P06 Suitcase Packing

Overview

It's almost time for Spring break! In order to celebrate, this assignment will help you learn how to efficiently pack a suitcase using recursion.

Pretend that over your vacation you will be taking a flight with the imaginary Soul Airlines©, which charges an extortionate fee for each bag that you will bring on the plane. In order to minimize how much you have to pay, in this assignment you will write **recursive algorithms to fill as much area in a suitcase as possible with items** using various different strategies.

Grading Rubric

5 points	**Pre-assignment Quiz**: accessible through Canvas until 11:59PM on 3/17.
+5%	**Bonus Points**: students whose ***final*** submission to Gradescope is before **5:00 PM Central Time** on **WED 03/20** *and* who pass ALL immediate tests will receive an additional 2.5 points toward this assignment, **up to a maximum total of 50 points**.
16 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.
14 points	**Manual Grading Feedback**: TAs or graders will manually review your code. For this assignment, each recursive method will be graded manually on whether it is implemented recursively ***AND*** equal weight will be given to **inline commenting inside the method**. Be sure to explain your algorithms thoroughly!
50 points	**MAXIMUM TOTAL SCORE**

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

Additional Assignment Requirements and Notes

Keep in mind:

- Pair programming is **ALLOWED** for this assignment, BUT you must have registered your partnership before this specification was released. If you did not do so, you must complete this assignment individually.

- The ONLY external libraries you may use in your program are:
  ```
  java.util.ArrayList
  ```
  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.

- All methods in the **Packing** class must be **static**. You are allowed to define additional **private static** helper methods.

- Only the **PackingTester** 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.

- All other sources must be cited explicitly in your program comments, in accordance with the Appropriate Academic Conduct guidelines.

- Any use of ChatGPT or other large language models ***must be cited*** AND your submission MUST include a file called `log.txt` containing the full transcript of your usage of the tool. Failure to cite or include your logs is considered academic misconduct and will be handled accordingly.

- **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/375321/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!

1. Getting Started

1. Create a new project in Eclipse, called something like **P06 Suitcase Packing**.
   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. Download **six (6)** Java source files from the assignment page on Canvas:

   a. **Item**.java
   b. **Position**.java
   c. **Suitcase**.java
   d. **Packing**.java
   e. **Utilities**.java
   f. **PackingTester**.java

All methods in this program that you implement will be **static** methods, as this program focuses on procedural programming using recursion. The provided classes all contain JavaDoc comments, and both Packing and PackingTester contain TODOs to help you implement the assignment. This assignment will not require *many* lines of code, but will require careful thought and recursive thinking.

2. Overview of Problem and Provided Classes

In this program you will be implementing three (3) static methods in the **Packing** class, as well as a tester class **PackingTester**. The three static methods in the Packing class ***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 which immediately return, 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 implemented them **recursively**. We will also be **manually grading** these methods for **informative inline comments which explain your algorithmic and recursive thinking**. Be thorough!

Each of the three methods in the Packing class will solve a variant of the **suitcase-packing problem**:

Given:
- a list of rectangular items of various sizes to pack, and
- a rectangular suitcase with a given width and height

Your methods will:
- **pack as much of the suitcase as possible** with these items, without overlapping any items.

For simplicity, we will assume that you ***cannot*** rotate the items, and that there is only one position that each item can be added in, given the positions of all the previously added items. This position will be computed for you by the Suitcase class.

2.1 Item Class

The first provided class, **Item**, represents a rectangular item to be packed, including its width, height, and name. This class's fields are all public final fields, so once you create an Item you cannot modify it, and you can publicly access the fields directly without a getter method. You are not required to implement any methods in this class, but you may find it helpful to review their implementation and documentation.

