Homework 3: Dynamic Programming
Due: 5:00 PM, Mar 2, 2018

Contents

1 Travel Plans 2

2 Currency Exchange 4

Objectives

By the end of this homework, you will know:

- what dynamic programming is

By the end of this homework, you will be able to:

- apply dynamic programming to implement faster solutions to problems with recursive solutions than a naive recursive implementation

How to Hand In

For this (and all) homework assignments, you should hand in answers for all the non-practice questions. For this homework specifically, this entails answering the Travel Plans and Currency Exchange questions.

In order to hand in your solutions to these problems, they must be stored in appropriately-named files with the appropriate package header in an appropriately-named directory.

The source code files should comprise the hw03.src package, and your solution code files, the hw03.sol package.

Begin by copying the source code from the course directory to your own personal directory. That is, copy the following files from /course/cs0180/src/hw03/src/*.java to ~/course/cs0180/workspace/javaproject/src/hw03/src:

- ITravelPlans.java containing public interface ITravelPlans
- ICurrencyExchange.java containing public interface ICurrencyExchange

Do not alter these files!

After completing this assignment, the following solution files should be in your ~/course/cs0180/workspace/javaproject/sol/hw03/sol directory, all as part of the hw03.sol package when you hand in:
• Travel Plans

- AbsTravelPlans.java, containing public abstract class AbsTravelPlans implements ITravelPlans.
- TravelPlans.java, containing public class TravelPlans extends AbsTravelPlans
- TravelPlansTest.java, containing public class TravelPlansTest
- travelPlans.tex, which should contain your answers to the non-coding portions of the Travel Plans problem.

• Currency Exchange

- AbsCurrencyExchange.java, containing public abstract class AbsCurrencyExchange implements ICurrencyExchange.
- BUCurrencyExchange.java, containing public class BUCurrencyExchange extends AbsCurrencyExchange.
- CurrencyExchangeTest.java, containing public class CurrencyExchangeTest
- currencyExchange.tex, which should contain your answers to the non-coding portions of the Currency Exchange problem.

To hand in your files, navigate to the `/course/cs0180/workspace/javaproject/` directory, and run the command `cs018_handin hw03`. This will automatically hand in the contents of your entire javaproject directory. Once you have handed in your homework, you should receive an email, more or less immediately, confirming that fact. If you don’t receive this email, try handing in again, or ask the TAs what went wrong.

Problems

1 Travel Plans

Oh no! Hades is causing trouble and Hercules needs to save the people of Greece. However, to get to Hades, Hercules needs to travel from his home in Thebes to the Underworld. To complete this long journey he must stay at inns along his route. There are exactly $n$ inns that he may stay in, conveniently spaced at regular intervals of exactly 1 mile. The first inn is exactly 1 mile from Thebes. Hercules calculates that he can walk up to $d$ miles per day (inclusive) before he drops dead of exhaustion.

At every inn, Hades has placed a task that Hercules must complete if he wishes to stay there. However, some of Hades’ tasks are easier than others to complete. Killing the Hydra is very difficult but saving a farmer’s pig from under a wheelbarrow is much easier. Let $c_i \geq 0$ denote the amount of energy Hercules uses to complete task $i$ and thus stay at inn $i$.

Hercules wants to use the least amount of energy possible on his route to the underworld so that he has enough strength to fight Hades once he gets there. He opts to use computer science to plan his route and determine which tasks to complete (and thus which inns to stay at) so that he loses...
the least amount of energy. Hercules must stay at the last inn, to rest before he fights Hades in
the Underworld. He uses dynamic programming to find (1) the minimum amount of energy he
can expend on his journey from Thebes to the last inn and (2) the path of minimum energy loss (i.e., a
sequence of inns which minimize the sum of their tasks’ energy costs, where none of the inns are
greater than \( d \) miles apart. This sequence should always terminate with the last inn).

