PRÁCTICA 3 – Programación Dinámica

Llenar el camión con los productos más beneficiosos

Objetivos

- Construir soluciones a un problema utilizando el **método algorítmico de programación dinámica**. Analizar y comparar los algoritmos implementados desde diferentes perspectivas.
- Realizar el **análisis de la eficiencia** de las soluciones aportadas, y una comparativa tanto desde el punto de vista teórico como práctico.

Requerimientos

Para superar esta práctica se debe realizar lo siguiente:

- Recordar (de EDA I) las **estructuras de datos** más apropiadas para la resolución de problemas.
- Conocer cómo implementar el **esquema de programación dinámica** para responder a una necesidad concreta (en este caso, resolver un problema de optimización).
- **Evaluar los algoritmos de programación dinámica implementados**, y compararlos en su caso con implementaciones de algoritmos greedy, relacionando ambos métodos algorítmicos.

Enunciado del problema

En **EDAMarket** existe mucho movimiento de verduras para intentar proporcionar el abastecimiento de este producto a todo **EDAland**. EDAMarket tiene su sede principal en Almeria, cuya producción en verduras es la principal del país, fundamentalmente por su calidad. La gran parte de distribuidores de verduras vienen frecuentemente a **EDAMarket-Almeria** para conseguir los mejores productos que éste ofrece a diario. Un ejemplo de este distribuidor es **Karrayon**, que viene de Madrid con su camión para poder llevárselo lo más lleno posible y con los productos que les puedan reportar el mayor beneficio. Para poder conseguir esto, éste siempre está el primero en la puja que se hace en EDAMarket-Almeria. En resumen, Karrayon, que siempre apuesta por la calidad de lo que distribuye, le ha pedido a **theBestSoft** (**EDASoft**) que le desarrolle una aplicación para que le asesore en el llenado de su camión **con aquellos objetos (verduras) que quepan en él (y estén disponible en la puja), sabiendo que éste tiene una capacidad (Peso Máximo Autorizado) limitada, y que le puedan proporcionar el mayor beneficio posible**.

Es decir, debemos resolver el problema en el que tenemos **n** objetos (verduras), cada una con un peso natural p_i ($p_i > 0$) y con un beneficio (que se puede obtener al venderlo) de un valor real b_i ($b_i > 0$) $\forall$ $1 \leq$ **i** $\leq$ n, y que el camión tiene un PMA (Peso Máximo Autorizado) de **P**. Lo que pretendemos es **maximizar** $\sum_{i=1}^{n} x_i \cdot b_i$ con la **restricción** de que $\sum_{i=1}^{n} x_i \cdot p_i \leq P$, donde $x_i \in \{0, 1\}$ expresa que sí hemos escogido (1) o no (0) el objeto **i**. Para ello, **EDASoft** conoce la función de recurrencia, en la que dado el conjunto **S** de *n* objetos, sea $\mathbf{S_i}$ el conjunto de los **i** primeros objetos $1 \leq$ **i** $\leq$ n, entones tenemos **camion(i, j)** como el máximo beneficio obtenido a partir de los **i** elementos de $\mathbf{S_i}$ para un camión de peso máximo **j**.

$$camion(i,j) = \begin{cases} camion(i-1,j) & x_i = 0 \\ camion(i-1,j-p_i) + b_i & x_i = 1 \end{cases}$$

$$camion(i,j) = \begin{cases} camion(i-1,j) & 1 \leq j < p_i \\ \boldsymbol{max}\{camion(i-1,j), camion(i-1,j-p_i) + b_i\} & j \geq p_i \end{cases}$$

donde $1 \leq \mathbf{i} \leq n$ y $1 \leq \mathbf{j} \leq P$ (P se corresponde con el PMA del camión). La recursión está bien fundada puesto que uno o los dos argumentos decrecen estrictamente. Además, los casos base se presentan, bien cuando no tenemos objetos que considerar, o bien cuando no nos queda espacio en el camión. En ambos casos el único beneficio posible es 0.

$$camion(0,j) = 0 \qquad 0 \leq j \leq P$$
$$camion(i,0) = 0 \qquad 0 \leq i \leq n$$