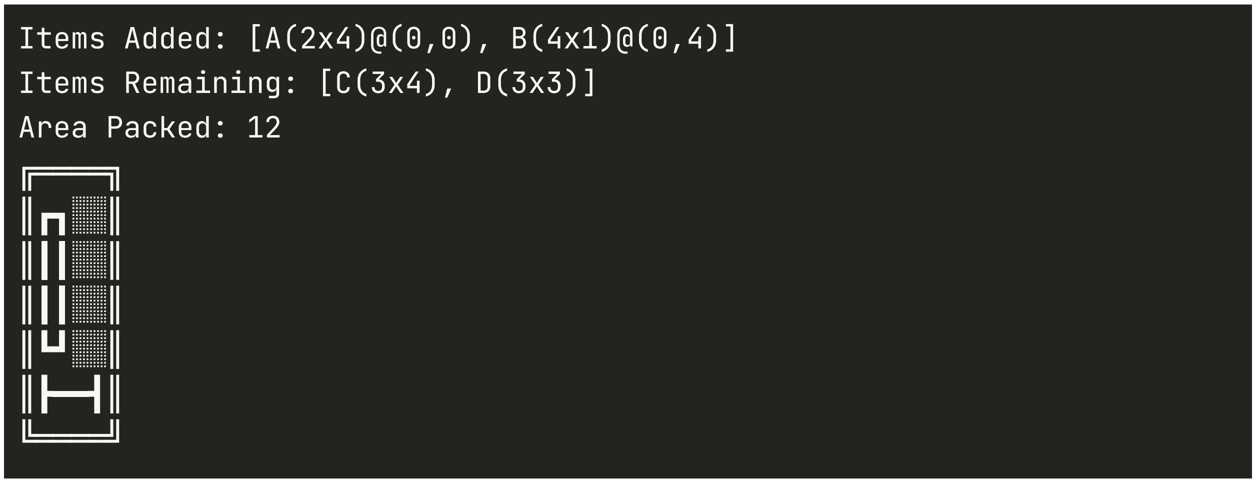
2.2 Position Class

The second provided class, **Position,** represents a discrete x,y coordinate pair in the suitcase that each item can be packed at. For instance, as in a multidimensional array, an item can be packed at position `(3,2)`, but not at `(3.1,2.5)`. Like the Item class, this class's fields are also public final fields. You are not required to implement any methods in this class, and you will not need to use this class directly.

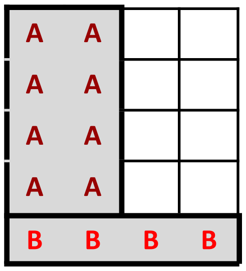
2.3 Suitcase Class

The provided **Suitcase** class represents both the contents of a suitcase, and a list of items that have yet to be packed. You do not need to implement any methods in this class, but you should fully read the JavaDocs in order to understand the methods available to you, as they will be very important in order to complete the main recursive methods.

There are two things that the Suitcase class tracks: **which items have been packed** in the suitcase and at what positions, and **which items have not yet been packed**. For instance, here is the `toString()` output of a 4 x 5 (width 4 and height 5) Suitcase object with two packed items, "A" of size 2 x 4 at (0,0) and "B" of size 4 x 1 at (0,4), and two unpacked items, "C" of size 3x4, and "D" of size 4 x 1.


```
Items Added: [A(2x4)@(0,0), B(4x1)@(0,4)]
Items Remaining: [C(3x4), D(3x3)]
Area Packed: 12
```

The text **visualization may look narrower than you may expect** for a 4 x 5 suitcase, as each character is taller than it is wide. The dotted "░" characters within the visualized suitcase represent empty space that does not contain any item, and the filled spaces such as "├─┤" represents items of varying shapes and sizes. You can look at the JavaDoc comments for the `Suitcase.toString()` method for a complete description of how the visualization is generated. In this writeup, we will visualize the same suitcase like this:



A very important aspect of the Suitcase class is that **each method which modifies the state of the suitcase will return an *entirely new,* cloned Suitcase object on which the changes will be made, preserving the state of the old object**. You can think of Suitcase objects similarly to immutable objects such as Strings. This design may seem overly-complicated, but it has been designed to aid you in implementing the main recursive methods.

