P06 Changemaker

Overview

In this assignment you will use **recursion** to solve a variation of the change-making problem. In this problem we pretend that we are a cash register dispensing change to a customer, where the goal is to find a combination of coins of different denominations that add up to a given amount, using the minimal total amount of coins, and where we have a limited supply of each denomination.

Learning Objectives

After completing this assignment, you should be able to:

- **Formulate** a recursive solution to a problem by describing the base cases, recursive cases, and how to decompose it into smaller subproblems
- **Implement** a recursive solution to a problem by making recursive calls to solve the subproblems, and combining the results
- **Compare** the recursive formulations of similar problems
- **Explain** why recursion can be a useful problem-solving tool
- **Develop** unit tests to verify the correctness of your algorithms

Grading Rubric

5 points	**Pre-assignment Quiz**: accessible through Canvas until 11:59PM on **03/27**.
18 points	**Immediate Automated Tests**: accessible by submission to Gradescope. You will receive feedback from these tests *before* the submission deadline and may make changes to your code in order to pass these tests. Passing all immediate automated tests does **not** guarantee full credit for the assignment.
15 points	**Additional Automated Tests**: these will also run on submission to Gradescope, but you will not receive feedback from these tests until after the submission deadline.
12 points	**Manual Grading Feedback**: TAs or graders will manually review your code, focusing on the recursive nature of your methods (4 points per recursive method).

Additional Assignment Requirements and Notes

- Pair programming is **ALLOWED** for this assignment. If you choose to work with a partner, you must review the partnership guidelines and declare your partnership before **11:59 PM** on **Sunday 03/26** using this form.

- The ONLY external libraries you may use in any of your classes are:
  `java.util.Arrays`
  Use of *any* other packages (outside of java.lang) is NOT permitted.

- You are allowed to define any **local** variables you may need to implement the methods in this specification (inside methods). You are NOT allowed to define any additional instance or static variables or constants beyond those specified in the write-up, except for public static helper methods.

- Only the `ChangemakerTester` class may contain a main method.

- All classes and methods must have their own Javadoc-style method header comments in accordance with the CS 300 Course Style Guide.

- Any source code provided in this specification may be included verbatim in your program without attribution.

- **Run your program locally before you submit to Gradescope**. If it doesn't work on your computer, *it will not work on Gradescope*.

Need More Help?

Check out the resources available to CS 300 students here:
https://canvas.wisc.edu/courses/344658/pages/resources

CS 300 Assignment Requirements

You are responsible for following the requirements listed on both of these pages on all CS 300 assignments, whether you've read them recently or not. Take a moment to review them if it's been a while:

- Academic Conduct Expectations and Advice, which addresses such questions as:
  - How much can you talk to your classmates?
  - How much can you look up on the internet?
  - What do I do about hardware problems?
  - and more!

- Course Style Guide, which addresses such questions as:
  - What should my source code look like?
  - How much should I comment?
  - and more!

Getting Started

1. Create a new project in Eclipse, called something like **P06 Changemaker**.
   a. Ensure this project uses Java 17. Select "JavaSE-17" under "Use an execution environment JRE" in the New Java Project dialog box.
   b. Do **not** create a project-specific package; use the default package.

2. Create two (2) Java source files within that project's src folder:
   a. **Changemaker.java** (does NOT include a main method)
   b. **ChangemakerTester.java** (includes a main method)

Note: all methods in this project will be **static**.

Implementation Requirements Overview

In this program you will be implementing a class `Changemaker` with three static methods, as well as a tester class `ChangemakerTester`. These three static methods **must be implemented using recursion**, either by directly calling itself (possibly more than once), or by utilizing a private static helper method which is recursive. **You may use a looping construct** such as a while-loop or for-loop in these methods, however you must still utilize recursion by setting up base cases, and recursive cases which make progress towards a base case via recursive call(s). We will be manually grading these methods to ensure you have done so properly.

Each of the three methods in the `Changemaker` class will solve a variant of the change-making problem. In this problem, we imagine that you have a cash register containing a certain amount of coins of various values, and that the goal is to return exact change for a specified value to a customer using these coins.

