CS21120 Assignment

Countdown numbers game

Release date 22nd October 2018

Hand in date 23rd November 2018 (15:59 via Blackboard)

Feedback date 14th December 2018 (via Blackboard)

Aims and Objectives

The aim of this assignment is to give you experience in writing algorithms and using some basic data structures.

The objective is to write a system for creating an arithmetical number game.

Overview

In this assignment, you are asked to develop a system for creating and solving an arithmetical number game in the style of the TV show "Countdown". In this game the contestant chooses 6 numbers, with between 0 and 4 being "large numbers" (25, 50, 75 and 100) and the rest being "small numbers" (numbers 1 to 9, with each occurring twice) selected at random from the two sets of numbers. The system then generates a random 3-digit number. The contestant has 30 seconds to combine the numbers using the standard 4 arithmetic operators of *add*, *subtract*, *multiply* and *divide* to construct a number as close as possible to the target number. At no point in the calculation is a partial result allowed to be negative or fractional.

You are provided with some code as follows:

a) Code to play the game in the class *CountDownFX*,
b) Code for iterating over an *IExpression*,
c) Two Java *interfaces* (*IExpression* and *IExpressionFactory*) that you will need to implement in your solution

You need to complete the system by implementing a class that represents the solution as a binary tree, with the input values at the leaves and operators at internal nodes i.e. an *expression tree*. Examples of the final program running are given at the end of this document in Appendix 1 and some example expression trees are shown in Appendix 2.

Requirements

You are required to implement the game by implementing your own expression binary tree class. As part of this, you need to implement methods to construct specified trees, to construct random trees using the supplied numerical values and operators, and to evaluate the expression tree. Tree evaluation and random tree generation need to ensure that the Countdown rules are enforced (no negative or fractional values at any point in the evaluation). You need to implement a single class that implements the *IExpression* and *IExpressionFactory* interfaces provided in order to complete the system outlined above. It is also acceptable if your *IExpression* class is an inner class of your *IExpressionFactory* class.

The following gives an indication of how the marks are distributed and some suggestions for implementation:

1) *Implementation of IExpression interface methods* (30%): You should implement a sensible structure for your principle class, including declaration of the class, fields, appropriate methods defined properly etc. The public class in your submission should implement both the *IExpression* and *IExpressionFactory* interfaces (*IExpression* could be an inner class of *IExpressionFactory*). The class should represent a binary tree where the leaves are numbers (of type int) and the internal nodes hold Operations and two non-null children. The *evaluateCountdown* method tries to evaluate the equation recursively. It should check that the equation obeys the Countdown rules with no negative or fractional value at any point in the calculation and throw an Exception if the rules are violated. To check the division, you can use the mod operator ('%') to check that the remainder after division is zero.

2) *Implementation of IExpressionFactory.createRandomEquation interface method* (30%): This method should return a random equation made up of a (random) subset of the values passed. Each value passed should be used at most only once. The operation used at each node should be randomly selected from the 4 available operations. Note that an array representing an *Enum* can be returned using the *Enum values()* method. You do not need to check that the generated equation obeys the Countdown rules here as that is checked in the application using the *evaluateCountdown* method. There are various approaches possible for this, generally along the lines of:
    a. create an *IExpression* tree of one leaf node for each selected value,
    b. put them into a data structure,
    c. remove pairs of trees from the data structure and combine them into a single tree using an operation chosen at random,
    d. place the tree back into the data structure,
    e. repeat (c) and (d) until only one tree is left

    Careful consideration needs to be made of when to randomise to ensure all possible trees can be generated (i.e. in the initialisation, when removing elements or when inserting elements). Comment on the efficiency of your solution in your *Javadoc* comment.

3) *Implementation of IExpression.findBestSolution interface method (20%)*: **Note that this aspect is very challenging and should only be attempts after successfully completing the other aspects**. The approach given above in part (2) of generating a random equation is quick and guarantees that an exact solution is possible, but this isn't required in the game. The alternative is to generate a random 3-digit number, but then finding the best solution is difficult. Even iterating over all possible equation trees given the available numbers and operations, is not easy. Other heuristic methods may be possible, but not all are guaranteed to find the best solution (random guesses, some kind of bracketing algorithm, a genetic search algorithm). A successful attempt at doing this would gain all of these additional marks. A good partial attempt could still gain a significant proportion of them.