When an item is packed, it will be placed at the left-top-most position in which it fits in the Suitcase. This corresponds to the strategy of starting from the top-left corner of the grid, and sliding the item top-to-bottom, and after each column is completed moving one position to the right, until an open position is found. You can read the private helper method `Suitcase.packPosition()` to see this in action.

For an example of the behavior of the Suitcase class, consider the following code:

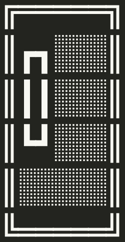
```java
Item phone = new Item("Phone", 2, 3);
Item shirt = new Item("Shirt", 2, 2);

ArrayList<Item> items = new ArrayList<>();
items.add(phone);
items.add(shirt);

Suitcase s1 = new Suitcase(5, 4, items);

System.out.println("initial s1:\n" + s1);

Suitcase s2 = s1.packItem(phone);

System.out.println("added s1:\n" + s1);
System.out.println("s2:\n" + s2);

Suitcase s3 = s2.packItem(shirt);

System.out.println("s3:\n" + s3);
```

When we call `s1.packItem(0)`, no changes will be made to the original suitcase `s1`, but the returned suitcase `s2` will have the phone item added to it at the top-left position `(0,0)`, as seen in the following output generated by the above code. Note that on the second time that `s1` is printed, `s1.packItem(0)` has been called, but the suitcase object is unchanged.

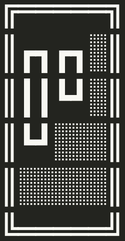
Lastly, note that all items are packed at the left-top-most position into which they fit, even if other positions are available.

```
initial s1:
Items Added: []
Items Remaining: [Phone(2x3), Shirt(2x2)]
Area Packed: 0
╔═════╗
║░░░░░║
║░░░░░║
║░░░░░║
║░░░░░║
╚═════╝
added s1:
Items Added: []
Items Remaining: [Phone(2x3), Shirt(2x2)]
Area Packed: 0
╔═════╗
║░░░░░║
║░░░░░║
║░░░░░║
║░░░░░║
╚═════╝
s2:
Items Added: [Phone(2x3)@(0,0)]
Items Remaining: [Shirt(2x2)]
```

```
Area Packed: 6
```



```
s3:
Items Added: [Phone(2x3)@(0,0), Shirt(2x2)@(2,0)]
Items Remaining: []
Area Packed: 10
```



The most important methods to familiarize yourself with in the Suitcase class are:

- the **constructor**, which initializes an empty suitcase of the given size and (a deep copy of) the given list of items to pack,
- the **`getUnpackedItems()`** and **`getPackedItems()`** methods, which return (a deep copy of) the list of currently unpacked and packed items,
- the **`canPackItem()`** method, which determines whether a given item fits into the Suitcase, and
- the **`packItem()`** method, which packs an item into a **new copy of the Suitcase**.

2.4 Packing Class

The **Packing** class is the main class of this assignment. There are three static recursive methods that you will implement:

1. `public static Suitcase rushedPacking(Suitcase suitcase)`
2. `public static Suitcase greedyPacking(Suitcase suitcase)`
3. `public static Suitcase optimalPacking(Suitcase suitcase)`

These methods will solve the problem of packing as much area of the suitcase as possible, each using a different strategy. It's very important (for your grade AND your skillset) that the **methods are recursive**, and that you add **sufficient inline comments to explain your algorithm**.

In order for a method to qualify as **recursive**, it must **either call itself** (possibly multiple times) – or make a call to a **private static helper method which calls itself** – as an integral part of building its solution.

Your inline comments should be detailed enough to be able to ***fully* explain your thought process for the algorithm to another student in the class**, including details like:

- the purpose of a non-trivial control-flow structure (e.g. if-statement, for-loop, break, continue),
- what your base cases are,
- how you reduce a recursive case to smaller cases, and
- how you combine the results of those smaller cases into the solution to the larger problem.

