P06 - Calculating Partitions

Overview

So far your approaches to solving various problems in programming involve iteration. However, sometimes, depending on the problem, a recursive approach is simpler in implementation than a completely iterative solution. (You may not believe this right now as you are learning and wrestling with the concept, but it's a great and vital programming tool.)

In this assignment you will solve three different problems about **partitions of N** using recursion. If you don't know what a partition of N is, we will break it down for you in Section 2.1. The problems you are solving are relevant to many different applications such as number theory, graphs, and even abstract algebra! (Each of these with possible applications in CS topics like graphics and cryptography (security)!)

Grading Rubric

5 points	**Pre-assignment Quiz**: accessible through Canvas until 11:59PM on 10/27.
+5%	**Bonus Points**: students whose ***final*** submission to Gradescope is before **5:00 PM Central Time** on **WED 10/30** *and* who pass ALL immediate tests will receive an additional 2.5 points toward this assignment, **up to a maximum total of 50 points**.
17 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.
14 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, focusing on algorithms, use of programming constructs, and style/readability.
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 register your partnership ***before the autograder is released***. If you do not do so, you must complete this assignment individually or be subject to Academic Misconduct sanctions.

- The ONLY external libraries you may use in your program are:
  `java.util.ArrayList` - **All four (4) source files**

  `java.util.Random` - **PartitionCalculatorTester.java**

- 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.

- You are allowed to define additional **private** helper methods.

- Only `PartitionCalculatorTester` 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 **screenshots** of your interactions with the tool ***clearly showing all prompts and responses in full***. 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/427315/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:

- Appropriate Academic Conduct, which addresses such questions as:
  - How much can you talk to your classmates?
  - How much can you look up on the internet?
  - How do I cite my sources?
  - 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 Calculating Partitions**.
   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 three (3) Java source files from the assignment page on Canvas:
   a. **Partition.java** (does NOT include a main method)
   b. **PartitionCalculatorTester.java** (includes a main method)
   c. **TesterUtility.java** (does NOT include a main method)

3. Download two (2) more starter files from the assignment page on Canvas:
   a. **partitions.csv** (contains the answers for the first 2 problems)
   b. **permutedPartitions.csv** (contains the answers for the last problem)

4. Create one (1) Java source file within that project's src folder:
   a. **PartitionCalculator.java** (does NOT include a main method)

1. Provided Starter Code

We provided several files to help you out in implementing your code, in particular, `Partition.java` and `TesterUtility.java`. Before you start implementing code, read over them **CAREFULLY** as they will help you with both of the recursive methods and testers. Here we provide a ***brief overview*** of them.

1.1 Partition class

The provided **Partition** class represents a collection of numbers which will (hopefully, eventually) add up to the given value N. You do not need to implement any methods in this class, but you should fully read the javadoc comments in the provided source code file, in order to understand the methods available to you. They will be very important to complete the main recursive methods.

Each Partition object tracks the **numbers currently assigned to it** and their sum, as well as a boolean indicating whether the **order of those numbers is significant**. (The order will only be significant for **Problem 3** later.)

Consider this `toString()` output of two partial Partitions of 5, with their `orderMatters` set to false:

```
Partition of N=5: [1, 2] --INVALID
Partition of N=5: [2, 1] --INVALID
p1.equals(p2): true
```

Since neither of these Partitions have contents that add up to their specified N value, they are both considered "invalid" partitions of N. To make them valid, they will need to have more numbers added to them. However, in their current state, they are considered **equal**, since both contain one 1 and one 2.

It's also totally okay to add numbers to the Partition that make its total ***greater than*** N! If I call `addNumber(3)` on the first partition, this is just "overfilling" and results in an invalid Partition:

```
Partition of N=5: [1, 2, 3] --INVALID
```

While Partition doesn't come with its own `isValid()` method, you can determine if any given Partition is valid for yourself in exactly the same way that Partition's `toString()` does, since you have accessor methods for all of Partition's values (except orderMatters, which you can set directly for this assignment).

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

- The **constructor**, which initializes an empty partition of N
- The **`getSum()`** and **`getN()`** accessor methods, which will help you compare the current state of the partition to its goal state
- The **`addNumber()`** method, which allows you to add a new number to the partition
- The **`copyOf()`** method, which creates a new DEEP copy of the partition so you can work from the same starting state but in a different direction – say, by adding a value to one copy but not the other

1.2 Tester Utility class

The **TesterUtility** class contains methods which will help you determine expected values for your tester methods and compare generated partition lists.

