LAB 5 - Knapsack Problem

Submission Deadline: April 7, 11:59 pm
Assessment: 5% of the total course mark.

Overview:

In this assignment you are required to write a `Java` implementation of several algorithms for the knapsack problem. This problem has two variants: 1) the 0-1 variant and 2) the *fractional* variant. The fractional variant can be solved using a greedy algorithm, while the 0-1 variant can be solved via dynamic programming. Note that a greedy algorithm can also be designed for the 0-1 variant, but it generates a suboptimal solution in general. You have to implement this method as well. To conclude, you have to write three `Java` classes: 1) the class `KnapsackGreedy`, which implements greedy algorithms for both variants; 2) the class `KnapsackDP`, which implements optimal algorithms for the 0-1 variant; 3) the class `KnapsackTest`, which tests your implementation. All classes have to be declared in the same package. **You are not allowed to use any predefined `Java` methods other than for input and output**.

Description of the knapsack problem:

In the knapsack problem, we are given a set S of n items, denoted $s_1, s_2, \cdots s_n$, each item having a value and a weight, both being positive integers. The value and the weight of item s_i are denoted v_i and w_i, respectively, for $1 \leq i \leq n$. We are also given a target weight W (a positive integer).

1) The 0-1 variant requires finding a subset A of maximum total value, whose total weight is smaller than or equal to W. Thus, the solution A can be represented as an array ind, where $ind[i] = 1$ if item s_i IS in A and $ind[i] = 0$ if element s_i IS NOT in A. Note that the total value and the total weight of A can be calculated as follows

$$totalValue(ind) = ind[1] * v_1 + ind[2] * v_2 + \cdots + ind[n] * v_n, \quad (1)$$
$$totalWeight(ind) = ind[1] * w_1 + ind[2] * w_2 + \cdots + ind[n] * w_n. \quad (2)$$

   Note that in the case when W is larger than the weight of each item, the solution to the problem is the empty set. The corresponding array ind has all elements equal to 0, $totalValue(ind) = 0$ and $totalWeight(ind) = 0$.

   Details about the solution algorithms to the 0-1 variant of the knapsack problem are given in the appendix.

2) In the fractional variant we allow the numbers in array ind to take fractional values between 0 and 1, inclusive, thus $ind[i]$ represents the fraction of the item s_i that we include in the solution. The goal remains the same: to maximize $totalValue(ind)$, while $totalWeight(ind) \leq W$.

SPECIFICATION: ◇ Class `KnapsackGreedy`.

Each object of this class represents an instance of the knapsack problem. It is your decision what instance fields to declare (in other words, what attributes to store for each object). The class must include at least one **constructor**, which initializes the instance fields according to an input string.

The class must contain the following `public static` methods (**you may declare additional `private` or `public` methods**).

- `public static String fractional(String input)` - solves the fractional knapsack problem specified by the input string using the greedy algorithm and outputs a string that represents the solution; the method should first create a `KnapsackGreedy` object that corresponds to the problem specified by the input, then solve the problem for this particular object.
- `public static String greedy01(String input)` - implements a greedy algorithm for the 0-1 knapsack problem specified by the input string and outputs a string that represents the result; the method should first create a `KnapsackGreedy` object that corresponds to the problem specified by the input, then solve the problem for this particular object.

For both methods, the **input string** consists of a sequence of positive integers separated by white spaces. The first integer is n, the next integer is W. The following two integers are v_1 and w_1, the next two integers are v_2 and w_2, and so on. The last two integers are v_n and w_n. You may assume that the input string satisfies the format with no errors.

Example. If the input string is "4 8 7 10 2 3 1 5 9 2", then $n = 4, W = 8$, $v_1 = 7, w_1 = 10, v_2 = 2, w_2 = 3, v_3 = 1, w_3 = 5, v_4 = 9, w_4 = 2$.

The **output string** specifies the solution by listing the value of $ind[i]$ in increasing order of i, separated by a comma and a space, followed by the total value and the total weight. The result of `greedy01` for the above example is the subset formed of s_2 and s_4. Then the output string should be:
"0, 1, 0, 1, total value = 11, total weight = 5".
Note that there is space after each comma and before and after each "=" sign.

◇ Class `KnapsackDP`. Each object of this class represents an instance of the 0-1 knapsack problem. It is your decision what instance fields to declare (in other words, what attributes to store for each object). The following are recommended instance fields:

```
private  int num;
private  int targetWeight;
private  int[] value;
private  int[]  weight;
private  int[][] x;
private  int[][] totalValue;
private  int[][] totalWeight;
```

The two dimensional arrays are for storage of the results of the subproblems that you will need to solve.

The class must include at least one **constructor**, which initializes the instance fields according to an input string. The class must contain the following `public static` methods (**you may declare additional** `private` **or** `public` **methods**).