IMPORTANT: Implementations which are not recursive and/or which do not have sufficient explanatory inline comments ***will lose*** (a lot of) ***points***!

2.5 PackingTester Class

In the **PackingTester** class you will implement five different test methods, including one which is a randomized test method. In order to implement the testers you will first need to understand the three packing strategies, so wait until Section 6 for more info on the tester class.

2.6 Utilities Class

The **Utilities** class contains methods which will help you generate random test cases for PackingTester. The `randomItems()` method creates an ArrayList containing Items of random sizes, and the `randomlyPack()` method will randomly pack a suitcase with items from the given list in a random order. The class has a constant static `Random RANDOM` object which is used to generate these items and make random packing choices. The seed value (the argument to the Random constructor) is set to a fixed value in order to create consistent and reproducible results for debugging purposes. Feel free to change the seed to a different value to generate different test cases, or replace the constructor call with the 0-argument constructor `new Random()` to generate a different seed every time your program runs.

3. Rushed Packing

Now to describe the three strategies that you will use to pack suitcases. The first strategy, called “rushed packing”, pretends that you have an ordered list of items that you will try to pack, one at a time, until the suitcase is as full as possible.

3.1 Method Description

The **`rushedPacking()`** method will **recursively** solve the suitcase packing problem by trying to pack the remaining items **in the order they appear in the `itemsRemaining`** ArrayList of the Suitcase class.

For instance, if the items `[A(2x4), B(4x1), C(3x4), D(3x3)]` have not yet been packed, then item `A` should be packed first, if it can fit. If `A` cannot fit, then `B` should be packed first instead, and so on. (It may help you to first implement the method iteratively to understand the problem better, and then change the method into a recursive one.)

The **rushedPacking** method **must be implemented recursively**, either by calling itself, or by calling another private static helper method which is itself recursive. **You can use loops** in this method, but it still must call itself at least once, except in the base case(s).

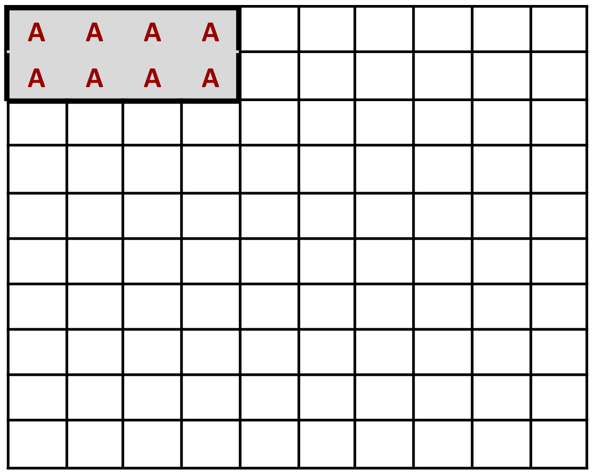
See the below visualization of how the rushed packing strategy should progress. **Start coding only after you've worked through the example below in Section 3.2, created and solved a few examples of your own, and have integrated them into your tester methods.** Consider planning your base cases, recursive cases, and recursive calls by describing or drawing the approach you'll take before starting to code. **Remember to include inline comments to explain your algorithm!**

3.2 Example

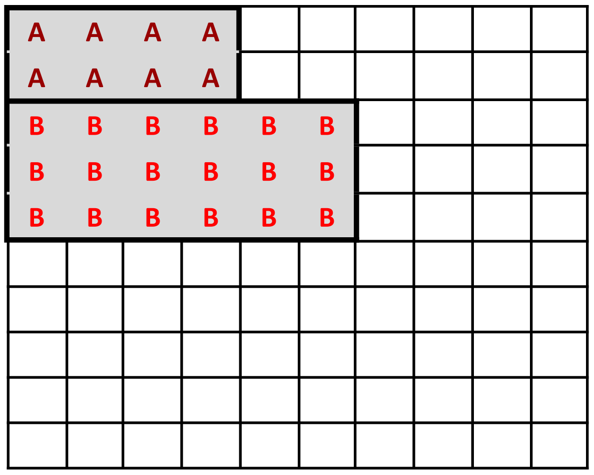
Consider the following example with seven items:

