Comp 482 Fall 2017

Computer Project #2 0-1 Knapsack Problem
Points: 35
Due: Nov 7

0-1 Knapsack Problem Statement: Given n items each with a positive integer weight and a positive integer benefit and given a knapsack that holds at most weight W, fill the knapsack with a subset of the n items so that (1) the total weight of the items <=W and (2) the total benefit of the items is maximized. You cannot use fractional items.

In this project you will implement in Java two different methods that solve the Knapsack Problem(KNP) and a greedy method which gives an approximate solution to the KNP Problem. You will also create an algorithm to generate all subsets of { 0,…,n-1} independently of the KNP algorithm. All work must be your own. Do not download solutions from the internet or from other outside sources.

```
Input:   int n   //number of items          //index items from 1 to n
         int[] w // w[i] = weight of item i  //Set w[0] = -1
         int[] b  // b[i] = benefit of item i  //Set b[0] = -1
         int W    //weight capacity of the knapsack
```

Output: *(See format of output below under test cases)*

a. Optimal set of items to put into the knapsack
b. Total weight of the items in knapsack. Total benefit of the items in knapsack

Create a Knapsack class that contains the methods described in Items 1,2,3 below. See class outline below for the exact signatures needed for all public methods. All other methods must be private.

1. **Generate Subset Method.** Create a public static method with signature *int[] generateSubset(int k, int n)* that generates the kth subset of {0, …n-1} where $n>0$ and $0<= k <= 2^n -1$ and the kth subset is the binary representation of k using n bits.

2. **Brute Force Method to Solve KNP.** Create a public method *BruteForceSolution()* that generates and prints all optimal solutions of the KNP problem. Use your *generateSubset* method to generate all subsets of the input items. Test each subset for feasibility and optimality.

3. **Dynamic Programming Method to solve KNP.** Create a public method *DynamicProgrammingSolution(boolean printBmatrix)* that generates and prints one optimal solution to the KNP problem. You must use the dynamic programming recurrence equations

from class. (See below) You must implement the dynamic programming algorithm. The 2-d matrix B must be explicitly created using a $O(n^2)$ algorithm that fills in the array.

```
$B[0,w] = 0$  for  $0 <= w <= W$
for $k > 0$,
    if $w < w_k$ then $B[k,w] = B[k-1,w]$.  Otherwise,
    $B[k,w] = \max \{ B[k-1,w] , B[k-1, w - w_k] + b_k \}$
```

Once the OPT matrix is filled in. Your program must use it to find the optimal set of items to be added to the Knapsack, as well as the optimal benefit. Print the OPT matrix if requested.

4. **Greedy Approximation Algorithm**. Create a public method *GreedyApproximateSolution()* to generate and print an approximate solution to the KNP Problem using the following algorithm. Add items (whole items only) to the knapsack in decreasing benefit/weight ratio order. That is, add the highest benefit/weight ratio first. Add items as long as the sum of the weights of the added items <=W. This is a variation of the solution to the Fractional Knapsack Problem. Just don't add in the last fractional item.

5. **Experiment**. Perform some experiments with random data and various values of n to see how good or bad the Greedy Approximation is. Look up on the internet for a theoretical result. Write a one or two page report on your findings.

6. **Test Cases**. Test your methods using the following arrays of weights w_i, values v_i and knapsack capacities.
   Test Case #1

   ```
    int n = 6;
   int[] weights = {-1, 2,4,3,4,4,1};
   int W = 10;
   int[] values = { -1,1,2,3,3,3, 6}
   ```

Other test cases will be posted online by Oct 31.

Test Case Output Format: Your subsets of items must be formatted as {1,3,4,6} to indicate items 1,3,4,6 have been selected. These must be the indices that match the input indices. The computer program output format of your test cases should look like this:

```
Test Case #1
Knapsack Problem Instance
Number of items = 7  Knapsack Capacity = 100
Input weights:  [-1, 60, 50, 60, 50, 70, 70, 45]
Input benefits:  [-1, 180, 95, 40, 95, 40, 40, 105]

Brute Force Solutions
Optimal Set= { 4,7 } weight sum = 95   benefit sum = 200
Optimal Set= { 2,7 } weight sum = 95   benefit sum = 200
```

```
Dynamic Programming Solution
Optimal Set= { 2,7 } weight sum = 95   benefit sum = 200


Greedy Approximate Solution
Optimal Set= { 1 }   weight sum = 60   benefit sum = 180
```

.

7. **HAND IN on Nov 7**(Stapled in this order)**:**
   a. Cover Page
   b. One or two page report on the performance of the Greedy Approximation method.
   c. Output from Test Case #1.
   d. Output from other Instructor Test Cases. (Posted Oct 31)
   e. Files: Knapsack.java , KnapsackDriver.java. This last file contains the program that produced the output for the test cases and for your experiment.

Electronic Submission:

Upload your file *Knapsack.java* to Canvas by 12 noon on Nov 7.

```
//Outline for Project 2
//All methods specified below must match the signatures exactly
//Add additional private methods and fields as needed. Do not add any public methods or fields.

import java.io.*;

public class Knapsack
{
  //set private fields here

    public Knapsack(int W, int[] w, int[] b)
    {
      //constructor


    }

    public static  int[] generateSubset(int k, int n)
    {
    //  0 <= k <= 2^n - 1
    //  Generates the kth subset of { 0,1,..., n-1 }
    //  in the form of the binary representation of k
    }

    public void BruteForceSolution()
    {
        //Prints all optimal solutions to the 0-1 knapsack problem
        //using brute force algorithm described in Project 2
        //Print solution in format specified in Project 2
    }




    public void DynamicProgrammingSolution(boolean printBmatrix)
    {

          //Prints one optimal solutions to the 0-1 knapsack problem
         // using dynamic programming algorithm described in Project 2
         // Print solution in format specified in Project 2
        // If printmatrix is true, print the OPT matrix.
    }

   public void GreedyApproximateSolution()
   {
      //Prints one approximate solution to the 0-1 knapsack problem using a variation of
      // used to solve the Fractional Knapsack Problem.


   }
```

Project 2   Comp 482
Instructor Testcases
Solve the 0-1 Knapsack Problem for each of the test cases using the   Brute Force, Dynamic
Programming and the Greedy Approximation methods.   For Testcase #4 ONLY, print the B
matrix.

```
Testcase #1
      int n = 7;
      int[] weights = {-1, 60, 50, 60, 50, 70, 70, 45};
      int W = 100;
      int[] benefits = {-1, 180, 95, 40, 95, 40, 40, 105};
Testcase #2
      int n2 = 18;
      int[] weights2 = {-1,25,4,2,5,6, 2,7,8,2,1, 1,3,5,8,9,  6,3,2};
      int W2 = 39;
      int[] benefits2 = {-1,75,7,4,3,2,  6,8,7,9,6,  5,4,8,10,8,
1,2,2};
Testcase #3
      int n3 = 20;
      int[] weights3 = {-1, 10,14,35,12,16, 20,13,7,2,4, 3,10,5,6,17,
                        7,9,3,4,3};
      int W3 = 29;
      int[] benefits3 = {-1, 2,13,41,1,12, 5,31,2,41,16,
                         2,12,1,13,4, 51,6,12,1,9};
Testcase #4
      int n4 = 7;
      int[] weights4 = {-1, 2,5,3,2,5,3,7 };
      int W4 = 10;
      int[] benefits4 = {-1, 5,10,5,20,15,5,10};
```