Lab 4: Dynamic Programming: Part 1.

Due date: Thursday, April 29, at the beginning of lab period.

Lab Assignment

Assignment Preparation

This is an **individual** lab. The goal of this lab to give you an opportunity to solve a few simple problems using dynamic programming techniques and to compare the running time and the quality of answers for dynamic programming and greedy algorithms.

The Task

You will implement dynamic programming solutions for the following problems:

- Making Change. For this problem, you will conduct a study that compares the dynamic programming technique to the greedy algorithm.
- Rod Cutting. You will implement the dynamic programming algorithm for solving this problem using three approaches:
  1. iterative bottom-up approach;
  2. recursive top-down approach *without memoization*.
  3. recursive top-down approach *with memoization*.

Note: Memoization and other techniques will be discussed during the Thursday, April 22 class.

Task 1: Making Change

You will implement a dynamic programming solution for the Making Change problem and will conduct a comparative study using this implementation and using your greedy algorithm from Lab 2.

Making Change Problem

The formulation of the problem is copied verbatim from your **Lab 2** handout.

Problem. A cashier at a store accepts money in payment for goods and needs to make change. The money comes in a finite number of fixed coin and bank note denominations. Assuming that the cashier has access to unlimited quantities of bank notes and coins of every denomination, the cashier needs to give change *using the smallest number of coins/bank notes* given a specific amount of money.

Dynamic Programming for Making Change Problem

Your first task is to implement and validate a dynamic programming method for making change. You will use the framework constructed in **Lab 2**. There, you were required to implement a Java class `MoneyChanger` according to the following specifications (repeated verbatim from the **Lab 2** handout).

- `int NCoins;` instances of the `MoneyChanger` class will have an `NCoins` field, which will store information about the number of different denominations of coins/bank notes used to make change.
- `int[] Money;` instances of the `MoneyChanger` class will have a `Money` field. This field shall contain the list of all available coin denominations for making change. sorted in **descending** order. For example, for U.S. currency, the array will contain the following values:

  { 10000, 5000, 2000, 1000, 500, 100, 25, 10, 5 ,1}[1]
- `MoneyChanger(int N)`: a constructor creating an empty `MoneyChanger` instance with N different currency denominations.
- `void setCurrencyArray(int[] Coins)`: sets the `Money` component of the `MoneyChanger` class to the `Coins` array passed into this method.
- `int[] makeChange(int amount)` is your implementation of the **greedy** algorithm for making change. It returns an array which stores, for each denomination from the `Money[]` array, the number of coins/notes of this denomination in the change returned.

[1] Assuming no $2 bills and no half-dollar coins, as their circulation is very limited. Also, assuming no $500 and $1000 or larger bills, as they are not in active circulation.

In addition to the `MoneyChanger` class, you should implement (outside of this class) a printChange() method (figure out the return type and the input parameters that fit your implementation best), which, given an array of change (e.g., returned by the `MoneyChanger.makeChange()` method) and an instance of the `MoneyChanger` class as input, will print the information about the change.

New Requirements. Modify your **Lab 2** solution as specified below:

(DP1.) Rename the `makeChange()` method to `makeChangeGreedy()`.

(DP2.) Make sure your `makeChangeGreedy()` method properly implements the greedy algorithm for making change (fix all previously known bugs, if any, and address any comments from the grader, once you get them).

(DP3.) Implement a new method, `int[] makeChangeDP()`, which will compute the change using the dynamic programming technique.

Please note, that for this assignment, you are allowed to choose your own implementation of the `makeChangeDP()` method. While the bottom-up iterative implementation will probably be the fastest, you are allowed to use any implementation, *as long as you acknowledge the type of the implementation you chose in your report and provide appropriate analysis of the efficiency/running time of your implementation.*

Comparative Study

Using your implementation of the dynamic programming and greedy algorithms for Making Change problem, you will conduct a comparative study that will study the following questions:

(Q1) Is it true that for some instances of the Making Change problem, a greedy algorithm is optimal?

(Q2) Is it true that for some instances of the Making Change problem, a greedy algorithm is NOT optimal?

(Q3) How likely is a greedy algorithm to provide an optimal solution for an instance of the Making Change problem?

(Q4) What is the difference between the optimal solutions (computed by the dynamic programming algorithm) and the greedy solutions to the Making Change instances?

(Q5) What is the difference between the running time behavior of the dynamic programming algorithm (as implemented by you) and the greedy algorithm?

The details for each question of the study are provided below.

Detecting Optimality of the greedy algorithm. We know (suspect?) that on some sets of coin denominations, a greedy algorithm can provide an optimal solution to the Making Change problem. Please note, that a problem instance can have **multiple different optimal solutions**, and that two different algorithms providing optimal solutions need not report the same one. On the other hand **all optimal solutions to the** Making Change problem **will have the same size, i.e., the number of coins** reported.