- `public static String recMemo(String s)` - solves the 0-1 knapsack problem specified by the input string in $O(nW)$ time using recursion with memoization and outputs a string that represents the solution; the method should first create a `KnapsackDP` object that corresponds to the problem specified by the input, then solve the problem for this particular object by invoking a private instance method to perform the recursion with memoization.
- `public static String nonRec(String s)` - solves the 0-1 knapsack problem specified by the input string nonrecursively in $O(nW)$ time (using bottom-up dynamic programming) and outputs a string that represents the solution; the method should first create a `KnapsackDP` object that corresponds to the problem specified by the input, then solve the problem for this particular object.

The **input and output strings** obey the same format as for the previous class.

◇ Class `KnapsackTest`. This class may follow the template posted on avenue. You have to include test cases with $n \geq 10$ that illustrate at least the following situations: 1) the greedy algorithm for the 0-1 variant gives the optimal solution; 2) the greedy algorithm for the 0-1 variant gives a suboptimal solution; 3) the optimal solution to the 0-1 variant uses all available target weight W; 4) the optimal solution to the 0-1 variant does not use all available target weight W. You must submit the output of a sample test. The output should also include messages that specify what case is illustrated by each test case.

Submission Instructions: Submit the source code for each of the `Java` classes `KnapsackGreedy`, `KnapsackDP` and `KnapsackTest` in a separate text file. In addition, submit a text file containing the output of a sample test. Include the name of the class, your name and student number in the name of each file.

Appendix: The algorithms to solve the 0-1 variant of the knapsack problem are based on the following observation. If s_n IS NOT in the solution subset A, then A must also be a solution to the 0-1 knapsack problem corresponding to the set S' that contains only the items $s_1, s_2, \cdots, s_{n-1}$, the target weight being W. On the other hand, if s_n IS in the solution subset A, then $A \setminus \{s_n\}$ must be a solution to the 0-1 knapsack problem corresponding to the set S', but with the target weight being $W - w_n$. Note that the latter situation can happen only if $W \geq w_n$.

The above observation implies that the problem can be solved recursively. In the recursive process we solve a series of subproblems. Let $P(i, t)$, for $0 \leq n$, denote the subproblem corresponding to the input set that consists only of items $s_1, s_2, \cdots, s_i$, the target weight being t. The values and the weights of the items remain the same as in the original problem. When $i = 0$ the input set is empty. If $t \geq 0$, the solution to $P(0, t)$ is the empty set.

In order to solve $P(i, t)$, we need to first solve $P(i-1, t)$. What we do next depends on whether $t < w_i$ or $t \geq w_i$.

- ◇ If $t < w_i$, then we know that s_i cannot be in the solution to $P(i,t)$, therefore we conclude that the solution to $P(i,t)$ is the same as the solution to $P(i-1,t)$.
- ◇ If $t \geq w_i$, then there is the possibility that s_i is in the solution to $P(i,t)$. Therefore, we have to also solve $P(i-1,t-w_i)$. Now we have two possibilities: 1) the solution to the problem $P(i,t)$ could be the subset B, where B is the solution to $P(i-1,t)$, or it could be the subset $C \cup \{s_i\}$, where C is the solution to $P(i-1,t-w_i)$. Both possibilities satisfy the requirement that the total weight be smaller than or equal to the target t. To determine which of the two possibilities gives us the solution to $P(i,t)$, we only need to compare their total values and pick the one with the largest total value.

The base cases of the recursion are $P(0,t)$. Note that in order to arrive at the solution to the original problem $P(n,W)$, it is sufficient to solve the subproblems $P(i,t)$ for all $0 \leq i \leq n-1$ and $0 \leq t \leq W$, i.e, a total of $O(nW)$ subproblems. If we implement the algorithm recursively, some of the subproblems will be solved multiple times leading to a $O(2^n)$ running time, which is intractable for large n. This can be avoided by storing the solutions to the subproblems that are solved. This idea can be implemented in two ways:

- ◇ Recursion with memoization: use a recursive algorithm, but store the solution when a subproblem is solved. Before making a new recursive call, first check if the subproblem has already been solved and if it has, use the available result instead of making the recursive call.
- ◇ Bottom-up dynamic programming: solve all the subproblems bottom up, i.e., in increasing order of i and t, and store the results. This way, when you need to solve $P(i,t)$, you already have the solutions to $P(i-1,t)$ and $P(i-1,t-w_i)$ available.

Note that it is not necessary to store the whole solution to each subproblem. For each $P(i,t)$ it is sufficient to store the value of $ind[i]$, the total value and the total weight of the solution. For this you need two-dimensional arrays. For instance, you can use a two-dimensional array x and store in $x[i][t]$ the value of $ind[i]$ corresponding to the solution to subproblem $P(i,t)$.