**Task:** Write a recurrence relation that expresses the total amount of energy Hercules uses while
traveling from Thebes to inn \( i \) (including the cost of inn \( i \)). Assume \( d \) and \( n \) are integers, with \( d > 0 \)
and \( n > 0 \). Assume that inn 0 (not 1) is the first inn – this may alter when the base case applies.
Put your answer in `travelPlans.tex`. *Be sure to define (in English!) any variables you use.*

**Note:** Remember that the first inn (inn 0) is 1 mile from Thebes!

**Task:** Create an abstract class `AbsTravelPlans` that implements `ITravelPlans` and do the
following to set it up.

1. Define a table (think array!) in which to store intermediate solutions. This should be defined
   as a field.

2. Write the signature for an abstract method `fillTable`, to be overridden in the concrete
classes that inherit from `AbsTravelPlans`, which will fill in the values of the table defined
above using dynamic programming.

3. Write the signature for a concrete method `tableInit`, which takes no parameters and returns
   nothing (is a void method).

4. Write the signature for a concrete method `optimalCost`, which takes no parameters and
   returns a double. Since this implements the `ITravelPlans` interface, remember to use the
   @Override decorator.

5. Write the signature for a concrete method `optimalPath`, which takes no parameters and
   returns an array of booleans. Since this implements the `ITravelPlans` interface, remember to
   use the @Override decorator.

**Task:** Write a constructor for `AbsTravelPlans`. This constructor should:

1. Take as input an integer `distance` and an array `innCosts` of doubles. The array will be of
   length \( n \), since there are \( n \) inns.

2. Check that the inputs are valid (i.e., `distance > 0`), and throw an `IllegalArgumentException`
   otherwise.

3. Create a new table in which to store optimal values, whose size is equal to the size of `innCosts`.

4. Call two methods:

   (a) `tableInit`. This method, which we started in the last task, should initialize the values
       in the table. This should cover the base case of the recurrence relation and set all other
       values to placeholders. For this problem (and the next one), your placeholder should be
       `Double.NEGATIVE_INFINITY`. You will have to fill in this method in `AbsTravelPlans`.

   (b) `fillTable`. You do not need to fill out this method in `AbsTravelPlans`, since it will
       be defined concretely by any class that extends it. Just call it for now.
Task: Create a TravelPlans class which extends AbsTravelPlans. Decide whether you would like to solve this problem using bottom-up or top-down dynamic programming. Depending on your decision, implement the fillTable method using your chosen strategy.

Task: In your AbsTravelPlans class,

- Fill in the method optimalCost that you defined earlier. This should return the energy used in the optimal path (that of minimum energy loss) from Thebes to the last inn.
- Fill in the method optimalPath that you defined earlier. This should return an optimal path of inns to stay in from Thebes to the last inn. This path should be represented by an array of booleans of size n, which indicates the locations of the inns where Hercules should stop. Specifically, if Hercules should stop at inn i, then the boolean value associated with inn i should be assigned the value true.

  Hint: The last inn should always be assigned the value true since Hercules must stay there.

Hint: Use the optimal values table that you have already filled in to write these methods.

Hint: Especially for the first part of this question, you should not be doing lots of extra calculations. If your approach seems complicated, rethink it!

Note: The optimalCost method is a function in the mathematical sense: there is always a unique minimum cost of traveling from Thebes to the Underworld, say $c^*$. However, the optimal path need not be unique. Indeed, there may be multiple selections of inns suitably spaced along the route whose total costs sum to $c^*$. Your optimalPath function need only return one such selection; all are equally valid solutions to this problem.

Task: As always, be sure to write a TravelPlansTest class to test your implementation!

2 Currency Exchange

Hiro Hamada (the hero of “Big Hero 6”) needs to raise money to repair his injured robot, Baymax (Hiro’s co-hero in the film). Right now, Hiro has only 100 units of Gold, which he won in a grant by doing science (spoilers?). But as a child genius, Hiro decides to have a go at currency arbitrage to take advantage of the fact that currency exchange rates often differ in different locations. Though an expert in engineering, robotics, and science, Hiro can’t figure out the most efficient way to exchange his money. He has called upon you and your dynamic programming expertise for help in saving Baymax! 