To find whether the greedy algorithm computed an optimal solution you will execute both the greedy and the dynamic programming algorithm on the same instance and then, *you will compare the number of coins reported by both methods*. If the number of coins is the same, your greedy algorithm has reported ***an optimal solution*** (even if the solution it reported is different from solution reported by your dynamic programming algorithm).

Using this trick, study the questions above as follows.

Questions Q1 and Q2. For these questions, use the following collections of coins:

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

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

CS denominations: 1, 2, 4, 8, 16, 32, 64.

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

The crazy set: 1, 10, 18, 27, 35, 66.

Note: It is expected that the first three collections of coins allow the greedy algorithm to return optimal solutions, while the last two - do not.

For each coin collection, conduct the following study (write the appropriate Java program that uses the `MoneyChanger` class instances):

1. For each change amount from 1 to 128, compute the change as given by the greedy and by the dynamic programming algorithms.
2. Check if the greedy algorithm reported an optimal solution.
3. At the end of the run, determine if the greedy algorithm **always** returned an optimal solution for the given collection of coin denominations.
4. Plot the sizes of the solutions returned by both algorithms. The amount of change is the independent variable, whereas the number of coins in the change is the dependent variable. (Graphs for coin collections on which the greedy algorithm is optimal will contain only one set of points. Graphs of coin collections on which the greedy algorithm is NOT optimal shall contain solution sizes for **both** the dynamic programming and the greedy algorithm).

5. Record your observations. **Notice**, that empirical evidence, such as you are collecting is NOT, by itself a proof that one some inputs the greedy algorithm is optimal. It is *merely evidence in support of this hypothesis*. Please make certain you make proper and defensible statements to this respect in your write-up.

Question Q3. How likely is a greedy algorithm to provide an optimal solution for an instance of the Making Change problem?

You will adapt your original test framework from **Lab 2** to study this question. In particular, your code will work as follows:

1. generate 200 random coin sets (sets of denominations), just like you did in **Lab 2**. Make sure that each coin set contains a coin with denomination 1 (to ensure that **any** amount of change can be given).
2. For each randomly generated coin set, run both the greedy and the dynamic programming method to determine solutions to the Make Change problem for **all amounts of change** from 1 to *double the maximum coin value*.

   For, example, if your coin collection contains coin denominations $1, 13, 45, 67, 109, 231$, then, you will find change for each amount from 1 to $2 \cdot 231 = 462$.
3. Using the technique discussed above, determine if there is evidence supporting the statement that the greedy algorithm provides an optimal solution for the currect set of coin denominations.
4. Report the percentage of coin sets you generated, on which you have empirical evidence that the greedy algorithm returns an optimal solution. Analyse your findings.

Note: If 200 runs is too small a number to properly evaluate the answer to this question (e.g., because one of the two cases is **very** rare), increase the number of trials by one-to-two orders of magnitude (i.e., to 2000 or 20000 randomly generated coin sets), and run the increased battery of tests as described above.

Question Q4. What is the difference between the optimal solutions (computed by the dynamic programming algorithm) and the greedy solutions to the Making Change instances?

You can collect the data that would allow you to answer this question while performing the study that addresses question **Q3** (see above). The information to be collected is described below:

- Average Difference. For each coin set, on which the greedy algorithm is not optimal, collect the difference between the greedy solution and the optimal solution (in terms of number of coins) for each change amount, *and find the mean* of the difference.

- Average Ratio. For each coin set, on which the greedy algorithm is not optimal, collect the ratio between the number of coins in the greedy solution and the optimal solution. (note: make the number of coins in the greedy solution the numerator, thus making your ratios above 1).

Report your findings. Plot the average distances and average ratios against each other. Plot each of these quantities against the number of coins in the coin set. What is the distribution of each of the two quantities computed? Discuss your findings.

Question Q5. You can measure the running time of each algorithm by simply computing the time each method gets executed. To conduct a proper efficiency study, you need to construct some large test cases.

Create random coin sets of sizes

5, 10, 20, 50, 100, 200, 500, 1000, 2000, 5000, 10000

(feel free to insert more data points in betwen)

For each size, generate 20 different coin collections. For each coin collection, run both the greedy and the dynamic programming algorithm on the following values:

- 100;
- 10000;
- 1000000;
- $2 \cdot Max - 1$, where Max is the largest denomination in the collection.

Compute the average running time per size of coin denomination collection for each of these four values. Plot the average running time against the size of the input to the Making Change problem computed as the number of integers. (note that for a 100-denomination coin collection, the size of input is actually 101, as the amount to change is also part of the size of input).