Changemaker

The `Changemaker` class is a static utility class. That is, it only contains three public static methods, along with any other private static helper methods that you choose to implement. You will not need to do any object-oriented programming in this assignment! These three methods will be focused on solving three different aspects of the change-making problem. Add these three methods to your `Changemaker` class and return the default values of `0` or `null` as appropriate for now.

```
public static int count(int[] denominations, int[] coinsRemaining, int value)

public static int minCoins(int[] denominations, int[] coinsRemaining, int value)

public static int[] makeChange(int[] denominations, int[] coinsRemaining,
    int value)
```

Method Parameters/Problem Representation

Note that all three methods in the `Changemaker` class have the same parameters. These parameters are used as follows to describe the specific change-making scenario to be solved.

- The `denominations` array describes the **value of each type of coin** in your register
- The `coinsRemaining` array describes the **quantity of each type of coin** in your register
- The `value` parameter describes the **total amount of change to be dispensed** to the customer

denominations

The `int[] denominations` parameter represents a list of various denominations or values of coins. We can assume that each denomination will have a strictly positive integer value (≥ 1), that no two denominations are identical, and that the list is not necessarily sorted in any order. For instance, in a scenario where we have coins of values 1¢ (1 cent), 5¢ (5 cents), 10¢ (10 cents), and 25¢ (25 cents), the `denominations` array could be `{1, 5, 10, 25}`. The array could also be in any other order such as `{10, 1, 25, 5}`, but for simplicity we will order the array from smallest to largest in all of our examples.

coinsRemaining

The `int[] coinsRemaining` parameter represents how many coins of each corresponding denomination in the `denominations` array are remaining in our cash register. We can assume that there is a non-negative integer quantity of each denomination of coin (≥ 0), and that `coinsRemaining` is the same length as `denominations`. For instance, in a scenario where we have five 1¢ coins, no 5¢ coins, two 10¢ coins, and three 25¢ coins (with the same `denominations` array as above), the corresponding `coinsRemaining` array would be `{5, 0, 2, 3}`. If the `denominations` array were rearranged, the `coinsRemaining` array would be rearranged in the same manner.

value

Finally, the `int value` parameter represents the total value that we wish to make change for. We do not have any assumptions for this parameter besides that it is an integer value, but note that if it is negative then there is no possible way to make change for such a value, and that if it is zero then there is precisely one way to make change by returning no coins. For instance, if we wish to make change for 27¢, we would set `value = 27`.

ChangemakerTester

You must also implement a tester class `ChangemakerTester` with the following six tester methods. The requirements for these tester methods will be described in more detail after describing the problems to be solved in the `Changemaker` class. Add these methods to your `ChangemakerTester` class and return a default `false` value for now.

```
public static boolean testCountBase()
public static boolean testCountRecursive()

public static boolean testMinCoinsBase()
public static boolean testMinCoinsRecursive()

public static boolean testMakeChangeBase()
public static boolean testMakeChangeRecursive()
```

Counting Ways to Make Change

In the `count()` method, you will determine the **number of possible ways to make change** for a given value with a limited number of coins of varying denominations as described by the parameters above. In this method **we do not care about the optimal way to make change** with the least number of coins, just the **total number of possible ways**. If there is **no way to make change** using the given coins (for instance if `value` is negative or there are not enough coins), your method should return `0`. Also note that if `value = 0`, there is precisely one way to make change by dispensing no coins.

