Práctica 3

Antonio J. Galán Herrera

Ingeniería Informática (A)

Nota máxima alcanzada en SIETTE: 60 / 100.

Problema de la Mochila

Este problema viene dado por una mochila de capacidad W y un conjunto de ítems, cada uno con un peso w, un valor v y una cantidad q de ese mismo objeto; donde el objetivo es encontrar la combinación de ítems que aportan el máximo valor a la mochila y el peso de dichos ítems no supere la capacidad de la mochila.

Resolución

> 💡 Todos los métodos incluidos en la memoria cuentan con `javadocs` en el fichero de código, pero estos no se han añadido aquí para que sea más legible y ligero.

Si se necesita un contexto más amplio, recomiendo leer directamente el fichero, ya que está comentado.

Fuerza Bruta

> ⚠️ A veces presenta fallos en SIETTE con respecto a la complejidad, depende de la ejecución de la corrección.

Método resolver()

```
public SolucionMochila resolver(ProblemaMochila problema) {
    // Datos del problema
    int[] pesos     = problema.getPesos();
    int[] valores   = problema.getValores();
    int[] unidades  = problema.getUnidades();

    // Calcular las combinaciones de unidades de los ítems y sus soluciones asociadas
    SolucionMochila[] soluciones = generarSoluciones(combinaciones(unidades), pesos, valores);

    return mejorSolucion(soluciones, problema.pesoMaximo);
}
```

La idea planteada es: generar todas las combinaciones posibles de objetos, y crear una solución para cada una de ellas, eligiendo finalmente la que aporte el mayor valor a la mochila.

Métodos auxiliares

```
private static List<int[]> combinaciones(int[] unidades) {
    List<int[]> resultado = new ArrayList<>();

    combinaciones(unidades, 0, new int[unidades.length], resultado);

    return resultado;
}


private static void combinaciones(int[] unidades, int pos, int[] combinacion, List<int[]> resultado) {
    // Caso base: se ha llegado al final de la lista de ítems
    if (pos == unidades.length) {
        resultado.add(combinacion.clone());

    // Caso intermedio: se calcula una combinación de unidades para el ítem actual
    } else {
        for (int i = 0; i <= unidades[pos]; i++) {
            combinacion[pos] = i;
            combinaciones(unidades, pos+1, combinacion, resultado);
        }
    }
}
```

Calcula todas las combinaciones posibles de las unidades de los ítems de un problema de la mochila usando recursividad.

```
private static SolucionMochila[] generarSoluciones(List<int[]> combinaciones, int[] pesos, int[] valores) {
    SolucionMochila[] soluciones = new SolucionMochila[combinaciones.size()];

    for (int i = 0; i < combinaciones.size(); i++) {
        // Datos para la solución actual
        int[] combinacion = combinaciones.get(i);
        int pesoTotal = 0, valorTotal = 0;

        // Calcular el resto de atributos de la solución
        for (int j = 0; j < combinacion.length; j++) {
            pesoTotal  += pesos[j] * combinacion[j];
            valorTotal += valores[j] * combinacion[j];
        }

        // Generar una solución asociada a la combinación de unidades actual
        soluciones[i] = new SolucionMochila(combinacion, pesoTotal, valorTotal);
    }

    return soluciones;
}
```

Genera una lista de soluciones asociadas a una combinación de unidades.

```
private static SolucionMochila mejorSolucion(SolucionMochila[] soluciones, int capacidad) {
  SolucionMochila mejorSolucion = soluciones[0];

  for (SolucionMochila solucion : soluciones) {
      // El valor de los ítems es máximo y el peso no supera la capacidad de la mochila
      if (solucion.getSumaPesos() <= capacidad && solucion.getSumaValores() > mejorSolucion.getSumaValores()) {
          mejorSolucion = solucion;
      }
  }

    return mejorSolucion;
}
```

Calcula la mejor solución de un conjunto de soluciones: aquella cuyo valor es máximo, y el peso de los ítems no supera la capacidad de la mochila.

Programación Dinámica

Método resolver()

```
public SolucionMochila resolver(ProblemaMochila problema) {
    // Datos del problema
    capacidad = problema.getPesoMaximo();
    pesos     = problema.getPesos();
    valores   = problema.getValores();

    // Datos para la tabla
    int filas     = problema.getPesos().length + 1;
    int columnas  = problema.getPesoMaximo() + 1;
    int[][] tabla   = new int[filas][columnas];

    // Generar tabla
    inicializarTabla(tabla, filas, columnas);
    rellenarTabla(tabla, filas, columnas);

    // Generar vector de la solución
    int[] combinacion = encontrarCombinacion(tabla, filas, columnas);

    // Mostrar tabla (debug)
    // mostrarTabla(tabla, problema.getPesos(), problema.getPesoMaximo());

    return generarSolucion(tabla, combinacion);
}
```

Aplicando la ecuación de Bellman propuesta en el enunciado de la práctica, se ejecutan métodos acordes a un proceso manual de la creación de la tabla que se irá rellenando.

