Ejercicio 2 (II, ITIG)

Se dispone de un fondo de inversión que dispone de una cantidad de dinero M para invertir en acciones. Al invertir la cantidad X en una acción el fondo espera recibir un cierto beneficio o interés. Se desea diseñar un algoritmo que intente repartir la cantidad M entre N acciones distintas de forma que se obtenga el máximo beneficio esperado posible. Para poder resolver el problema se dispone de una función beneficio(i,x) que devuelve el beneficio o interés que obtiene el fondo al invertir la cantidad x en la acción i. Para cualquier acción i se cumple que beneficio(i,0) = 0 y que si x > y, entonces beneficio(i,x) >= beneficio(i,y).

Se dispone de los siguientes tipos (en notación compacta):

Tipo Acción: $b_j^{k,f}$
j: Integer **// Identifica a la acción**
k: Integer **// Cantidad a invertir en la acción**
f(Integer x): Integer **// Beneficio al invertir la // cantidad x en la acción**

Tipo SolucionInversiones: $s^{lacciones,\ ben}$
lacciones: Lista<Accion> **// Lista de inversiones**
ben: Integer **// Beneficio de la solución**

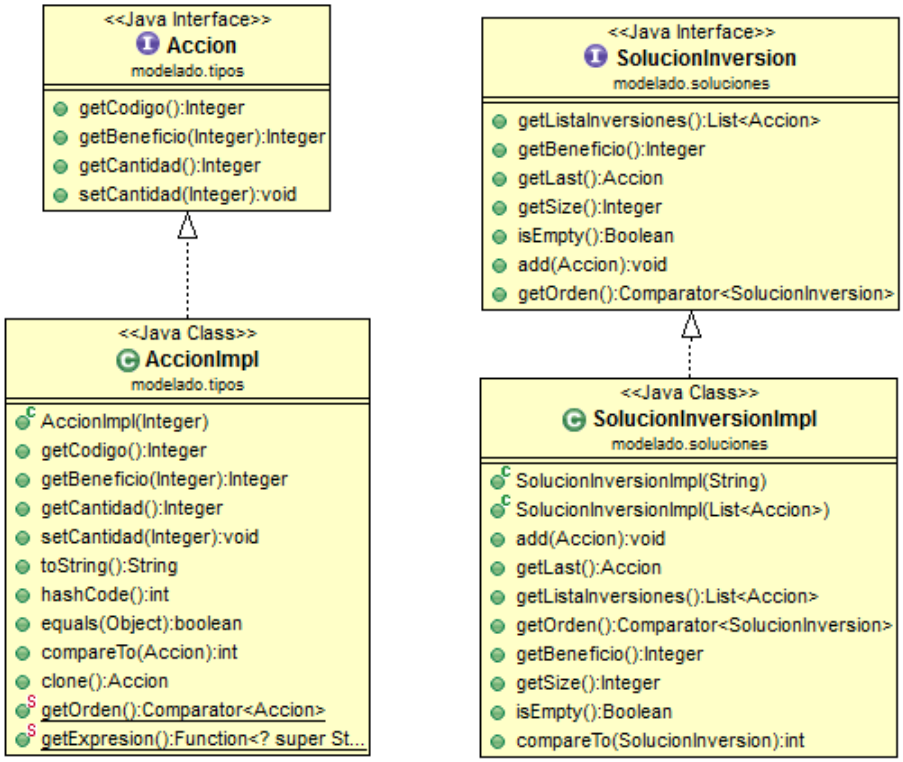
Se desea resolver el problema usando Programación Dinámica, de acuerdo a la siguiente ficha, para lo que generalizamos el mismo mediante las propiedades: *C* (Entero, consultable), cantidad a invertir, y *J* (Entero, consultable), que representa al subconjunto de las acciones con índices en el intervalo [0, *J*). Así pues, en el problema generalizado se trata de obtener el máximo beneficio al invertir la cantidad C entre las acciones 0..J.

