Com S 227
Fall 2020
Miniassignment 3
Extra credit: 40 points

Due Date: Friday, November 6, 11:59 pm (midnight)

5% bonus for submitting 1 day early (by 11:59 pm Nov 5)

10% penalty for submitting 1 day late (by 11:59 pm Nov 7)

No submissions accepted after November 7, 11:59 pm

General information

This assignment is to be done on your own. See the Academic Dishonesty policy in the syllabus, http://www.cs.iastate.edu/~cs227/syllabus.html **, for details.**

You will not be able to submit your work unless you have completed the *Academic Dishonesty policy acknowledgement* on the Homework page on Canvas. Please do this right away.

If you need help, see your instructor or one of the TAs. Lots of help is also available through the Piazza discussions.

Note: This is a miniassignment and the grading is automated. If you do not submit it correctly, you will receive at most half credit.

This is a fun and interesting problem, and if you have a good understanding of Lab 7 it should not be too difficult. The 40 points will be added to the miniassignment category, which will be counted as 90 possible points for 4% of your grade, so (at best) the extra credit could add about 1.7% to your grade.

Overview

This is a short assignment to have some fun and give you some practice with recursion. Your task is to implement the recursive method **`findCombinations`** in the class **`RecursionGame`**.

This is loosely based on a well-known game called the "twenty-four game". The idea is that you are given a list of numbers and a target value (in the traditional form of the game, the target value is always 24). You must combine your numbers using addition, subtraction, multiplication, and division to obtain the target (implicitly you are also using parentheses since you can group the operations). For example, suppose your list of numbers is 2, 3, 4, 5, with a target value of 21. Some possible solutions would be ((5 – 2) * (3 + 4)), ((3 – 2) + (4 * 5)), or (3 * (2 + 5)).

(Note that in our version of the rules, you are *not* required to use all the numbers, and division is only allowed when there is no remainder. This is different from the traditional version.)

The goal of the **`findCombinations`** method is to create a list containing all possible solutions, represented as strings. It is ok for the list to contain duplicates and the results do not need to be in any order. Please note, for example, that we regard "(3 * (2 + 5))" and "(3 * (5 + 2))" as *different* solutions (this simplifies things in the long run).

Warm up

Before you start, be sure you understand the file lister example from Lab 7:

```
public static void listAllFiles(File f)
{
  if (!f.isDirectory())
  {
    // Base case: f is a file, so just print its name
    System.out.println(f.getName());
  }
  else
  {
    // Recursive case: f is a directory, so print its name,
    // and then recursively list the files and subdirectories it contains
    File[] files = f.listFiles();
    for (int i = 0; i < files.length; i += 1)
    {
      listAllFiles(files[i]);
    }
  }
}
```

This example illustrates a common use case for recursion: we have a tree-like structure (the hierarchy of directories and files on your hard drive) and we need to explore every branch of the tree to find all the files. The key to using recursion is to solve a problem by reducing it to smaller instances. In the case of the file hierarchy, a "smaller instance" corresponds to descending into a lower level of the tree.

As an aside: a minor change to the example above allows us to collect a *list* of all the files instead of printing them:

```
private static void createListOfFiles(File f, ArrayList<String> results)
{
  if (!f.isDirectory())
  {
     results.add(f.getName());
  }
  else
  {
    File[] files = f.listFiles();
    for (int i = 0; i < files.length; ++i)
    {
      createListOfFiles(files[i], results);
    }
  }
}
```

Note that the **`results`** list is created once by the *caller* of the method, and the same list is passed into each recursive call, so that in the end it contains the name of every file that was ever found.

The search for solutions to the twenty-four game is similar, in the sense that recursion is used to explore a tree-like structure representing all possible combinations of the numbers with arithmetic operations. In the case of searching a file hierarchy, the "tree" actually exists on your hard drive, but in the case of the twenty-four game, the "tree" is conceptual. The figures on the following pages illustrate this.

To descend down branches of this conceptual tree of possible combinations, we need to decompose the problem into smaller sub-problems so that recursion can be used. There are two possible ways to reduce the problem to a smaller sub-problem.

a) Choose two numbers *a* and *b* in the list, remove them from the list, and replace them with a single number *a + b, b + a, a * b, b * a, a – b, b – a, a / b,* or *b / a.* This reduces the list size by 1.
b) Choose a single number in the list, and remove it. This allows us to find solutions in which not all the numbers are used; again the list size is reduced by 1.