Trabajo a desarrollar

Deberá proponer e implementar soluciones con el esquema algorítmico de **programación dinámica** a los problemas que se plantean a continuación:

1. Implementar en Java un *método recursivo* que, verificando la recurrencia anterior, devuelva el máximo beneficio que podría transportar el camión si éste tiene un PMA de `P`. Tenemos un array `p` con los pesos de los objetos, y otro array `b` con sus beneficios. La cabecera de la función a implementar puede ser:
   ```
   double camionRecursive(int n, int P, int p[], double b[])
   ```
2. Implementar en Java un método que evite el recálculo que se realiza en el caso anterior utilizando únicamente la recursividad. Para ello, deberá utilizar una **tabla** (matriz bidimensional) con **n + 1** filas y **P + 1** columnas en la que se irán almacenando los valores para ser utilizados posteriormente (muestre el valor final que tendría dicha tabla e interprételo) y no necesitar el recálculo que se hace con la recursividad. Para ello se deben implementar dos funciones (`camionTable`, que llama internamente a `camionRec`), cuyas cabeceras pueden ser:
   ```
   double camionRec(int n, int P, int p[], double b[], double[][] table)
   double camionTable(int n, int P, int p[], double b[])
   ```
3. Suponiendo que los objetos (verduras) no se pueden fraccionar (romper) y que los pesos son exactos (número naturales positivos), determinar qué objetos debe elegir **Karryon** para maximizar el valor total de lo que pueda llevarse en su camión, sabiendo que los precios (beneficios) de los objetos que ofrece EDAMarket-Almeria son números reales positivos. Implementar en Java la solución al problema, según el algoritmo visto en clase (`knapSackDP`) en su versión iterativa (utilizando Programación Dinámica) y exponer el resultado en función de array `Sol` ($Sol[i] \in \{0,1\}$) y en función de lo que proporciona la función `test`, que nos proporciona realmente los objetos a considerar según la solución `Sol`. Probar su correcto funcionamiento para el caso: **`P`** = 20, **`p`** = {3, 6, 7, 1, 4, 5, 1, 3}, **`b`** = {8.0, 8.0, 2.0, 3.0, 4.0, 3.0, 2.0, 5.0}. Para ello se deben implementar dos funciones (`camionDP` y `test`), cuyas cabeceras pueden ser:
   ```
   double camionDP(int P, int p[], double b[])
   void test(int j, int c, int p[], double b[], double camion[][])
   ```