- The **`getPartitionCount()`** and **`getPartitions()`** methods read expected values in from the provided CSV files – so make sure those files are correctly located in your PROJECT FOLDER, not the src directory or any other subdirectory.
- The **`comparePartitionsLists()`** method is effectively a complicated equals method for lists of partition objects, determining whether both lists contain equivalent partitions when order does or doesn't matter (and it *never* matters for the order of the Partitions themselves).

2. Calculating Partitions of N

2.1 Definitions

Let's start with some definitions and observations that are ***vital*** to working through the problem.

- A **partition of N** is a set of positive integers (i.e. whole numbers > 0), that when summed together equals `N`. For all N > 1, there are more than one partition.
  - Ex. n = 5 ⟶ [3,2]
  - Ex. n = 3 ⟶ [1,1,1]
  - Ex. n = 6 ⟶ [6]

- A set of positive integers is considered a **VALID** partition of N ***if and only if*** all they add up to `N`.
  - Ex. n = 4 ⟶ [2,1] – is NOT VALID, (2+1= 3)
  - Ex. n = 10 ⟶ [4,3,2] – is NOT VALID, (3+2+4 = 9)
  - Ex. n = 7 ⟶ [2,2,2,1] – is VALID, (2+2+2+1 = 7)
  - Ex. n = 2 ⟶ [1,1] – is VALID, (1+1 = 2)
  - Ex. n = 13 ⟶ [13] – is VALID (13 = 13)

- A set of positive integers is considered **overfilled** if the sum of all the numbers is *greater than* the N value.

- Two partitions of N are considered the **SAME** if they have the same N value and same numbers, ***regardless of order***. (Ex. [1,3,1] and [3,1,1] are considered the same partition of n = 5.)

2.2 [Problem 1] - Count the *number* of Partitions

The first recursive method that you are required to write solves the following problem:

Given a value N, return the *total number* of partitions for that value N.

`PartitionCalculator.java` needs a method with the signature below:

```
public static int numOfPartitions(int N)
```

For example if we call `numOfPartitions(5)`, the return value is 7 (as there are only 7 ***unique*** partitions of 5):

1. `[5]`
2. `[4,1] ⇔ [1,4]`
3. `[3,2] ⇔ [2,3]`
4. `[3,1,1] ⇔ [1,3,1] ⇔ [1,1,3]`
5. `[2,2,1] ⇔ [2,1,2] ⇔ [1,2,2]`
6. `[2,1,1,1] ⇔ [1,2,1,1] ⇔ [1,1,2,1] ⇔ [1,1,1,2]`
7. `[1,1,1,1,1]`

We strongly recommend implementing a private, recursive helper method for this problem. Build solutions using the Partition objects – and remember, you can make MULTIPLE recursive calls from a single method! You'll almost certainly need to create multiple Partition objects over the course of solving this problem, even though the public-facing method involves only integer values.

➔ Remember to account for the cases where a number (like 1) is used ***more than once*** or ***not at all*** in a given Partition. How can you accomplish this with NO LOOPS?

We're NOT worried about efficiency for this program; the concern here is recursive thinking. As long as you have a fully recursive solution, it can be as inefficient as it needs to be. (Stuck? See **Tips & Hints**.)

This method **MUST be recursive OR** make a **call to a recursive helper method**. (We will check this as a part of manual grading.) Using a loop of any kind will result in loss of ALL manual points for this method, ***even if there is a recursive call***.

There are several tester methods associated with this method, ***be sure to implement them as you go!*** (Testing information can be found in **Section 3**.)

2.3 [Problem 2] - Calculate all of the Partitions

The second recursive method that you are required to write solves the problem below:

Given a value N, return *all of the partitions* for that value N.

`PartitionCalculator.java` needs a method with the signature below.

```
public static ArrayList<Partition> calculatePartitions(int N)
```

For example if we call `calculatePartitions(4)`, there are a total of 5 of them, the returned list should contain the following partitions (in any order):

1. `[4]`
2. `[3,1] ⇔ [1,3]`
3. `[2,2]`
4. `[2,1,1] ⇔ [1,2,1] ⇔ [1,1,2]`
5. `[1,1,1,1]`

You've very nearly solved this problem in the previous method; now instead of COUNTING those Partitions you've created, we're actually going to save the Partition objects themselves.