`A(4x2), B(6x3), C(7x4), D(4x5), E(4x5), F(5x4), G(2x6)`

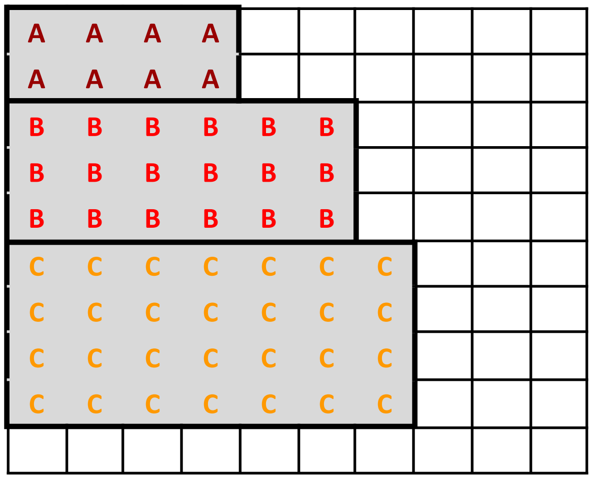
Which we wish to pack inside a 10x10 suitcase. For the rushedPacking method, in each round of the algorithm we will go through the unpacked items in the provided order until we find one which fits. This process is illustrated below, with white squares representing unfilled spaces, and gray spaces with a letter representing a packed item.

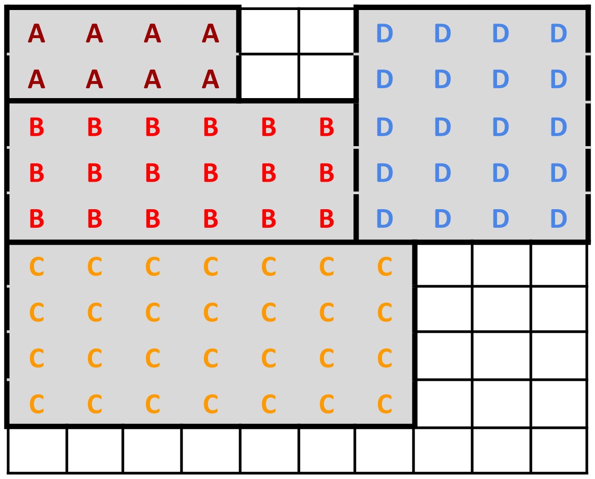


1. A is added



2. B is added


3. C is added


4. D is added

Each of the items A, B, C, and D are added in the first four rounds as they can be packed. After this, no more items are packed, as E, F, and G are all too large to fit. Therefore in this case the rushed packing strategy will choose to pack only these four items, for a total area of 74 out of 100 squares in the suitcase being packed. Not a bad strategy for a first try, but we can do better!

4. Greedy Packing

The second strategy, called "greedy packing", will pack items using a greedy algorithm based on packing the items with the largest size first (you might try to do something similar when packing in real-life).

4.1 Method Description

The **`greedyPacking()`** method will **recursively** solve the suitcase packing problem by trying to pack the remaining items **in order of largest to smallest by area (width x height)**. For instance, if the items `[A(2x4), B(4x1), C(3x4), D(3x3)]` have not yet been packed, then item `C` should be packed first, if it can fit, as it has the largest area of 3*4=12. The `Item.width` and `Item.height` fields can help you calculate an item's area. If `C` cannot fit, then `D` should be packed first instead, and so on. If there is a tie between two items, the item appearing earlier in the list should be packed first. It may help you to first implement the method iteratively to understand the problem better, and then change the method into a recursive one.

The **greedyPacking** method **must be implemented recursively**, either by calling itself, or by calling another private static helper method which is itself recursive. **You can use loops** in this method, but it still must call itself at least once, except in the base case(s).