4. Si mostramos también los valores que tiene la tabla o matriz bidimensional `camion` (variable local al método) del algoritmo del apartado 3, ¿Se puede deducir alguna relación con los valores de la matriz bidimensional `table` del caso 2?. Exponerlo todo de forma razonada.
5. Implementar en Java, la otra forma de resolver el problema (otra forma de llenar la matriz **B** de forma iterativa, en este caso será la matriz `camion`) vista en clase (Algoritmo 6.4 en la transparencia 93), y obtener la 'salida de la función `test`' sin necesidad de utilizar dicha función recursiva (fragmento de código iterativo). Probarlo también para el mismo caso del apartado 3 (P, p[], b[]). Para ello se deben implementar una única función (`camionDP2`), cuya cabecera puede ser:
   `double camionDP2(int P, int p[], double b[])`
6. ¿Sería posible encontrar el mayor beneficio o beneficio total que se obtiene al resolver nuestro problema con programación dinámica, pero utilizando únicamente un array unidimensional (`camion`) como estructura de datos, en lugar de utilizar una tabla o array bidimensional como hemos visto en los anteriores casos?. De ser posible, implementarlo en Java con la siguiente cabecera y explicar su funcionamiento de forma detallada:
   `double camionDP3(int P, int p[], double b[])`
   https://www.enjoyalgorithms.com/blog/zero-one-knapsack-problem
   ¿Sería sencillo en este caso encontrar el array `Sol` ($Sol[i] \in \{0,1\}$)?. Justifique su respuesta.
7. Este apartado es **obligatorio para los grupos de 3** miembros y **optativo para los grupos de 2**. Suponiendo que ahora hay una cantidad inagotable de objetos de tipos (clases) distintos. Es decir, en vez tener **n** objetos distintos, disponemos ahora de **n** tipos de objetos distintos. Cada tipo de objeto es indivisible (no se puede romper), tiene un cierto peso (natural positivo) y un cierto beneficio (real positivo). Lo que se pretende es encontrar qué cantidad de objetos de cada tipo se debe coger para maximizar el valor total de lo que puede llevarse en el camión con PMA `P`.
   a) Indicar la expresión de recurrencia para este problema de una forma similar a como se ha expuesto para **camion(i, j)**, y exponer la similitud con la recurrencia que tenemos para el *problema del cambio de monedas*, que nos puede servir para su implementación.
   b) Implementar en Java el algoritmo y probarlo para el caso del apartado 3 (**`P`**, **`p`**[], **`b`**[]). Para ello se debe implementar una única función (`camionDPMP`), cuya cabecera puede ser:
      `double camionDPMP(int P, int p[], double b[])`
   c) Intente implementar en Java la forma de obtener el array `Sol` ($Sol[i] \in \{0,1\}$) para este caso. Y justifíquelo todo detalladamente.
8. Suponiendo que los objetos **no** se pueden fraccionar, pero los **pesos pueden no ser exactos** (los pesos pueden ser números reales positivos). Indicar razonadamente qué tendría de especial este caso con respecto a los apartados vistos anteriormente, por ejemplo, para el apartado 3.
9. Finalmente, imaginemos el caso de que los objetos se pueden fraccionar, ¿sería sencillo encontrar una solución con *programación dinámica* para determinar qué objetos elegir para maximizar el valor total de lo que puede llevarse en el camión?. ¿Sería más sencillo realizarlo con un algoritmo *greedy* (voraz)?. De ser así, implementarlo y probarlo para varios casos. Además, exponer las principales diferencias (en forma de tabla) entre los dos esquemas algorítmicos (programación dinámica y greedy) para este problema concreto.

Para ello deberá realizar los siguientes apartados:

- **Estudio de la implementación**: Explicar los detalles más importantes de la implementación llevada a cabo, tanto de las estructuras de datos utilizadas para resolver el problema en cuestión, como de todos los algoritmos implementados. El código debe de estar razonablemente bien documentado (JavaDoc).
- **Estudio teórico**: Estudiar la *complejidad* de los algoritmos (PD) implementados, en función del *tiempo de ejecución* (complejidad temporal) y de los *recursos* consumidos (complejidad espacial). Comparar también los algoritmos propuestos, teniendo en cuenta las características del problema concreto que resuelven y cómo se actualizan las tablas utilizadas para ello. Comparar las dos técnicas algorítmicas vistas en esta práctica: *programación dinámica* y *greedy*, destacando sus características, ventajas, desventajas, etc.
- **Estudio experimental**: La validación de los algoritmos de (PD) implementados para resolver los problemas planteados, pueden utilizar los conjuntos de datos disponibles en la siguiente Web: https://people.sc.fsu.edu/~jburkardt/datasets/knapsack_01/knapsack_01.html. Podemos ver que existen varios conjuntos, y en ellos se pueden observar datos relativos a: capacidad de la mochila, pesos de los objetos, valor o beneficio de los objetos y la selección óptima de pesos (solución). Para ello, se deberán comprobar el correcto funcionamiento y comparar los tiempos de ejecución de los algoritmos implementados. Se contrastarán los resultados teóricos y los experimentales, comprobando si los experimentales confirman los teóricos previamente analizados. Se justificarán los experimentos realizados, y en caso de discrepancia entre la teoría y los experimentos se debe intentar buscar una explicación razonada. Además, se generarán **datos aleatorios** (de peso y beneficio), en concordancia con el formato de los disponibles en la Web, para valores de **n** muy grandes, comprobando si se aprecia alguna variación en los tiempos de ejecución de los algoritmos.