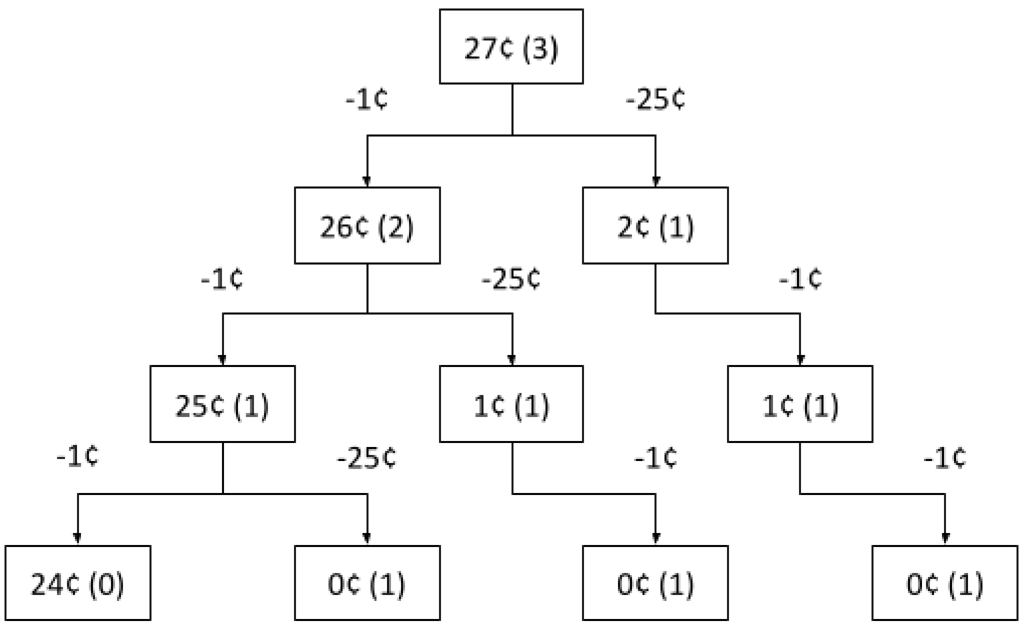
In this method we **care about the order that the coins are dispensed** in. For instance, if we dispensed a 1¢ coin followed by a 5¢ coin, this counts as a distinct possibility from dispensing a 5¢ coin followed by a 1¢ coin. For more intuition, think of a real-world cash register where the coins are dispensed one at a time and may be given to you in different orders.

The method should be implemented **recursively** by either directly calling itself (possibly more than once), or by calling a private static helper method which is recursive. **You may use a looping construct** such as a for-loop or while-loop in your implementation, however you must still implement your method recursively by calling the method within the body and making progress towards a base-case.

Examples

Consider a scenario where `denominations` = `{1, 5, 10, 25}`, `coinsRemaining` = `{3, 0, 0, 3}`, and `value` = `27`. In this case the count method should return `3`. This corresponds to the three different possibilities for dispensing two 1¢ coins and one 25¢ coin in one of three different orders: [1¢, 1¢, 25¢], [1¢, 25¢, 1¢], or [25¢, 1¢, 1¢].

To help you design your algorithm, below we provide an example of how the tree of recursive calls and return values for the `count` method in this scenario may look. Each rectangular node represents a call to the `count` method, and contains the `value` parameter of the method call on the left, and the return value of that call in parentheses on the right. The arrows indicate which recursive calls are made, and are labeled with the next coin that we have chosen to dispense.



Hints

Consider the following before starting to code your method:

- What are the base cases for this problem? Recursive cases?
- Which denominations can you choose for the first coin to dispense?
- What subproblems should you reduce the problem to by making a recursive call?
- Will the `denominations` array ever need to be modified?
- How should you update `coinsRemaining` when making a recursive call depending on which coin is dispensed? Be careful: arrays are mutable, so modifying `coinsRemaining` before making one recursive call will leave those changes in place for the next recursive call unless you revert them
- You may consider using a helper method `private static int[] useCoin(int[] coinsRemaining, int coinToUse)` which returns a new deeply copied (either manually or

using [`Arrays.copyOf()`]) array with the index of the selected denomination `coinToUse` decremented from `coinsRemaining`. This helper method is not required and the method can be implemented without it.

Minimum Coins Needed

Now we can move on to the more interesting part, where we determine the *optimal* way to make change with the `minCoins()` method. This method should return the **minimum total number of coins** needed to make change for the given value using a limited number of coins of various denominations as described before. This time, **if there is no way to make change** using the given coins your method should instead return `-1`, as a return value of `0` now indicates that it is possible to make change using no coins. As with the first method, this one must also be implemented **recursively**.

