Instructions	Implement a RecursiveWithdraw strategy for the Purse, that uses recursion to decide what items to withdraw. Write JUnit tests that include cases where (a) greedy withdraw fails but withdraw is possible (recursion should succeed), (b) edge cases to verify recursion is correct, including some case where withdraw should fail. The JUnit test succeeds if recursion fails, that is, the code is correct!
What to Submit	Add the code to your coinpurse project and push it to Github.

Withdraw Strategies

The original withdraw algorithm for the Purse tries to remove the highest value item first, then add smaller items until it gets the requested amount.

Unfortunately, it doesn't always work. Suppose the Purse contains these coins:

5-Baht, 2-Baht, 2-Baht, 2-Baht

if we try to withdraw 6-Baht, the greedy withdraw algorithm fails. There are *many, many* cases like this, so don't try to fix withdraw with some *cludgy* code like "if it fails then I'll ignore the first item and try again".

A strategy that will *always* find a solution (if it exists) is recursion. Recursion can be time consuming, but the Purse typically does not contain many items so it should be feasible.

Problem 1: Write JUnit Tests for Failed Withdraws

1.1 Add **at least 2 withdraw tests** to the PurseTest class to test cases where a withdraw should be possible, but the greedy withdraw fails.

1.2 Also add **at least one test** for some **edge case** where recursion might make a mistake, such as a deep recursion, or a code that forgets to check the currency, e.g. Purse contains 2-Baht, 2-Cats, 2-Baht and try to withdraw 6 Baht. The JUnit test should pass when withdraw does not succeed (that is, code is correct).

Another good edge case is something that requires withdraw to alternately pick an item and skip an item, to check for misplaced if - else logic.

Problem 2: Implement a Recursive Withdraw Strategy

Write a RecursiveWithdraw strategy to compute a withdraw using *recursion*. The algorithm is similar to the **`groupSum`** problem on **`codingbat.com`**.

2.1 Write a RecursiveWithdraw strategy class in the coinpurse.strategy package.

2.2 Repeatedly test your code using unit tests and command line. If you discover new test cases that failed during command line testing, create a JUnit test for that case.

2.3 **Perform Code Review**: When your code passes all tests, take a break and then perform *Code Review*. Read your RecursiveWithdraw code line-by-line and explain to yourself what each line does. Try to find bugs that testing missed. If you find any "missed" bugs, create a new JUnit test that does detect it.

2.4 Don't *hard-code* the RecursiveWithdraw object into Purse. In the main class call purse.setWithdrawStrategy() give the Purse a reference to a RecursiveWithdraw object.
The word "RecursiveWithdraw" should not appear *anywhere* in the Purse class!

Programming Hints

1. Use Recursion to choose which items to withdraw by examining one item from the money list (first item or last item), and then recursively call withdraw() to try to find the remaining amount using other items in the list.

2. For the recursive step, create a *sublist* of the current list of Valuable that excludes the one item you are withdrawing (or not withdrawing) at this step:

```
List<Valuable> sublist = list.subList( start, end );
```

list.subList(start, end) creates a *view* of the list starting at index **start**, and up to (but not including) index **end**. It is a *view,* not a copy. This is efficient -- no copying of the List. If the List is modified, the view will change, too. In Jshell:

```
> List<String> fruit = List.of("apple", "banana", "fig", "grape");
> fruit.subList(1,4)
["banana", "fig", "grape"]
> fruit.subList(0,2)
["apple", "banana"]
```

subList is a useful method and it really is a view, not a copy, of the list. If you change the List then the view changes. If you change the view, it changes the list (so be careful).

3. The recursive method should use the typical recursion pattern:

```
List<Valuable> withdraw(amount, currency, money):

// 1. Base Case:  Did the withdraw succeed? Are there any elements in the list?
//    If withdraw succeeds, create a new List and return it, else return null.

// 2. Recursive Step. select one item in the list, and apply recursion
Valuable first = money.get(0);  // or last item, if you prefer that

// 2.1 include this item in the money to witthdraw (if possible).
// Try to withdraw the remaining amount using the rest of the money list.
// Some code is missing here:
remaining = amount - first.getValue();
List<Valuable> result = withdraw( remaining, currency,
                                  money.subList(1,money.size()) );
// Did the recursive withdraw succeed?

// 2.2 don't use this item for withdraw.
// If Case 2.1 didn't succeed or currency of first item didn't match,
// try to withdraw the entire amount using the other items in the list.
```

4. **Don't create a new List (for the return value) at each recursive step!** Only create a new list for the solution in the **base case**, when you decide if you have found a solution. Higher level callers will *append* their item to this return list. Avoid unnecessary computation or object-creation in the recursive step.

How to See What RecursiveWithdraw is doing?

The withdraw strategy object doesn't print anything, so it is hard to debug.

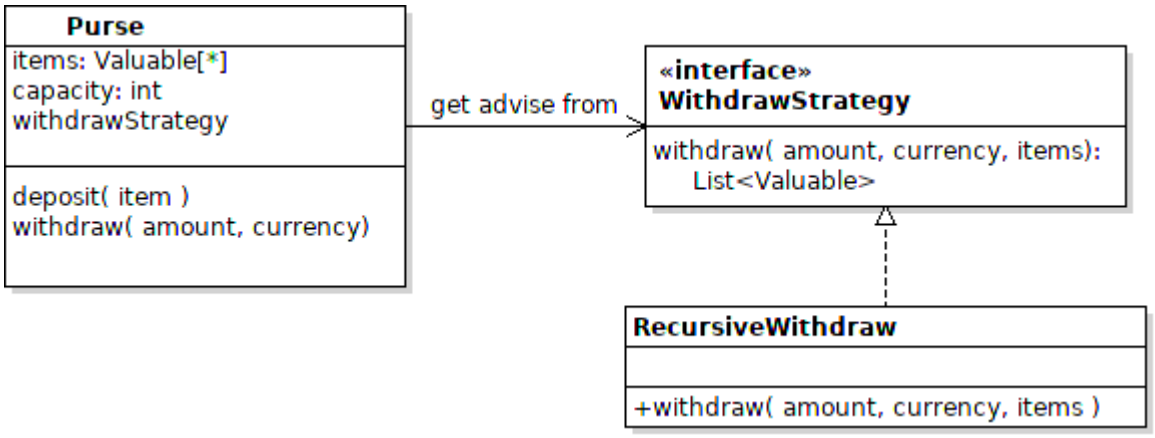
You could add code at the beginning and before every return to print something. To keep you code clean, write methods named enter() and leave() to print whatever is useful to you. For example:

```
public List<Valuable> withdraw( ... ) {
    enter(amount, currency, money);
    // do something
    List<Valuable> result = ...
    // call leave before returning.  leave() will return its parameter
    return leave(result);
}
//TODO: write enter() and leave() to print a message on console.
```

Reference

Big Java chapter 13 covers Recursion.

codingbat.com "Recursion-2" problem set covers recursion with backtracking. The **`groupSum`** problem is exactly like this one.



Example of Recursive Withdraw

Here is an example recursive withdraw using a list of numbers. To simplify the example, plain numbers are used.

withdraw(amount, list) - try to withdraw amount from list of numbers. Returns a List of elements to withdraw.

"Case 1" means "use first list item as part of the withdraw, and recursively withdraw the remaining amount"

"Case 2" means "don't use first list item as part of the withdraw, recursively withdraw the entire amount"

"Base Case" means the base case for recursion is used to decide what to return.