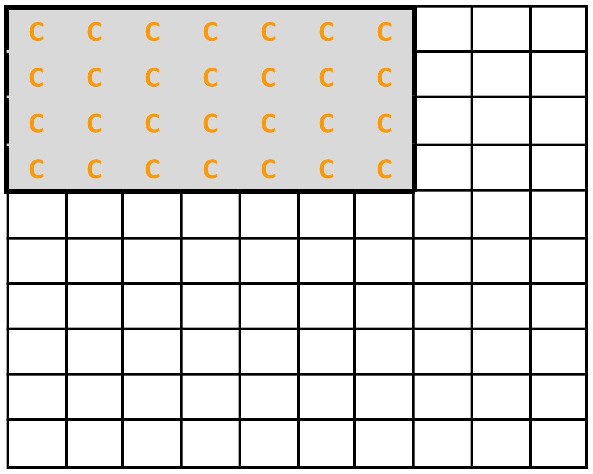
See the below visualization in Section 4.2 for an example of how the greedy packing strategy should progress. **Start coding only after you've worked through the example and have created your tester methods.** Consider planning your base cases, recursive cases, and recursive calls by describing or drawing the approach you'll take before starting to code. **Remember to include inline comments to explain your algorithm!**

4.2 Example

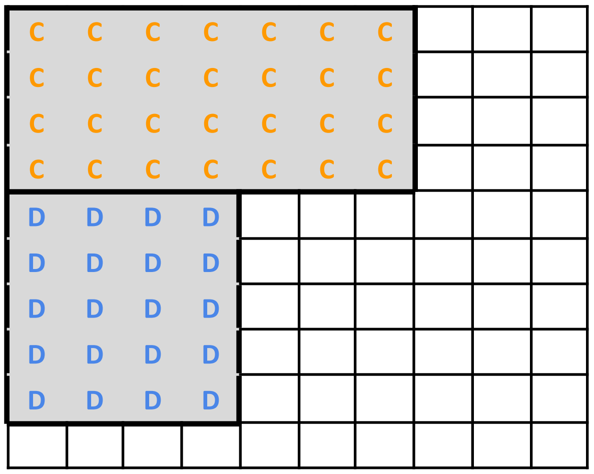
Consider the same scenario as before with seven items that we wish to pack into a 10x10 suitcase.

`A(4x2), B(6x3), C(7x4), D(4x5), E(4x5), F(5x4), G(2x6)`

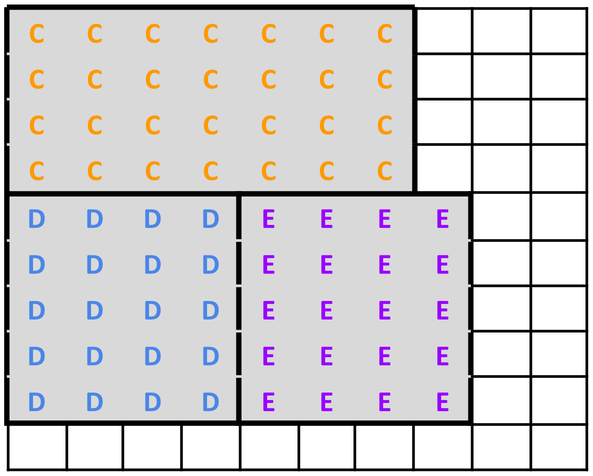
For the greedyPacking method, in each round of the algorithm we will go through the unpacked items and add the largest one by area which fits. If there is a tie, the first one in the list will be chosen. This process is illustrated below.