This method MUST be recursive OR make a call to a recursive helper method. (We will check this as a part of manual grading.) You should be able to accomplish this WITHOUT needing a looping construct. Using a loop of any kind will result in loss of ALL manual points for this method, ***even if there is a recursive call***.

There are several tester methods associated with this method, ***be sure to implement them as you go!***

2.4 [Problem 3] - Calculate all permutations of Partitions

The second recursive method that you are required to write solves the problem below:

Given a set of all partitions for value N, return *all permutations of each partition* given.

You can assume that you are given a complete set of Partitions as input. `PartitionCalculator.java` needs a method with the signature below.

```
public static ArrayList<Partition>
        calculateAllPermutations(ArrayList<Partition> partitions)
```

For example, if we call `calculatePartitions()` passing it the list of partitions of 4 that we generated in Problem 2, the returned list should contain the following nine (9) partitions with `orderMatters` = true (in any order):

1. `[4]`
2. `[3,1]`
3. `[1,3]` (**Note**: no longer "the same as" `[3,1]`!)
4. `[2,2]`
5. `[2,1,1]`
6. `[1,2,1]`
7. `[1,1,2]`
8. `[1,1,1,1]`

This method MUST be recursive OR make a call to a recursive helper method. (We will check this as a part of manual grading.) You MAY need a looping construct to complete this method. Make sure that this loop is NOT redoing any work that the recursion is doing. If the loop does redo work accomplished by recursion, you will lose points in manual grading.

There are several tester methods associated with this method, ***be sure to implement them as you go!***

2.5 Tips & Hints

- Start with the "trivial" cases (base cases) first – and be aware that there may be more than one! After you figure out a base case, try to solve a problem that is ***one step above*** that base case, that is, after one recursive call you will hit the base case. Then add one more step above that, and so on. You should start seeing a pattern to how the problem is solved.

- Several `ArrayList` methods utilize `equals()`. Partition.java overrides that method to make it easy to utilize those methods! Make sure a Partition's `orderMatters` is set to the correct value for the given problem, or equals() may give you some unexpected results. You may access/set that value directly in this program; that's fine.

- To help with figuring out how to use/join the return value of recursive call(s), assume they return the correct answer. This helps you focus on JUST the current problem at hand and not all of the smaller sub-problems.

- Use the debugger tool! Tutorial on how to use the one in Eclipse can be found here.

- Test as you go! And draw lots of diagrams and pictures as you work through stuff!

- Note there are multiple different ways you can approach these problems using recursion. The idea is you need to explore all possibilities in a structured manner. Some solutions might start at the shortest length partition while others might start at the longest length partition.

- ***If the base case does not work, then the recursive case will not work***.

- Usually at around n=4 or n=5 is when you can start seeing if the recursive case is not working properly.

3. Testing

The provided tester file contains the signatures and Javadoc method headers describing the required tester methods. You are HIGHLY encouraged to use methods provided in `TesterUtility`. If calling some of these methods results in an `IOException`, make sure that the provided csv files are in the base folder of your project (NOT in the src folder).

NOTE: The csv files only contain the **number** of partitions for N between [1, 50]; the **partitions** for N between [1, 35]; and the **permutations of partitions** for N between [1, 12]. A recursive implementation very likely has a HIGH time complexity, so we do not recommend going past these values.

To give you a benchmark, Problem 3 with N = 12 takes *several minutes* to complete on my machine!

3.1 On 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 and is celebrating its 30th year (the professor is actually still here at the university) – it is a widely used technique for finding security vulnerabilities.

3.1.1 Fuzz Testing `numOfPartitions()`

Here are the following requirements for this test:

- Generate a random number of times to test the method between [100, 199].
- For each test, generate a new random N value for a test case in the range [1,50].
- Test that the method returns the correct value. If it does not for ANY of tests, the implementation fails.

3.1.2 Fuzz Testing `calculatePartitions()`

Here are the following requirements for this test:

- Generate a random number of times to test the method between [1, 20].
- For each test, generate a new random N value for a test case in the range [1,35].
- Test that the method returns the correct value. If it does not for ANY of tests, the implementation fails.

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, make a final submission of your source code to Gradescope.

For full credit, please submit the following files (**source code**, *not* .class files):

- `PartitionCalculator.java`
- `PartitionCalculatorTester.java`

Additionally, if you used generative AI at any point during your development, ***you must include screenshots*** showing your FULL interaction with the tool(s).

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 (which must pass ALL immediate tests) 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 Central Time on the due date with no penalty.

Copyright notice