Examples

Consider the same scenario from the first example where `denominations` = `{1, 5, 10, 25}`, `coinsRemaining` = `{3, 0, 0, 3}`, and `value` = `27`. As we saw, there were different ways to make change by dispensing two 1¢ coins and one 25¢ coin in the three different orders of [1¢, 1¢, 25¢], [1¢, 25¢, 1¢], or [25¢, 1¢, 1¢]. As all of these solutions use exactly **three coins in total**, and there is no way to make change using only two coins, your `minCoins()` method should return `3`.

As another example, when `denominations = {1, 5, 10, 25}`, `coinsRemaining = {5, 1, 3, 1}`, and `value = 59`, your `minCoins()` method should return `8`, as the way to make change using the minimum total number of coins is with one 25¢ coin, three 10¢ coins, and four 1¢ coins. If instead we had `coinsRemaining = {59, 12, 6, 3}` and `value = 59` (so that we can use as many coins of each denomination that we want), then your `minCoins` method should return `7`, as now the way to make change using the minimum total number of coins is with two 25¢ coins, one 5¢ coin, and four 1¢ coins.

Finally, if `denominations = {1, 5, 10, 25}`, `coinsRemaining = {3, 6, 3, 2}`, and `value = 29`, your `minCoins()` method should return `-1`, as there is no way to make change using the given coins because no combination will add to exactly `29`.

Hints

Beware, you cannot always just choose the largest denomination available! If you do, you might either falsely conclude that you cannot make exact change, or might not make change optimally. For example, consider the case when `denominations = {1, 5, 6, 9}`, `coinsRemaining = {11, 3, 2, 2}`, and `value = 11`. If you greedily choose to use the larger 9¢ coin first, you would then be required to use two 1¢ coins, requiring a total of three coins. However, if you choose one 6¢ coin and one 5¢ coin, then you would only use two coins in total, so `minCoins()` should actually return `2` in this case[1].

This method can be implemented similarly to how you implemented `count()`, except that we now care about finding the *minimum* number of coins, rather than the *total* number of possibilities. Be careful about handling the case when a recursive call returns `-1`, as this indicates that there is no way to make change for that specific subproblem, and hence cannot be considered when finding the minimum number of coins. It may help to review how you would find the smallest *non-negative* value in an array.

Making Change

The `minCoins()` method computed the optimal number of coins needed to make change, but it's not very practically useful as it doesn't tell us the actual number of coins of each denomination that were used! In the final method `makeChange()`, you will fix this problem by computing an **array representing the exact number of each type of coin needed to make change optimally** so that you can dispense them to your customer. If there is **no way to make change** using the given coins you should return a `null` array (**not** an array of length 0 or an array containing only `0`). If `value = 0`, you should return an array filled with `0` to indicate that change can be made using no coins. This method must also be implemented **recursively**.

Specifically, this method should return an **array of the same length as** the `denominations` and `coinsRemaining` arrays, where the value at index `i` represents how many coins of the corresponding value `denominations[i]` were selected. For instance, if the `makeChange` method is called with `denominations = {1, 5, 10, 25}` and returns the array `{2, 0, 3, 1}`, this means that two 1¢ coins, no 5¢ coins, three 10¢ coins, and one 25¢ coin were selected.

If there are **multiple different ways to make change using the same optimal total number of coins**, you can return an **arbitrary solution among them**. For instance, consider the case when `denominations = {1, 5, 7, 11}`, `coinsRemaining = {12, 3, 2, 2}`, and `value = 12`. There are two optimal ways of making change using either a 1¢ coin and an 11¢ coin, or a 5¢ coin and a 7¢ coin. In this case, your `makeChange` method should return either of `{1, 0, 0, 1}` or `{0, 1, 1, 0}` corresponding to one of these two solutions. You may choose to return whichever solution is more convenient based on your implementation.

Examples

Consider the scenario where `denominations` = `{1, 5, 10, 25}`, `coinsRemaining` = `{3, 0, 0, 3}`, and `value` = `27`. To use the minimum number of coins we should dispense two 1¢ coins and one 25¢ coin, so in this case `makeChange()` should return the array `{2, 0, 0, 1}`.