Task 2: Rod Cutting

Rod Cutting Problem: Given an integer N, the length of a metal rod and an array$P[1..N]$ of prices for rods of length $1..N$ respectively, output a set of numbers $l_1 \ldots, l_k$, such that $\sum_{i=1}^{k} l_i = N$ and

$$\sum_{i=1}^{k} P[l_i]$$

is maximized.

Your task. For this task, implement the dynamic programming algorithm for solving the Rod Cutting problem using three different approaches: *iterative bottom-up approach*, recursive top-down approach without memoization and recursive top-down approach with memoization. Each of the three approaches is discussed below.

Iterative Bottom-up approach. This implementation shall, compute, one-by-one the optimal ways to cut rods of sizes $1, 2, 3, \ldots$, and up until N: the problem input. The optimal value and the optimal solution for each size is saved in a data structure that is maintained throughout the process.

Recursive Top-down approach without memoization. This is, essentially (and strictly speaking), a divide-and-conquer implementation, rather than a dynamic programming one. In order to cut the rod of length N, your algorithm will represent N as all sums $N = M_1 + M_2$, and then recursively finds the optimal cuts for rods of lengths M_1 and M_2. There is no memoization, so, optimal cuts for rods of sizes less than N will be evaluated multiple times.

Recursive Top-down approach with memoization. This is the top-down version of dynamic programming. This is essentially a version of the recursive top-down approach described above, with the caveat that for each rod size, the optimal solution is **computed exactly once**, memoized and used in subsequent calls to the recursive method.

Implementation Details

You will implement a Java class `RodCutter`, which will contain the following features:

- `int[] Price`: an array of rod prices. `Price[i]` is the price of a rod of length i.
- `RodCutter(int[] P, int N`: initializes an instance of the `RodCutter` class using the array P as the list of rod prices. It is assumed that N is the size of the array `P`.
- `cutIterative()`: this method implements the *bottom-up iterative* version of the dynamic programming algorithm for solving the Rod Cutting problem. *You may select any return type that is convenient for your implementation.*
- `cutRecursive(int N)`: this method implements the *top-down recursive, no memoization* version of the dynamic programming algorithm for solving the Rod Cutting problem. *You may select any return type that is convenient for your implementation.*

- `cutRecursiveMem(int N)`: this method implements the *top-down recursive* algorithm for solving the Rod Cutting problem, which uses memoization. *You may select any return type that is convenient for your implementation.* (Note: make sure your `RodCutter` class definition has the data structures necessary to support memoization.)

Feel free to implement other features for this class. For debugging purposes, you shall also implement a `PrintSolution()` method, which prints the solution computed by any of your implementations of the solution algorithm (you get to choose what input parameters this method has).

Comparative Study

The key goal of your study is to compare the running time of your three implementations. You will collect two metrics: **the number of comparisons** it takes to compute the solution and **the running time of the solution algorithm**.

Number of comparisons. A *countable* comparison in your algorithm implementations is any executed comparison, which "touches" any value representing the cost of a specific fragment of a rod (e.g., the cost of a rod fragment of size $k < N$, where N is the full length of the rod). An *ignorable* comparison is any direct or implied comparison performed in the course of running a counter-driven loop.

Running time. Measure the exact time your program spends running your `cutIterative()`, `cutRecursive()` and `cutRecursiveMem()` methods.

Collect these two metrics as follows:

- Generate 2000 instances of the `Rod Cutting` problem. For each instance, remember its size $M = N + 1$: the length of the `Price[]` array plus one number to store the value N itself.
- Run all three methods (`cutIterative(), cutRecursive(), cutRecursiveMem()`) on each instance of the problem. Collect the two metrics.

Plot the two metrics vs. input size for each of the three algorithms used. Write a report, provide analysis of observed behavior.

Deliverables

This part of the lab has both electronic and hardcopy deliverables. All electronic deliverables shall be submitted by the assignment deadline using the following `handin` command:

Use `handin` to submit:

```
$ handin dekhtyar-grader lab04-349 <files>
```

The hardcopy deliverables shall be submitted to the instructor during the lab period on Tuesday, April 29.

Electronic Deliverables. Submit all the Java files you created/modified for this lab. Submit a README file with your name, and instructions for running your code for the grader. Submit a softcopy of your report (feel free to combine all your analysis into a single document) names `Lab04Report.pdf`

Hardcopy Submission. A hardcopy of your `Lab04Report.pdf` file should be submitted to the instructor before or during the lab period on April 29.

Note that we expect that all your programs will compile from the command line. Make sure you test for that.

Good Luck!