4) *Javadoc (20%).* You are not asked to provide a separate report, but you should thoroughly comment your code using correctly formatted *javadoc* comments. You should treat the commenting as your report, especially the top, class-level comment. (Don't just use the *@Override* annotations.) In particular, you should:
    a. explain what you have done and why,
    b. explain any unusual implementation features,
    c. explain any problems you have faced (including a clear statement that a particular feature isn't working),
    d. give proper credit for any third-party code or other resources used or parts of the code that you had help with and from whom, and
    e. provide a self-evaluation, giving a mark for each of the sections described here, with a reason for each, and an overall mark.

Submission

You are asked to submit only a **single** *.java* file containing all your code and documentation (in the form of comments). Your code **must** be defined in the package *cs21120.assignment2018.solution* and all your code should be contained in a **single file** called *Expression<userid>.java* , where *<userid>* should be replaced with your user ID (email) e.g. mine would be *ExpressionBpt.java*. You must use the class name *Expression<userid>* for your public class, and any support classes you implement should be *inner classes* of this i.e. defined within the same file and within the scope of your primary class.

Marking

This assignment will use an element of automated marking. I will apply my own *JUnit* tests to your code, so it is important that you follow the specification exactly. In particular, it is essential that you follow the instructions for the naming and packaging of your classes. Also, **do not modify** any of the supplied code, including package name. If you find any bugs please report them.

Resources

For this assignment, you are permitted to use classes from the java collections framework (*java.util.** package) but you should implement your own class for the implementation of *IExpression* and *IExpressionFactory*. Any resources you use must be acknowledged in your *javadoc* comments. You are provided with some code for the *EquationIterator* class, the *CountDownFX* class (which can be used to test your implementation), the *IExpression* and *IExpressionFactory* interfaces with accompanying documentation and some example input data. **Do not modify the supplied code!** To run the game, you are provided with a simple JavaFX application *CountDownFX* – you will need to change the factory line to create an instance of your class. You are also provided with some example JUnit tests, which can help guide your implementation. Please note, these are not exhaustive, and other / different tests will be applied to your submissions.

Academic conduct

As with all such assignments, the work must be your own. Do not share code with your colleagues and ensure that your own code is securely stored and not accessible by your classmates. Any sources you use should be properly credited with a citation, and any help you get (e.g. from demonstrators) should be acknowledged. Your documentation (report) must accurately reflect what you have achieved with your code, any discrepancies between your code and documentation could be treated as academic fraud.

Support

If you have any problems with understanding or completing the assignment please ask for help, either by seeking help from the advisors, by email or via the Blackboard forum for the assignment.

Bernie Tiddeman (bpt@aber.ac.uk) *22nd October 2018 updated 17th November 2018*

Appendix 1 – Example Program Execution

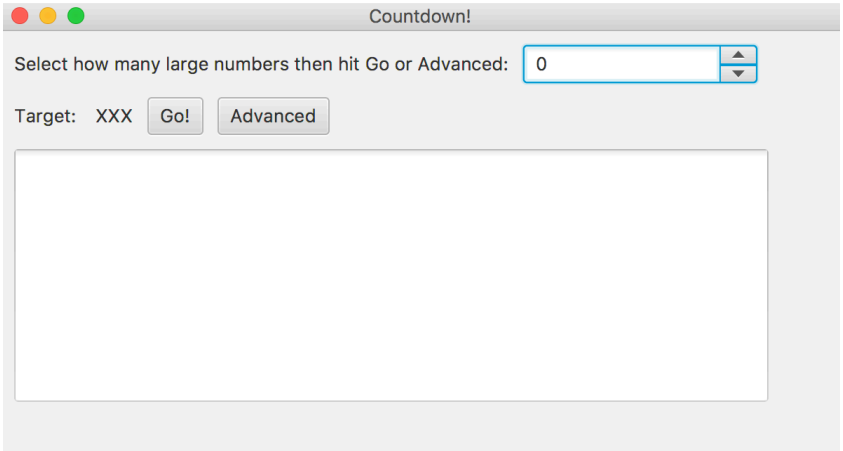


Figure 1 The application at startup.

