera que se satisfagan sus necesidades y se minimice el coste de la asignación.

Ejemplo:

Miembro	Proc.	Ram.	Disco
1	2GHz	4GB	500GB
2	2GHz	4GB	1000GB
3	2,4GHz	8GB	250GB
4	2GHz	16GB	500GB

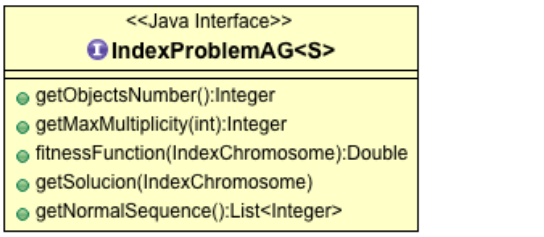
Config.	Proc.	Ram.	Disco	Precio	Dispo.
A	2,4GHz	4GB	1000GB	500 €	2
B	2GHz	16GB	500GB	750 €	2
C	3,2GHz	12GB	250GB	900 €	2

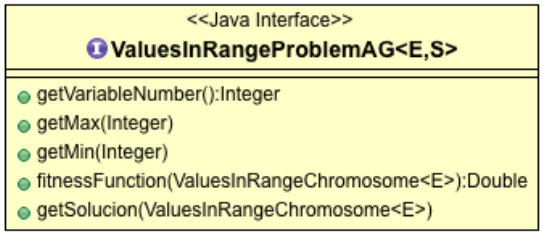
Solución 1	Solución 2
1→A	1→B
2→A	2→A
3→C	3→A
4→B	4→B
Coste: 2650€	Coste: 2500€ **Óptima**

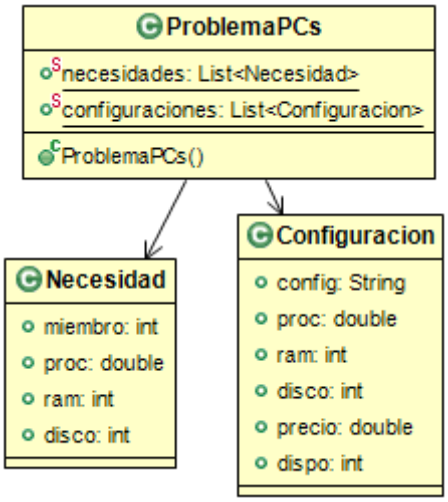
SE PIDE:

1. Justificar qué cromosoma es el más apropiado.
2. Implemente la clase ProblemaPCsAG.

NOTA: Suponga que los datos del problema, contenidos en un fichero, ya han sido cargados en *ProblemaPCs*.







SOLUCIÓN

1. Cromosoma Range donde cada necesidad es representada por un gen cuyo valor indica la configuración a usar.
2. Implementación:

```
public class ProblemaPCsAG implements ValuesInRangeProblemAG<Integer,
List<String>> {
public Integer getVariableNumber() {
	return ProblemaPCs.necesidades.size();
}
public Integer getMax(Integer i) {
	return ProblemaPCs.configuraciones.size()-1;
}
public Integer getMin(Integer i) {
	return 0;
}
public List<String> getSolucion(ValuesInRangeChromosome<Integer> cr) {
	return cr.decode().stream()
		.map(conf->ProblemaPCs.configuraciones.get(conf).config)
		.collect(Collectors.toList());
}
public Double fitnessFunction(ValuesInRangeChromosome<Integer> cr) {
	double value= calculaValor(cr.decode());
	int incumplido= calculaDesviaciones(cr.decode());
	return -value - incumplido*incumplido*10000;
}
private int calculaDesviaciones(List<Integer> decode) {
	int necesidadNoSatisfechas=(int) IntStream.range(0,
	ProblemaPCs.necesidades.size())
		.filter(nec-> !esSatisfecha(ProblemaPCs.necesidades.get(nec),
		ProblemaPCs.configuraciones.get(decode.get(nec))))
		.count();
	Map<String, Integer> repes= decode.stream()
		.map(conf-> ProblemaPCs.configuraciones.get(conf).config)
		.collect(Collectors.groupingBy(conf->conf,
		Collectos.summingInt(conf->1)));
	int confExcedidas=ProblemaPCs.configuraciones.stream()
		.filter(conf-> repes.get(conf.config)!=null)
		.mapToInt(conf-> repes.get(conf.config)-conf.dispo)
		.filter(diff-> diff>0)
		.sum();
	return confExcedidas+necesidadNoSatisfechas;
}
private boolean esSatisfecha(Necesidad necesidad, Configuracion
configuracion) {
	return necesidad.disco<=configuracion.disco &&
		necesidad.proc<= configuracion.proc &&
		necesidad.ram<= configuracion.ram;
}
private double calculaValor(List<Integer> decode) {
	return decode.stream()
		.mapToDouble(conf->ProblemaPCs.configuraciones.get(conf).precio)
		.sum();
}
```