The base case is easy: if you have a list of numbers of size 1, you can just check whether that value is equal to the target.

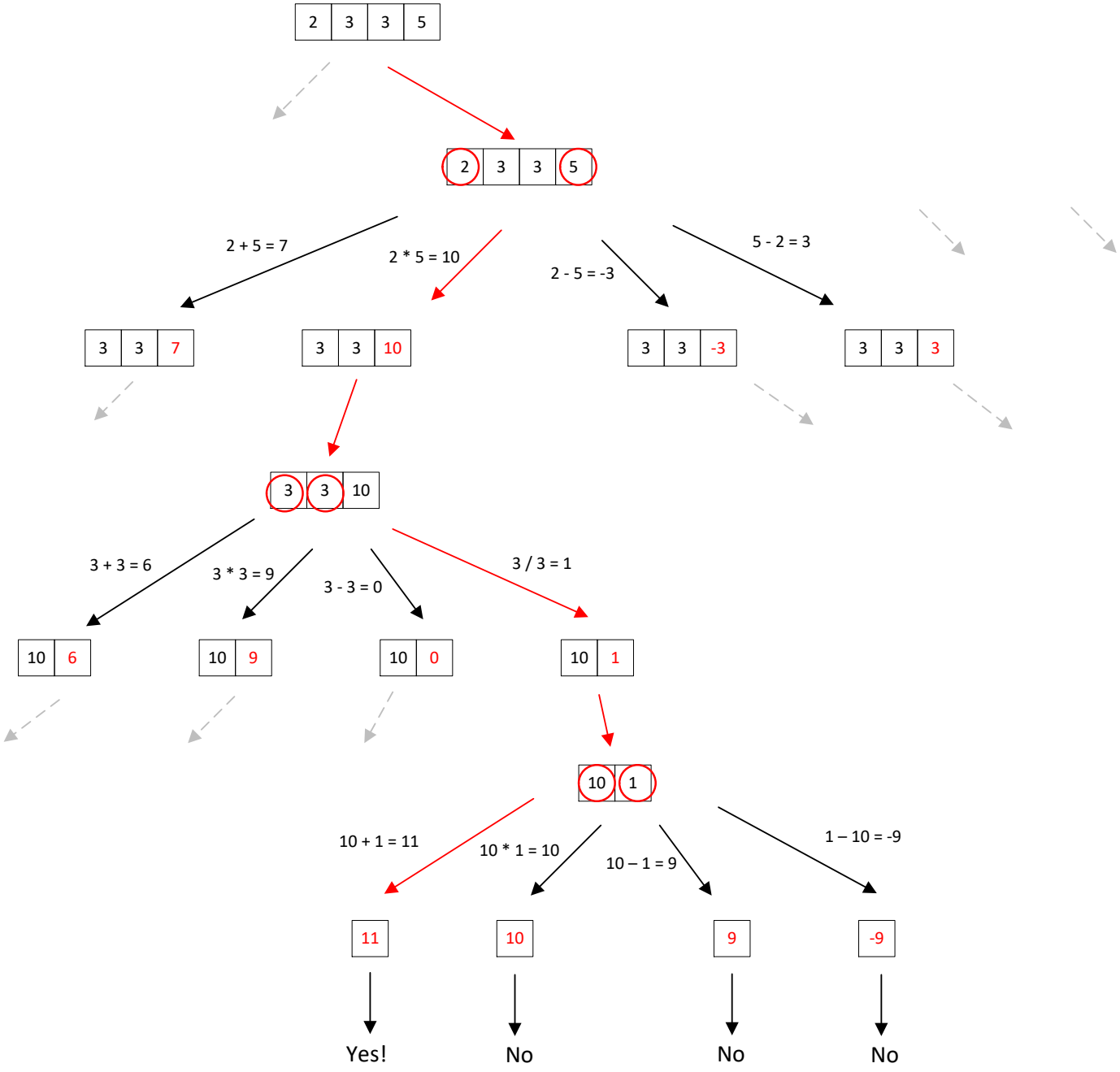
The idea is illustrated in the following figures using the list of numbers 2, 3, 3, 5 and the target value 11. In Figure 1, we start by choosing a pair of values from the list. In this example, we are examining the path in which we chose 2 and 5. We can replace the 2 and the 5 with 2 + 5, 2 * 5, 2 – 5, or 5 – 2. The possible division expressions 2 / 5 and 5 / 2 are disallowed because both have a nonzero remainder. This leads to four lists with only three numbers in each.

