Dr. Hasmik Gharibyan
CSC/CPE349

Project 4 (100 points): Change Making Problem

Part 1: dynamic programming algorithm; Part 2: greedy algorithm

Deadline: submit your files **electronically** by **midnight** (end of day) on **Monday, 05/17/21.**

Late submission: you can submit your work within 24 hours after deadline (penalty applies):
- -by 2 a.m. on 5/18/21 for **10-point penalty**,
- -from 2:01 a.m. till *noon* on 5/18/21 for **25-point penalty**,
- -from 12:01 p.m. till *midnight* (end of the day) on 5/18/21 for **50-point penalty**

Programming Language: ***Java***.

Change Making Problem:
A cashier at a store accepts money in payment for goods, and needs to give back change in the amount of *n* cents. The money comes in a finite number of fixed coin denominations. Assuming that the cashier has access to *unlimited quantities* of coins of every denomination, the cashier needs to give change using the least number of coins for the given amount *n*. To make the problem more comprehensive, let's include also dollar bills – we can think of an x-value banknote as a coin of x*100 denomination (e.g. $10 can be considered a 1000-cent coin). To sum up, the problem is:
Problem: given a set of coin-denominations $\{d_1, d_2, \ldots, d_k\}$ and an amount *n*, make change for *n* using the ***minimum*** number of coins. There is *no* limit on the number of coins of any d_i.
Precondition: to guarantee that any *n*-amount can be changed with the given set of coins, let's agree that *the set will always contain a 1-cent coin*.

Objectives of the assignment:
1. For the same problem (Change Making problem) design two algorithms using two different design strategies: an **O(n^2)** *dynamic programming* algorithm and an **O(*n*)** *greedy* algorithm.
2. Demonstrate that the greedy algorithm returns an optimal solution in certain scenarios (e.g. if the coin set represents US denominations). However, there is no guarantee an optimal solution will be found if the coin-denominations in the set are allowed to be random values.

Do NOT change the class names and method signatures specified in the assignment.

PART1: **Design and implement a *dynamic programming* algorithm for the above problem.**
You are required to use the **BOTTOM-UP approach (i.e. iterative approach)** -- NO recursion.

Implement a class called ***ChangeMaker*** that has **two *public* methods**: *main* and *change_DP*.
You may have private methods as you find appropriate, to support your public methods.

- *change_DP* method: this method should have the following signature:

public static int[] change_DP(int n, int[] d) //needs the *n* amount and the *d*-array of coin-values

Your *change_DP* method should implement a dynamic programming algorithm (***Bottom-up* method)** that receives the *n* amount and the *d*-array, and returns an array containing the ***count of coins*** **for each d_i-denomination** to be used when making change for the given *n* amount.

Example: if *d*-array contains 100,25,10,5,1 and *n*=87, then the return array will contain: 0,3,1,0,2 values respectively. These data indicate that there are 0 coins in d_1-denomination (i.e. no 100-cent coins), 3 coins of d_2-denomination (i.e. three 25-cent coins), 1 coin in d_3-denomination (i.e. one 10-cent coin), 0 coins in d_4-denomination (i.e. no 5-cent coins), 2 coins in d_5-denomination (i.e. two 1-cent coins). From the contents of this array it is easy to see that 6 coins were used.

Details of implementation: We can take a sub-problem of this problem to be one where the amount to be changed is j (j≤*n*) with the same set of coin-denominations. Let's define c_j to be the value of the optimal solution for such sub-problem (i.e. the min # coins needed to change j). You need to define an array C[0..*n*] to hold c_j values for all sub-problems; your goal is to get the C[*n*].

1.Get convinced that the Money Changing problem has an optimal sub-structure

2.Get convinced that the value of the optimal solution can be defined recursively as follows:

$$C[j] = \begin{cases} \infty & if\ j < 0 \\ 0 & if\ j = 0 \\ 1 + \min\limits_{1 \le i \le k} \{C[j - d_i]\} & if\ j > 0 \end{cases}$$

//Make sure you understand this formula

Attention: You will **not** implement the j<0 case (can't have a negative amount). The negative-argument case only arises when referring to C[j-d_i]-values *while finding the min* (it will happen if j<d_i). To **avoid** the negative-argument case, **before** referring to each C[j-d_i] value while finding the min, first check and make sure that $j \geq d_i$ (if j<d_i, then simply skip that C[j- d_i]).