Suppose there are three currencies, Gold, US Dollars, and Pounds. You have researched the best exchange rates and stored them in a matrix (i.e., a two-dimensional array) rates that looks something like this:

<table>
<thead>
<tr>
<th></th>
<th>Gold</th>
<th>US Dollars</th>
<th>Pounds</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gold</td>
<td>1</td>
<td>0.66</td>
<td>0.77</td>
</tr>
<tr>
<td>US Dollars</td>
<td>1.53</td>
<td>1</td>
<td>1.16</td>
</tr>
<tr>
<td>Pounds</td>
<td>1.30</td>
<td>0.86</td>
<td>1</td>
</tr>
</tbody>
</table>

\[^1\]One might even say that you alone can ensure the success of “Big Hero’s” hero Hiro’s heroism
The value $rates[i][j]$ is the exchange rate for changing currency $i$ into $j$. So if $x_i$ is some number of units of currency $i$, then $x_j = x_i \times rates[i][j]$ is the number of units of currency $j$ that $x[i]$ units of currency $i$ can be exchanged for.

Note that entries along the diagonal in the $rates$ matrix—which specify how to convert from a currency into itself—need not be 1. Furthermore, $rates[i][j]$ is not necessarily the reciprocal of $rates[j][i]$. Because these are not reciprocal, if Hiro has 100 units of Gold, which he exchanges for Pounds, and then he changes those Pounds back into Gold, he can end up with more (or less) than 100 units of Gold.

For example, Hiro can exchange his 100 units of Gold into 77 Pounds ($(100)(0.77) = 77$). And then he can convert those Pounds back into Gold ($(77)(1.30) = 100.1$). And voila, he would have made an immediate profit for doing arguably no work at all!\[2\]

**Task:** Suppose you have a currency $i$ and a number of transactions $n$. Write a recurrence relation to calculate $T(i, n)$, which is the maximum amount of currency $i$ that can be had after $n$ transactions. A transaction is an exchange of one currency for another, and we start with (for concreteness) 100 units of currency $i$. For all $i \in \{1, \ldots, n\}$, Hiro must perform some transaction, but converting a currency into itself counts. Also, all the currency in his possession must be exchanged each time; Hiro cannot hold multiple currencies simultaneously. Put your recurrence in the currencyExchange.tex file. *Be sure to define (in English!) any variables you use.*

**Hint:** Remember that the final currency in Hiro's possession must be the same as the one he started out with.

**Hint:** You may assume that the $rates$ matrix is square, so that every currency can be converted into every other currency.

**Hint:** Think carefully about your base cases. There are two. Can you find them both?

**Task:** What would be the run time of a naive, recursive implementation of your recurrence relation? State your answer in big-$O$ notation, clearly explaining the meanings of all the variables you use. Give a brief justification; a formal proof is not necessary. Put this in currencyExchange.tex.

**Task:** Following our usual OOP class structure for dynamic programming, write an abstract class `AbsCurrencyExchange` that implements `ICurrencyExchange`. You should do the following to set it up.

1. Define a table in which to store intermediate solutions. This should be defined as a field.

2. Write the signature for an abstract method `fillTable`, which will be overridden in the subclasses of `AbsCurrencyExchange`.

3. Write the signature for a concrete method `tableInit`, which takes no parameters and returns nothing.

4. Write the signature for a concrete method `optimalValue`, which is specified in `ICurrencyExchange`. Make sure to add an @Override tag.

**Task:** Write a constructor for this class, which should take as input `numTrans`, an int representing the number of transactions, and `rates`, a matrix of doubles containing all the exchange rates. It should check the validity of these arguments, and throw an exception as relevant. After checking the

\[2\]That, my friend, is arbitrage.
validity of these inputs, it should then create a table of optimal values, call the initTable method to initialize this table, and then fillTable to fill in this table with the corresponding optimal values. This initTable method, as in the last problem, should initialize the values of the table with your base case values or Double.NEGATIVE_INFINITY.

Task: Create a BUCurrencyExchange class that extends AbsCurrencyExchange. In this class, you should override fillTable so that it fills in your optimal values table using bottom-up dynamic programming.

Task: What is the run time of fillTable? State your answer in big-O notation, clearly explaining the meanings of all the variables you use. Again, give a brief justification. Put your work in the currencyExchange.tex file.

Task: Implement the optimalValue method defined in ICurrencyExchange, which returns the maximum amount of Gold that Hiro can obtain after numTrans transactions, given that he begins with 100 units of Gold.

Task: Write a CurrencyExchangeTest class to test your implementation.

Please let us know if you find any mistakes, inconsistencies, or confusing language in this or any other CS18 document by filling out the anonymous feedback form: http://cs.brown.edu/courses/cs018/feedback