The same strategy can now be recursively applied to each of the three-element lists, and here we examine the branch in which the pair 3, 3 is chosen. These values can be replaced with 3 + 3, 3 * 3, 3 – 3, or 3 / 3, leading to four lists with only two numbers in each.

Again we recursively apply the same strategy. In this case there is only one possible pair to choose, so we replace the 10 and 1 with the possible values 10 + 1, 10 * 1, 10 – 1, and 1 – 10. Now each subproblem is a list of length 1, so we can directly compare each of the values with the target, 11, and there is one solution.

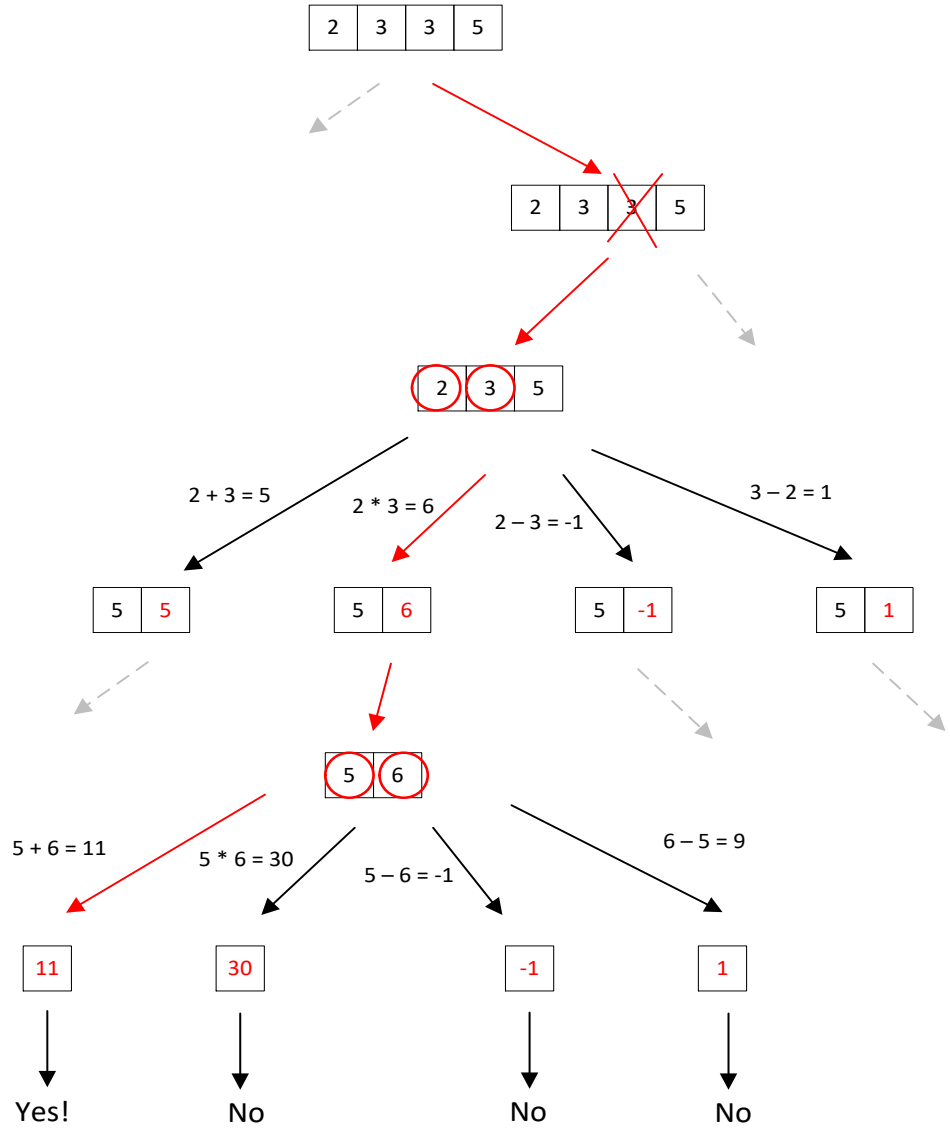
Figure 2 is similar, but also illustrates the fact that we need to allow subproblems that are formed just by removing an element and obtaining a shorter list, in order to obtain a solution that doesn't use all the numbers, such as (5 + (2 * 3)).