3-4. And now design an algorithm to solve all sub-problems from smallest to largest. Since later you will need to obtain the actual coin-distribution when making the change, you need to define an auxiliary array A. In each A[j] cell you will save info about the choice that was made when computing the value of respective C[j]; in other words, you will save the **i-value** representing the d_i coin that was chosen when computing the minimum of all C[j-d_i] values (e.g. if C[j-d_m] is the min among all C[j-d_i] values for i=1,2,…,k, then *m* is saved in A[j] cell).

Implement your algorithm to do the following:

1. Compute and fill-in the values in C and A arrays.
2. Arrange the creation of a k-length **result-array** B containing *counts* of coins in the coin-distribution for changing *n*. In this array each i cell contains the count of d_i-denomination coins used when making change for the given *n* amount. Note: the result is **not** the A array.
   Hint: To obtain this array, you can think of doing the "printing" of the optimal solution using the A-array; except, at each step instead of printing the x-value registered in an A-cell, increment the count-value of the x-cell in the B-array. Note that "printing" of the solution is similar to the one in Rod-Cutting Problem (last page of respective handout).

- *main* method: in this method you need to do the following.

1. Define a scanner object for keyboard input (i.e. connect it to *System.in*). Create a scanner for **`System.in`** only *once* (**only here as your 1-st step**) and use it for all **keyboard** inputs in this *main* method. **DON'T** re-define the scanner for ***System.in*** – **you will be penalized if you do**.
   Note: re-defining the scanner for ***System.in*** (i.e. defining another *new* object) will cause problems in grading (when your program is run with piped input), so DON'T do it.
2. Prompt the user to enter the following data in the following order: the number of coins (i.e. the value for *k*) and the list of k denominations (i.e. values for d_1, d_2, …, d_k) in **decreasing order**.
3. Input values from keyboard and save in respective variables: the first value in *k*, the remaining values in a *k*-length array of integers (the *d*-array). E.g., if input data represent k=5, and d={100,25,10,5,1} coin set, your input should be the following list: ***5 100 25 10 5 1*** and ***may be given in one line or in several lines (it should not matter).***
4. Prompt to enter a value for *n*: positive value if needs change, 0 to quit. Input the value for *n*.
5. **As long as *n*'s value is positive, repeat the following steps:**
   a) Invoke your *change_DP* method passing *n* and *d*-array as parameters. Save the address of the returned array so you can use it in the next step to output the results of the change.
   b) Arrange the output of the following info on the screen to represent money change results:
      - line 1 should show the amount to be changed: *n*'s value on
      - line 2 should show the change (the distribution of the amount) in the below given format
      - line 3 should show the number of coins used in the distribution

Example: for the input data k=5, d={100,25,10,5,1}, *n*=87, your output will look like this:

```
DP algorithm results
Amount: 87
Optimal distribution: 3*25c + 1*10c + 2*1c
Optimal coin count: 6
```

Attention: the formatting of line 2 of the data-output is as follows:
- you need to list coin-denominations in the order they appear in *d*-array
- for each used coin-denomination you need to indicate the *number of such coins*
- you need to use *, +, and *c* symbols in the distribution as indicated in the example

c) prompt the user to enter a value for *n* and input the value.

PART2: Design and implement a *greedy* algorithm for the Change Making problem.

I. Add a new *public* static method *change_greedy* in the *ChangeMaker* class to implement the *greedy* algorithm for the Change Making problem. This method has the following signature:

public static int[] change_greedy(int n, int[] d) //needs *n*-amount and the *d*-array of coin-values

This method receives the *n* amount and the *d*-array of coin denominations, and returns an array containing the ***count of coins* for each d_i-denomination** to be used when making change for the given *n* amount. Note: this is the *same* type result-array as returned by the *change_DP* method.

Example: if *d*-array contains 100,25,10,5,1 and *n*=87, then the return array will contain: 0,3,1,0,2 values respectively. These data indicate that there are 0 coins in d_1-denomination (i.e. no 100-cent coins), 3 coins of d_2-denomination (i.e. three 25-cent coins), 1 coin in d_3-denomination (i.e. one 10-cent coin), 0 coins in d_4-denomination (i.e. no 5-cent coins), 2 coins in d_5-denomination (i.e. two 1-cent coins). From the contents of this array it is easy to see that 6 coins were used.

Attention: to be as efficient as possible, i.e. O(*n*), the ***greedy algorithm*** for the Change Making problem will have the following ***pre-condition***: the *d*-array must be **sorted in decreasing order** of coin denominations (this will ensure that in every d[i..k] sub-list the first coin has the highest denomination). With this pre-condition, it will take only **O(1)** time to choose the highest denomination coin at each step of the *greedy* algorithm.