---

[1] For those optionally interested, for the standard US coin denominations of 1¢, 5¢, 10¢, and 25¢ the greedy choice will always work, and [there is a way to determine whether the greedy choice will work]() for an arbitrary set of denominations

When `denominations = {1, 5, 10, 25}`, `coinsRemaining = {5, 1, 3, 1}`, and `value = 59`, your `makeChange()` method should return `{4, 0, 3, 1}`, as the way to make change using the minimum total number of coins is with four 1¢ coins, three 10¢ coins, and one 25¢ coin.

When `denominations = {1, 5, 10, 25}`, `coinsRemaining = {59, 12, 6, 3}` and `value = 59` (so that we can use however many coins of each denomination that we want), then your `makeChange()` method should return `{4, 1, 0, 2}`, as now the way to make change using the minimum total number of coins is with four 1¢ coins, one 5¢ coin, and two 25¢ coins.

Finally, if `denominations = {1, 5, 10, 25}`, `coinsRemaining = {3, 1, 3, 1}`, and `value = 29`, your `makeChange()` method should return `null`, as there is no way to make change using the given coins because no combination will add to exactly `29`.

Hints

This method can be implemented similarly to how you implemented `minCoins()`, except that we additionally care about the array of coins used to make change optimally, rather than just the number of coins.

As this method only returns the array representing the number of coins of each denomination rather than the total number of coins in a solution, you may consider using a helper method `private static int sum(int[] coins)` which returns the sum of all values in an array to determine the total number of coins used in a solution.

ChangemakerTester

In the tester class `ChangemakerTester`, you will be required to implement six tester methods, with two test methods dedicated to each of the three methods you implemented in the `Changemaker` class.

Testers for count

The test methods `testCountBase()` and `testCountRecursive()` are dedicated to testing the `count()` method. `testCountBase()` should implement at least the following three scenarios for determining whether the `count()` method behaves correctly on a base case of the problem. Similarly `testCountRecursive()` should implement at least the following three scenarios for determining whether the `count()` method behaves correctly on a recursive case of the problem.

These scenarios do not need to be extremely complex, but you should carefully consider which scenarios you use and verify by hand that your expected answers are correct, as you **may re-use the same scenarios in the other tester methods**. However, for integrity purposes **do not use any of the exact scenarios described earlier in this writeup** (you can instead modify them). Below we also provide general examples which you may choose to use by filling in your own chosen values.

testCountBase

1. `count()` returns `0` when `value` is negative
2. `count()` returns `0` when `value` is positive but there is no way to make change. You can create such a scenario by choosing the sum total of all the coins in the register to be smaller than `value`.
3. `count()` returns `1` when `value = 0`

testCountRecursive

1. `count()` returns the correct result in a scenario in which at least **three different coins** can be used to make change. For instance, when `denominations = {1, 5, 10, 25}`, `coinsRemaining = {1, 1, 1, 1}`, and `value = 36` we can use a 25¢ coin, a 10¢ coin, and a 1¢ coin. The expected answer in this case should be `6`, as there are six different ways to order the choice of three coins: [1¢, 10¢, 25], [1¢, 25¢, 10¢], [10¢, 1¢, 25], [10¢, 25¢, 1], [25¢, 1¢, 10¢], and [25¢, 10¢, 1¢].
2. `count()` returns the correct result in a scenario in which there are **at least two different optimal ways to make change** using the same number of coins. You can use any such scenario you can think of, but one simple scenario you can use that will always work is to choose four values `denominations = {x, y, z, w}` where `x + w = y + z`, and let `coinsRemaining = {1, 1, 1, 1}` and `value = x + w`. In this case the two optimal solutions are to choose either `x` and `w`, or `y` and `z`, where each solution requires two coins. For instance, when `denominations = {2, 5, 7, 10}` and `value = 12`, the two possible optimal solutions are to choose a 2¢ and 10¢ coin, or a 5¢ and 7¢ coin. The expected answer in this case should be `4`, as there are two different ways to order the choice of the 2¢ and 10¢ coins, and two different ways to order the choice of the 5¢ and 7¢ coins.
3. `count()` returns the correct result in a scenario in which always **greedily choosing the largest coin first does not produce a result with the minimum number of coins** used. Similar to the example for the last scenario, we can create such a scenario by choosing four values `denominations = {1, y, z, w}` where `w + 2 = y + z`, and let `coinsRemaining = {2, 1, 1, 1}` and `value = w + 2`. In this case the optimal solution is to choose `y` and `z` for a total of two coins, and the greedy solution is to choose `w` and two 1¢ coins for a total of three coins. For instance, when `denominations = {1, 5, 6, 9}` and `value = 11`, the optimal solution is to choose the 5¢ and 6¢ coin, but the greedy solution is to choose the 9¢ coin and two 1¢ coins. The expected answer in this case should be `5`, as there are two different ways to order the choice of the 5¢ and 6¢ coins, and three different ways to order the choice of the one 9¢ two 1¢ coins.