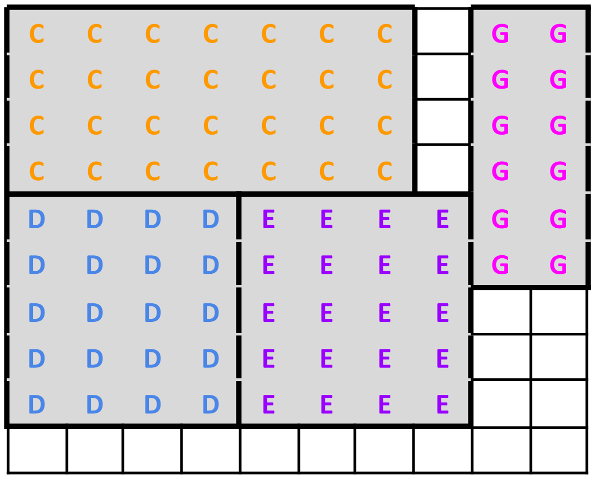
1. C (area = 28) is added



2. D (area = 20) is added



3. E (area = 20) is added



4. G (area = 12) is added

Each of the items C, D, E, and G are added in the first four rounds as they can be packed. After this, no more items are packed as A, B, and F are all too large to fit. In this case the greedy strategy filled a total of 80 spaces out of 100; 6 more than with the rushed packing strategy!

5. Optimal Packing

The final strategy to implement is very easily described; find the best possible packing order by trying them all. This is easier said than done and may not be very practical, but it will be sure to save you money!

5.1 Method Description

The **`optimalPacking()`** method will **recursively** solve the suitcase packing problem by trying **all possible different orders that the items can be packed in**, and returning the one which **packs the most area in the suitcase**. For instance, if we have items `[A(2x4), B(4x1), C(3x4), D(3x3)]`, then we could either try and pack `A` first, or try and pack `B` first, or try and pack `C` first, or try and pack `D` first. If we pack `A` first, then we can pack either `B`, `C`, or `D` next, and so on.

The `Suitcase.areaPacked()` method can tell you how much area of a Suitcase has been filled with items.

The **greedyPacking** method **must be implemented recursively**, either by calling itself, or by calling another private static helper method which is itself recursive. **You can use loops** in this method, but it still must call itself at least once, except in the base case(s).

See the below visualization in Section 5.2 for an example of the output of the optimal packing strategy. **Start coding only *after* you've created your tester method** – it's a little different from what you're used to!

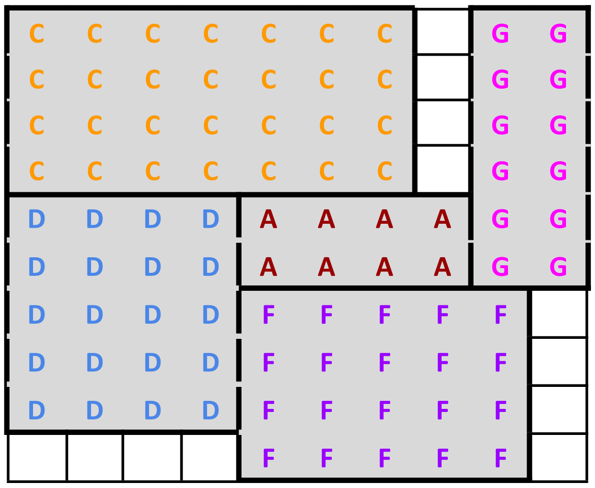
Consider planning your base cases, recursive cases, and recursive calls by describing or drawing the approach you'll take before starting to code. **Remember to include inline comments to explain your algorithm!**

5.2 Example

The same scenario is back, listed again for your convenience:

```
A(4x2), B(6x3), C(7x4), D(4x5), E(4x5), F(5x4), G(2x6)
```

For the optimalPacking method, we must try **ALL** different packing orders, and return the one which maximizes the total area packed in the suitcase. This time, as there are many recursive calls being made to different sub-problems (around 900 in our implementation) and thus many branches in our recursive call-tree, we will only illustrate the final, optimal solution below:



In this optimal solution, the items are added in the order C, D, A, F, and G, with no more items being small enough to fit in the remaining spaces. Out of a total of 100 squares, 88 are packed, beating out the rushed strategy by 14 squares and even the greedy strategy by 8 squares! Now we've got our money's worth!