The ***greedy algorithm*** for the Change Making problem works as follows:
Start with *n* as "remaining" amount and repeatedly do the following **until remaining amount is 0**:
(1) choose the highest denomination coin less than or equal to the remaining amount (moving from left to right in *d*-array, find the first coin that meets this requirement). Assume it is the d_m-coin.
(2) figure out how many of d_m-denomination coins can "fit" in that "remaining" amount, and save that count in the result-array of *counts* for d_m-coin. Adjust/update the "remaining" amount.

II. Add a new segment in the *main* method of *ChangeMaker* class. On page 2, in bullet 5 of the *main* method, **after the b-step** but **before the c-step** add a new segment to do EXACTLY the same work as in 5-a and 5-b steps, **except** in this segment you will need to invoke *change_greedy* method instead of *change_DP* and in the output print *Greedy algorithm results* instead of *DP algorithm results* (thus in step 5 you have one loop that runs+outputs results of **both algorithms**). With this addition, the *main* method will prompt for and input data, and produce output as follows:
Example:

```
Enter the number of coin-denominations and the set of coin values:
5
100 25 10 5 1

Enter a positive amount to be changed (enter 0 to quit):
87

DP algorithm results
Amount: 87
```

```
Optimal distribution: 3*25c + 1*10c + 2*1c
Optimal coin count: 6

Greedy algorithm results
Amount: 87
Optimal distribution: 3*25c + 1*10c + 2*1c
Optimal coin count: 6

Enter a positive amount to be changed (enter 0 to quit):
233
DP algorithm results
Amount: 233
Optimal distribution: 2*100c + 1*25c + 1*5c + 3*1c
Optimal coin count: 7

Greedy algorithm results
Amount: 233
Optimal distribution: 2*100c + 1*25c + 1*5c + 3*1c
Optimal coin count: 7

Enter a positive amount to be changed (enter 0 to quit):
0
Thanks for playing. Good Bye.
```

III. Design a class called *Tester* that contains only one method, *main*.

//Do NOT change anything in the *ChangeMaker* class

In the *main* method make arrangements to run tests on **EACH one of the below given 5 sets** of coin-denominations, **200 tests per set** making change for *n*=1,2,3,…,200 amounts respectively. **In each test** run *change_DP* and *change_greedy* methods of *ChangeMaker* class and compare the *optimal coin counts* computed by each algorithm, keeping track of matching results (compare only optimal coin counts in distributions, not the actual distributions).

Arrange your program to produce the following output where instead of x1, x2, x3, x4, and x5 there will be actual numbers of matches that occurred during testing.

```
Testing change_DP and change_greedy algorithms
Testing set1:  x1 matches in 200 tests
Testing set2:  x2 matches in 200 tests
Testing set3:  x3 matches in 200 tests
Testing set4:  x4 matches in 200 tests
Testing set5:  x5 matches in 200 tests
```

Sets of coin-denominations to test:

Set1. US denominations: 100, 50, 25, 10, 5, 1

Set2. Soviet denominations: 100, 50, 20, 15, 10, 5, 3, 2, 1

Set3. Powers of 2: 64, 32, 16, 8, 4, 2, 1

Set4. US without the nickel: 100, 50, 25, 10, 1

Set5. Some set: 66, 35, 27, 18, 10, 1

Note: The DP algorithm's results are always optimal. As for greedy algorithm's results, the first three sets of coins are expected to give optimal solutions, but the last two are not.

Submitting your work:

Turn in *ChangeMaker* and *Tester* source files electronically via "*handin*" procedure by the deadline (see the top of the first page of this document for the due date and time):

The account *id* is: ***hg-cpe103***
The *name* of the assignment is: ***Project4***
The *name* of the assignment for late submissions is: ***Project4_late.***

Important requirements:

1. Your programs must **compile and run from command line** on our departmental servers. So, before submitting, make sure your programs compile and run on lab machines.
2. Do not create packages for your source code, do not zip, and do not use folders for your submission – all these complicate the automated grading process.
3. Each file must have a **comment-header** which should include both authors' **names** and **ids** (as in *id@calpoly.edu*), as well as the **date** and the **assignment name** (e.g. Project 4). **Reminder**: only **one** submission needs to be made for a pair/team.
4. The header of each file must **start at the first line** (no empty lines or *import* statements before it) and should be followed by at least one empty line before the first line of the code.