Figure 1: Recursive decomposition leading to solution ((2 * 5) + (3 / 3)) = 11



Solution: ((2 * 5) + (3 / 3))

Figure 2: Recursive decomposition leading to solution (5 + (2 * 3))



Solution: (5 + (2 * 3))

Pseudocode

if the list has length 1
    if the value matches the target
        add it to the list of results
otherwise
    for each number x in the list
        create a copy of the list that does not include x
        find solutions using that list
    for each pair of numbers x, y in the list
        for each allowable arithmetic combination z of x and y
            create a copy of the list without x and y, but with z added
            find solutions using that list

When we say "each allowable arithmetic combination *z* of *x* and *y*", we always include *x + y, y + x, x * y, y * x, x – y,* and *y – x*. For division, x / y or x / y is only included when there is no remainder and no division by zero. (Having x + y and y + x is redundant, but it simplifies the code and the testing to always include them both.)

The `IntExpression` class

Note: the **`IntExpression`** *class is provided for you and you should not modify it.*

The examples also illustrate a further question: when we reach a base case and check that the target value has been found, how do we know what the operations were that led to it? Somehow, we must store not only the numbers in the list, but also the arithmetic expressions that were used to obtain them. For example, in the first step when we compute 2 * 5 = 10, we need to save not only the number 10, but also something like the *string* "(2 * 5)".

The class **`IntExpression`** is provided for you to make this easy. A **`IntExpression`** object just contains a number and a string. You can construct a **`IntExpression`** with a single integer or by combining two existing **`IntExpression`** objects with one of the operators '+', '*', '-', or '/'. Doing so automatically computes the result and stores a new string representation of the old values. Parentheses are always added. You obtain the integer value using the method **`getIntValue()`** and the string form using the method **`toString()`**. Here are some examples:

```
IntExpression v1 = new IntExpression(2);
IntExpression v2 = new IntExpression(3);
System.out.println(v1.toString()); // prints "2"
System.out.println(v2.toString()); // prints "3"
IntExpression v3 = new IntExpression(v1, v2, '+');
System.out.println(v3.toString()); // prints "(2 + 3)"
System.out.println(v3.getIntValue()); // 5
IntExpression v4 = new IntExpression(v2, v3, '*');
System.out.println(v4.toString()); // prints "(3 * (2 + 3))"
```

So instead of using an **ArrayList<Integer>**, we will use **ArrayList<IntExpression>** to represent a list of values. If you wanted to run the example in Figure 1, you could use a main method with statements such as the following:

```
int[] values = {2, 3, 3, 5};
ArrayList<IntExpression> expressions = new ArrayList<IntExpression>();
for (int x : values)
{
   expressions.add(new IntExpression(x));
}
ArrayList<String> results = new ArrayList<String>(); // empty list
RecursionGame.findCombinations(expressions, target, results);
```

It is worth noting the difference in roles between the parameters. The result list is always passed into each recursive call unchanged, so that at the end, it will contain all solutions that have ever been found. Likewise the target value never changes. But for each recursive call, you'll be constructing a new list of IntExpressions just for that call. To view the solutions, it is probably best to remove duplicates and sort first:

```
ArrayList<String> uniqueResults = new ArrayList<String>();
for (String r : results)
{
  if (!uniqueResults.contains(r))
  {
     uniqueResults.add(r);
  }
}
Collections.sort(uniqueResults);
for (String r : uniqueResults)
{
   System.out.println(r);
}
```