Testers for minCoins

The test methods `testMinCoinsBase()` and `testMinCoinsRecursive()` are dedicated to testing the `minCoins()` method, and are **identical to the earlier required tester method scenarios except for which method they test**. You may **re-use the exact scenarios you used in the previous tester method**, with the expected results now being the minimum total number of coins needed rather than the total number of ways of making change.

testMinCoinsBase

1. `minCoins()` returns `-1` when `value` is negative
2. `minCoins()` returns `-1` when `value` is positive but there is no way to make change
3. `minCoins()` return `0` when `value = 0`

testMinCoinsRecursive

1. `minCoins()` returns the correct result in a scenario in which at least three different coins must be used to make change. In the example given in scenario 1 from `testCountRecursive()` the expected answer is 3.
2. `minCoins()` returns the correct result in a scenario in which there are at least two different ways to make change using the same optimal number of coins. If you used the provided scenario 2 from `testCountRecursive()`, the expected answer is 2.
3. `minCoins()` returns the correct result in a scenario in which always choosing the largest coin first does not produce a result with the minimum number of coins used. If you used the provided scenario 3 from `testCountRecursive()`, the expected answer is 2.

Testers for makeChange

Finally, the test methods `testMakeChangeBase()` and `testMakeChangeRecursive()` are dedicated to testing the makeChange method, and are identical to the earlier tester methods except for which method they test. You may **re-use the exact scenarios you used in the previous tester methods**, with the expected results now being an array representing the number of coins of each denomination needed to make change with the fewest total number of coins.

Important: Remember that there may be multiple different results for `makeChange()` which are all correct, and your tester methods should accommodate this. That is, your tester method should return true if **any correct array is returned**, not just if a specific array or the array that your implementation returns is the actual result. Consider using the [`Arrays.equals()`] method to determine if two integer arrays are equivalent.

testMakeChangeBase

1. `makeChange()` returns `null` when `value` is negative
2. `makeChange()` returns `null` when `value` is positive but there is no way to make change
3. `makeChange()` returns an array of all `0` when `value = 0`

testMakeChangeRecursive

1. `makeChange()` returns an optimal array in a scenario in which at least three different coins must be used to make change
2. `makeChange()` returns an optimal array in a scenario in which there are at least two different ways to make change using the same optimal number of coins. If you used the provided scenario 2 from `testCountRecursive()`, the expected answer is either of `{1, 0, 0, 1}` or `{0, 1, 1, 0}`.
3. `makeChange()` returns an optimal array in a scenario in which always choosing the largest coin first does not produce a result with the minimum number of coins used. If you used the provided scenario 3 from `testCountRecursive()`, the expected answer is `{0, 1, 1, 0}`.

Assignment Submission

Once you're satisfied with your work, both in terms of adherence to this specification and the academic conduct and style guide requirements, submit your source code through Gradescope. For full credit, please submit ONLY the following files (source code, *not* .class files):

- Changemaker.java
- ChangemakerTester.java

Your score for this assignment will be based on the submission marked "**active**" prior to the deadline. You may select which submission to mark active at any time, but by default this will be your most recent submission.

Copyright Notice