Mis métodos auxiliares

```
private void inicializarTabla(int[][] tabla, int filas, int columnas) {
    // Inicializar la primera fila a 0
    for (int i = 0; i < filas; i++) {
        tabla[i][0] = 0;
    }

    // Inicializar la primera columna a 0
    for (int i = 0; i < columnas; i++) {
        tabla[0][i] = 0;
    }
}
```

Rellena la primera fila y la primera columna de la tabla del problema con 0.

```
private void rellenarTabla(int[][] tabla, int filas, int columnas) {
    for (int i = 1; i < filas; i++) {
        for (int j = 1; j < columnas; j++) {
            tabla[i][j] = elegirValor(tabla, i, j);
        }
    }
}
```

Rellena la tabla aplicando el criterio definido por la ecuación de Bellman.

```
private int elegirValor(int[][] tabla, int f, int c) {
    int valor;

    if (pesos[f-1] <= c) {
        valor = Math.max(tabla[f-1][c], tabla[f-1][c - pesos[f-1]] + valores[f-1]);

    } else {
        valor = tabla[f-1][c];
    }

    return valor;
}
```

Aplicación directa de la ecuación de Bellman que coloca en una posición de la tabla el valor.

```
private int[] encontrarCombinacion(int[][] tabla, int filas, int columnas) {
    int[] res = new int[valores.length];

    // Contadores
    int f = filas-1;    // Empieza en la última fila de la tabla
    int c = columnas-1;   // Empieza en la última columna de la tabla

    while (0 < f) {
        if (tabla[f][c] != tabla[f-1][c]) {
            res[f-1] = 1;
            c -= pesos[f-1];
        }

        f--;
    }

    return res;
}
```

Recorre la tabla generando la mejor combinación para crear la solución.

```
private SolucionMochila generarSolucion(int[][] tabla, int[] res) {
    int pesoTotal = 0;

    // Calcular el peso de la solución
    for (int i = 0; i < res.length; i++) {
        if (res[i] == 1) {
            pesoTotal += valores[i];
        }
    }

    // Generar una solución con la combinación, el peso y el valor de la última fila y columna de la tabla
    return new SolucionMochila(ArrayUtils.toArray(res), pesoTotal, tabla[valores.length][capacidad]);
}
```

Calcula la mejor solución a partir de una tabla.

```
public void mostrarTabla(int[][] tabla, int[] pesos, int capacidad) {
    // Imprimir una línea divisoria tan larga como la tabla
    System.out.print("\n------------------------------------------------------------\n");

    System.out.print("\t");

    for (int i = 0; i <= capacidad; i++) {
        System.out.print(i + "\t");
    }

    System.out.println();

    for (int i = 0; i < tabla.length-1; i++) {
        System.out.print(pesos[i] + "\t");

        for (int j = 0; j < tabla[i].length; j++) {
            System.out.print(tabla[i][j] + "\t");
        }

        // Mostrar separador final junto a la tabla
        if (i == tabla.length - 2) {
            System.out.print("\n------------------------------------------------------------\n");

        } else {
            System.out.println();
        }
    }
}
```

> 💡 Este método se creó para mejorar el proceso de debug, generando una representación de la tabla que se emplea en la programación dinámica.

Algoritmo Voraz

⚠️ A veces presenta fallos en SIETTE con respecto a la complejidad, depende de la ejecución de la corrección.

Método resolver()

```
public SolucionMochila resolver(ProblemaMochila problema) {
    // Ordenar los items de mayor a menor densidad mediante una función lambda
    problema.items.sort(
        (item1, item2) -> {
            double densidad1 = ((double)item1.valor)/((double)item1.peso);
            double densidad2 = ((double)item2.valor)/((double)item2.peso);
            int res;

            // Sustituir esta secuencia por 'Double.compare(densidad2, densidad1)' hace que
            // falle la complejidad del test de SIETTE, pese a la recomendación de IntelliJ
            if (densidad1 > densidad2) {
                res = -1;
            } else if (densidad1 < densidad2) {
                res = 1;
            } else {
                res = 0;
            }

            return res;
        }
    );

    // Los ítems deben ordenarse previamente para un funcionamiento correcto del algoritmo
    return algoritmoVoraz(problema);
}
```

La idea de este algoritmo es añadir los ítems de mayor a menor densidad en la mochila, con el fin de que las soluciones sean aproximadas a la solución más óptima (recordemos que un algoritmo voraz no garantiza la mejor solución).

$$\text{densidad}_i = \frac{\text{valor}_i}{\text{peso}_i}$$

Para ello, lo primero es reordenar los objetos de la lista para que luego en la iteración donde se meten en la mochila, empiece por aquel cuya densidad es mayor (aporta una mejor relación valor-peso).

Para ello sobreescribo el método `sort()` de la Colección `items` del problema, usando una función λ (recomendada por mi entorno, IntelliJ).

Mis métodos auxiliares

```
private SolucionMochila algoritmoVoraz(ProblemaMochila problema) {
    // Datos del problema
    int capacidad   = problema.getPesoMaximo();

    // Datos para el algoritmo
    int[] solucion  = new int[problema.items.size()];
    int peso        = 0;
    int valor       = 0;

    // Introducir los ítems en la mochila mientras quepan
    for (Item item : problema.items) {
        int i = item.index;
        int unidades = item.unidades;

        // Introducir tantas unidades de ese ítem mientras quepan
        while (0 <= capacidad - peso - item.peso && solucion[i] < unidades) {
            solucion[i]++;
            peso += item.peso;
            valor += item.valor;
        }
    }

    return new SolucionMochila(solucion, peso, valor);
}
```

Se recorre la lista de ítems (que estará ordenada) y se ocupa la mochila con los objetos con mejor relación valor-peso primero, mientras que sea posible.