For example, the complete list of solutions for the list 2, 3, 3, 5 with target 11 (after removing duplicates and sorting as strings) is:

```
((2 * 3) + 5)
((2 * 5) + (3 / 3))
((3 * 2) + 5)
((3 * 3) + 2)
((3 + 3) + 5)
((3 + 5) + 3)
((3 / 3) + (2 * 5))
((3 / 3) + (5 * 2))
((5 * 2) + (3 / 3))
((5 + 3) + 3)
(2 + (3 * 3))
(3 + (3 + 5))
(3 + (5 + 3))
(5 + (2 * 3))
(5 + (3 * 2))
(5 + (3 + 3))
```

The expression "(2 + 3)" is considered to be *different* from the expression "(3 + 2)". Your implementation should find all solutions and therefore needs to try both. Note, your solution can contain duplicates and the order does not matter. (The unit tests will remove duplicates and sort before checking against a correct solution.)

The SpecChecker

A SpecChecker will posted RSN.

Documentation and style

Since this is a miniassignment, the grading is automated and, in most cases, we will not be reading your code. Therefore, there are no specific documentation and requirements.
Do not use any static variables in your class.

If you have questions

For questions, please see the Piazza Q & A pages and click on the folder **`miniassignment3`**. If you don't find your question answered, then create a new post with your question. Try to state the question or topic clearly in the title of your post, and attach the tag **`miniassignment3`**. *But remember, do not post any source code for the classes that are to be turned in.* It is fine to post source code for general Java examples that are not being turned in. (In the Piazza editor, use the button labeled "pre" to have Java code formatted the way you typed it.)

If you have a question that absolutely cannot be asked without showing part of your source code, make the post "private" so that only the instructors and TAs can see it. Be sure you have stated a specific question; vague requests of the form "read all my code and tell me what's wrong with it" will generally be ignored.

Of course, the instructors and TAs are always available to help you. See the Office Hours section of the syllabus to find a time that is convenient for you. We do our best to answer every question carefully, short of actually writing your code for you, but it would be unfair for the staff to fully review your assignment in detail before it is turned in.

Any posts from the instructors on Piazza that are labeled "Official Clarification" are part of the spec, and you may lose points if you ignore them. Such posts will always be placed in the Announcements section of the course page in addition to the Q&A page. (We promise that no official clarifications will be posted within 24 hours of the due date.)

Getting started

1. Recursion can be hard to "get" at first. You will want to study examples that are already done, from class and lab, and make sure you can understand and re-do them. Be sure you have done lab 7.

2. Work out some examples on paper such as the one in Figure 1.

3. Be sure to try out the **`IntExpression`** class so that you know how use it. See the sample code in the preceding section "The IntExpression class".

4. You could start with a simpler version of the problem just to make sure you have the recursion working correctly. For example, suppose that all numbers *must* be used, and that only addition is allowed. See if you can do an example such as the set {1, 2} with target 3 (two solutions). Then you might try {1, 2, 3} with target 6 (12 solutions).

What to turn in

Note: You will need to complete the "Academic Dishonesty policy questionnaire," found on the Homework page on Blackboard, before the submission link will be visible to you.

Please submit, on Blackboard, the zip file that is created by the SpecChecker. The file will be named **`SUBMIT_THIS_mini3.zip`**. and it will be located in the directory you selected when you ran the SpecChecker. It should contain one directory, **`mini3`**, which in turn contains two files, **`RecursionGame.java`** and **`IntExpression.java`**.

Submit the zip file to Canvas using the Miniassignment 3 submission link and verify that your submission was successful by checking your submission history page. If you are not sure how to do this, see the document "Assignment Submission HOWTO" which can be found as item #9 on the Canvas front page.

We strongly recommend that you just submit the zip file created by the specchecker. If you mess something up and we have to run your code manually, you will receive at most half the points.

> We strongly recommend that you submit the zip file as created by the specchecker. If necessary for some reason, you can create a zip file yourself. The zip file must contain the directory **mini3**, which in turn should contain the files **RecursionGame.java** and **IntExpression.java**. You can accomplish this by zipping up the **src** directory of your project (NOT the whole project). The file must be a zip file, so be sure you are using the Windows or Mac zip utility, and not a third-party installation of WinRAR, 7-zip, or Winzip.