Entregas

Se ha de entregar, en fecha, en el repositorio GitHub (mismo repositorio para todas las prácticas de EDA II) con toda la documentación y el código fuente requerido en la práctica:

- En dicho repositorio, en la carpeta de fuentes `src/main/java`, crear un nuevo paquete llamado `org.eda2.practica03`, para el **código fuente** de la práctica. Además, en esta carpeta deben de incluirse los archivos de datos de la Web y los generados sintéticamente.
- En la carpeta `docs`, dedicada a la **documentación**, crear una subcarpeta `practica03` para guardar toda la documentación (documento en `pdf` y los fuentes utilizados para su creación (por ejemplo, `.docx`)).
- **Memoria** que explique todo lo que habéis realizado en la práctica. La memoria deberá tener el formato que se indica a continuación. Si se desea, también se podrá realizar una presentación de la práctica.
- **Código fuente** de la aplicación, desarrollada en JAVA o en C++, que resuelva todo lo planteado en la práctica.
- **Juegos de prueba** que consideréis oportunos incluir para justificar que todo funciona correctamente (fundamentalmente para los casos de los datos que hay disponibles en la Web). En este caso los juegos de prueba deben estar en la carpeta de fuentes llamada `src/test/java` y dentro de ella en un paquete llamado `org.eda2.practica03`.

La **memoria de la práctica** a entregar debe ser breve, clara y estar bien escrita, adaptándose en lo posible a la norma UNE 157001:2014 "Criterios generales para la elaboración de proyectos". Ésta debe incluir las siguientes secciones:

- Un apartado **Objeto** con un estudio teórico del método algorítmico utilizado en esta práctica (programación dinámica). Además, en esta sección deberá exponerse claramente ***qué miembro* del grupo ha actuado como *líder* y *qué tareas ha realizado cada miembro del grupo***. Es muy importante que todos los miembros dominen la práctica en su conjunto. Se puede incorporar la hoja de datos indicada para tal fin (`PR3_Tareas_a_repartir.xlsx`).
- Un apartado **Antecedentes** en el que se explique el motivo que lleva a realizar esta memoria y se enumeran los aspectos necesarios, en su caso, de las alternativas contempladas y la solución final adoptada.
- Una sección para cada uno de **apartados propuestos** a desarrollar en esta práctica (estudio de la implementación, estudio teórico y estudio experimental). Para el **estudio experimental**, el correcto funcionamiento de los algoritmos debe de justificarse con la correspondiente captura de pantalla en la que se pueda apreciar la salida correcta. Hemos de remarcar que deben incluirse los apartados en el mismo orden en el que se han expuesto. El apartado **estudio experimental** de la memoria recogerá los resultados finales de lo realizado.
- Se incluirá también un **anexo** con el diseño del código implementado, con diagramas de clases y cualquier otro diagrama que estiméis necesario incluir, **no introducir ningún código en este anexo**. Además, añadir en esta sección una lista de los archivos fuente que componen la práctica y una breve descripción del contenido de cada uno.
- En el apartado de **estudio experimental** o en un **anexo**, se expondrán los cálculos realizados en el estudio experimental, con la explicación del método seguido para la toma de datos y gráficos con el resultado obtenido (datos de muestras y ecuación seleccionada). Cuando se elija una curva para explicar el orden de un algoritmo (por ejemplo, O(n)), se deberá añadir el estadístico de la varianza explicada R^2, de forma que este valor (que varía entre 0 y 1) debería ser superior a 0.95 para que se acepte la solución propuesta. Valores por debajo 0.90 indican que el proceso de toma de muestras o el de elaboración de la hipótesis, ha sido incorrecto y debería llevar a su revisión. Cada muestra debería ser una media de varias medidas (al menos 10) de tiempo de ejecución y el tiempo medido debería ser, al menos, cercano al segundo para evitar interferencias del sistema operativo.
- Es importante incluir siempre las **fuentes bibliográficas** utilizadas (web, libros, artículos, etc.) y hacer referencia a ellas en el documento.

Evaluación

Cada apartado se evaluará independientemente, aunque es condición necesaria para aprobar la práctica que los programas implementados funcionen correctamente.

- La implementación junto con la documentación del código se valorará sobre un 40%
- El estudio de la implementación se valorará sobre un 10%
- El estudio teórico se valorará sobre un 15%
- El estudio experimental se valorará sobre un 35%

Se penalizará no entregar el apartado de introducción teórico o una mala presentación de la memoria.

Se podrá requerir la defensa del código y de la memoria por parte de profesor.

Fecha de entrega: 5 de Mayo de 2024