6. Testing Your Implementations

You will first need to write some tests using the techniques we've been using so far in all of our projects, such as working through an example by hand and then hard-coding the solution in the tester class. However, for the optimal packing strategy, it can be very difficult to determine the optimal packing by hand, and only using small examples may not test your code very thoroughly.

For that reason, in this assignment you will also use **randomized testing** to test the optimal strategy. More tools for your programming and testing toolbox!

6.1 Background: Fuzz Testing

The new randomized testing technique we will introduce is called *fuzz testing* or *fuzzing*. Fuzzing means generating a large number of random inputs for your program and checking that the program behaves as expected. Using many random inputs allows you to exercise a bunch of the different code paths in your implementation – ideally, we would exercise every possible path control flow could follow in your program with many different inputs, including edge cases.

As an aside, fuzz testing was invented here at UW-Madison (the professor is actually still here at the university) – it is a widely used technique for finding security vulnerabilities.

6.2 PackingTesting

First, you will write two test methods each for the `rushedPacking()` and `greedyPacking()` methods as you have been doing for previous assignments. Specifically, you must explicitly test both the base cases and the recursive cases. For an example of how to set up the Suitcase, see Section 2.3 above. When checking that the output from your recursive methods is correct, you should verify that the packed items and unpacked items in the Suitcase object are what you expect, where **order does matter in the lists!** You can use the `ArrayList.equals()` method to compare the lists with the expected lists.

To test the `optimalPacking()` method, you will randomly generate lists of items to pack, and then generate many *random* packing orders for these items to compare with the output from your `optimalPacking()` method. These random items and randomly packed suitcases can be generated using the Utilities class.

The idea behind this test method is that because the `optimalPacking()` method *should* return the best possible packing result, the area packed by it should always be at least (greater than or equal to) the area packed in any other packing order. For instance, if your optimal packing method packs 80 units of the suitcase, then you should check that every randomly generated packing has only <= 80 units packed. If **any** random packing performs better than the supposedly optimal packing, the test case should return false.

The specific requirements for all of the test cases are listed in the JavaDoc comments of the PackingTester class.

Extra Challenges (Not for Credit - For Recursion Fans)

You will notice that the assignment doesn't account for *different possible positions* that you could pack each item in. If you study the Suitcase class, you can see that each item is packed in the left-top-most position that it will fit in at the time that it is packed.

The Challenge: Can you write a method similar to `optimalPacking()` which tries all possible packing positions for each item, rather than just the left-top-most position, and returns an even better packing result? The `canPackItemAt()` and `packItemAt()` methods will be helpful; to use them you will have to change them from private to public. ***Beware!*** This will *massively* increase the number of recursive calls and may make your algorithm run for too long or run out of stack space on large examples, so try out your method on only small cases. If you'd like to submit this to Gradescope, make sure all methods are private.

As an added challenge, what if you were also allowed to rotate each item by 90 degrees when packing it, as you would in the real world?

Problems like our suitcase packing problem are called Constraint Satisfaction Problems (CSPs) or Constraint Satisfaction Optimization Problems (CSOPs), of which there are many different variants and applications. There are also many sophisticated algorithms for performing efficient optimal search or quickly finding approximately optimal results. Such algorithms are extremely useful when we want to find a solution for a real-world problem in a reasonable amount of time. See, for instance, approximating the traveling salesman problem to route your Grubhub driver to get your order faster!

Assignment Submission

Hooray, you've finished this CS 300 programming assignment!

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):

- **Packing.java**
- **PackingTester.java**

Additionally, if you used any generative AI at any point during your development, you ***must*** include the full transcript of your interaction in a file called

- log.txt

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.

Students whose final submission is made before 5pm on the Wednesday before the due date will receive an additional 5% bonus toward this assignment. Submissions made after this time are NOT eligible for this bonus, but you may continue to make submissions until 10:00PM CDT on the due date with no penalty.

Copyright notice