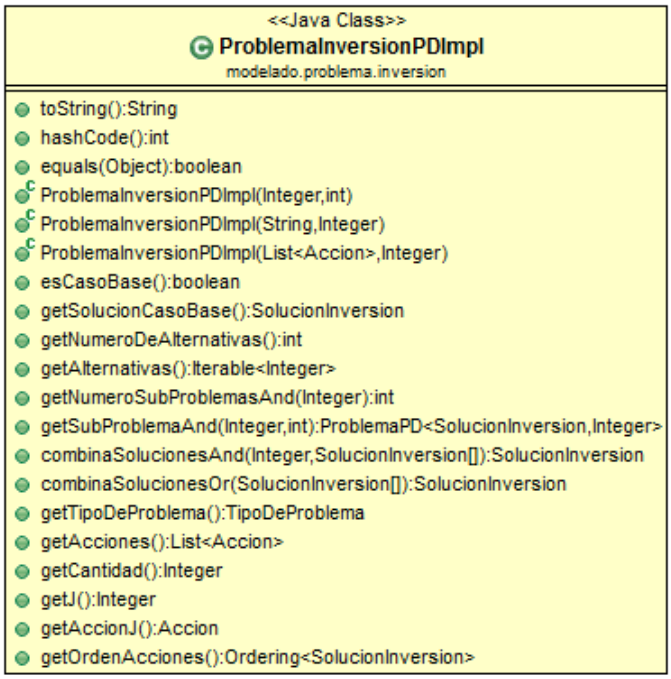
Ficha	
Inversiones Acciones	
Técnica: Programación Dinámica	
Representación: Cada problema un objeto	
Tamaño: J	
Propiedades Compartidas	*Acciones, List <Accion>* *M, entero no negativo*
Propiedades Individuales	*C, entero no negativo* *J, entero en [0,Acciones.Size)*
Solución: SolucionAcciones	
Objetivo: Encontrar $s^{lacciones,ben}$ tal que ben tenga el mayor valor posible	
Alternativas: $A_{c,j} = 0,1,\dots,C$	
$pib\ M, Acciones$ = *ib(M,Acciones.Size-1)* $ib\ c,j = \begin{cases} s^{\phi,0}, & c = 0 \\ s^{\ b_0^c\ ,b_0.f(c)}, & j = 0 \\ cO_{a_i \in A_{c,j}}\ cA(a_i, ib(c-a_i, j-1))\ , & c > 0, j > 0 \end{cases}$	
$cA(a_i, s)$: Si $a_i > 0$, añade la inversión de la cantidad a_i en la acción b_j a la solución s	
$cO_{a_i \in A_{c,j}}(s_{a_i}^{l,ben})$: Elige la solución con mayor valor de *ben*	
Complejidad	*O(M^2·Acciones.Size)*

SE PIDE:

a) Escribir el código del método **pD(ProblemaPD<S,A> p)** de la clase AlgoritmoProgramacionDinamica<S,A> .
b) Suponiendo definida la clase EnteroIterable, implementar de la clase ProblemaInversionPDImpl los métodos:
   - public boolean esCasoBase()
   - public int getNumeroDeAlternativas()

- public Iterable<Integer> getAlternativas()
- public int getNumeroSubProblemasAnd(Integer a)
- public ProblemaPD<SolucionInversion, Integer> getSubProblemaAnd(Integer a, int i)
- public SolucionInversion combinaSolucionesAnd(Integer a, SolucionInversion... ls)





Tiempo estimado: 45 min. Puntuación: 4 puntos

Solución

a) Escribir el código del método **pD(ProblemaPD<S,A> p)** de la clase AlgoritmoProgramacionDinamica<S,A> . **(0.75 puntos)**

```
private S pD(ProblemaPD<S, A> p) {
  S s = null;

  if(map.containsKey(p)) {
    s = map.get(p);
  }
  else if(p.esCasoBase()) {
    s = p.getSolucionCasoBase();
    map.put(p, s);
  }
  else {
    int numeroDeAlternativas = p.getNumeroDeAlternativas();
    S[] solucionesOr = Utiles.creaArray(tipoSolucion, numeroDeAlternativas);
    int j = 0;

    for(A a: p.getAlternativas()) {
      int numeroDeSubproblemas = p.getNumeroSubProblemasAnd(a);
      S[] solucionesAnd = Utiles.creaArray(tipoSolucion,
                                           numeroDeSubproblemas);

      for(int i = 0; i < numeroDeSubproblemas; i++) {
        ProblemaPD<S, A> pr = p.getSubProblemaAnd(a, i);
        s = pD(pr);
        solucionesAnd[i] = s;
      }

      solucionesOr[j++] = p.combinaSolucionesAnd(a, solucionesAnd);
    }

    s = p.combinaSolucionesOr(solucionesOr);
    map.put(p, s);
  }

  return s;
}
```

b) Suponiendo definida la clase EnteroIterable, implementar de la clase ProblemaInversionPDImpl los métodos:

- public boolean esCasoBase() **(0.5 puntos)**

```
public boolean esCasoBase() {
  return (getCantidad() == 0) || (getJ() == 0);
}
```

- public int getNumeroDeAlternativas() **(0.5 puntos)**

```
public int getNumeroDeAlternativas() {
  return (getCantidad() + 1);
}
```

- public Iterable<Integer> getAlternativas() **(0.25 puntos)**

```
public Iterable<Integer> getAlternativas() {
  return new EnteroIterable();
}
```

- public int getNumeroSubProblemasAnd(Integer a) **(0.25 puntos)**

```
public int getNumeroSubProblemasAnd(Integer a) {
  return 1;
}
```

- public ProblemaPD<SolucionInversion, Integer> getSubProblemaAnd(Integer a, int i) **(0.5 puntos)**

```
public ProblemaPD<SolucionInversion, Integer> getSubProblemaAnd(Integer a,
    int i) {
  return new ProblemaInversionPDImpl(getCantidad()-a, getJ()-1);
}
```

- public SolucionInversion combinaSolucionesAnd(Integer a, SolucionInversion... ls) **(0.75 puntos)**

```
public SolucionInversion combinaSolucionesAnd(Integer a,
    SolucionInversion... ls) {
  SolucionInversion s = ls[0].clone();
  if (a > 0) {
    Accion b = getAccionJ().clone();
    b.setCantidad(a);
    s.add(b);
  }
  return s;
}
```