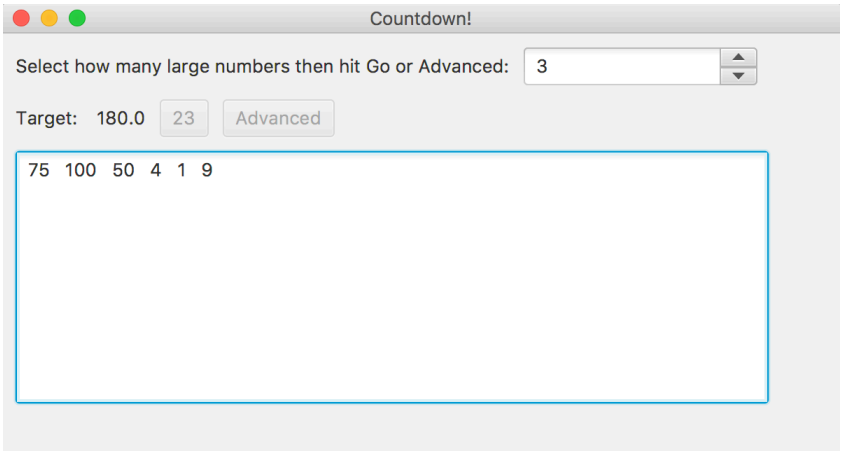


Figure 2 The application after setting the number of large numbers and clicking "Go!".

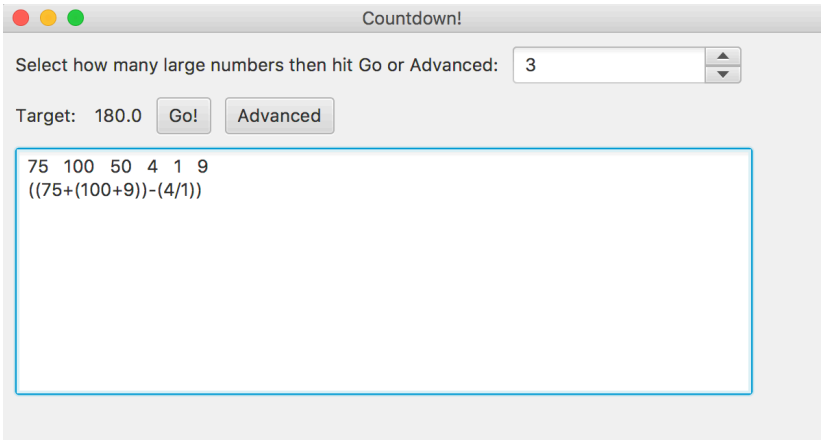


Figure 3 The application after the timeout, displaying the solution.

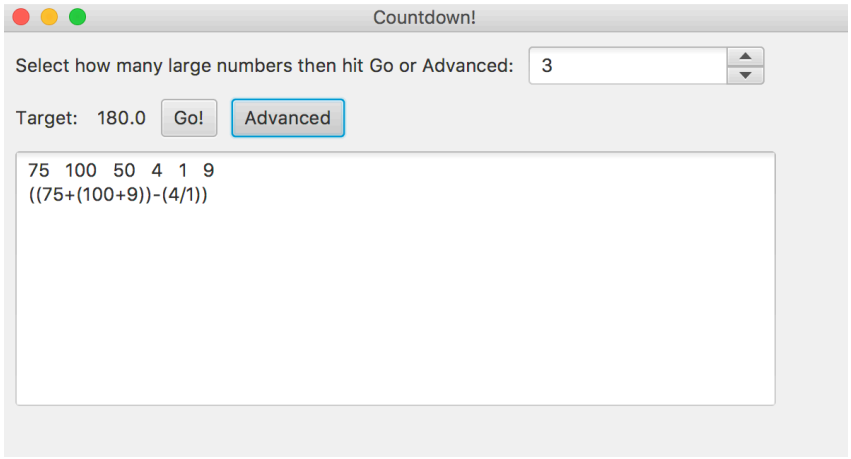


Figure 4 The application after clicking the "Advanced" button. The application may hang while it solves the problem, depending on the efficiency.

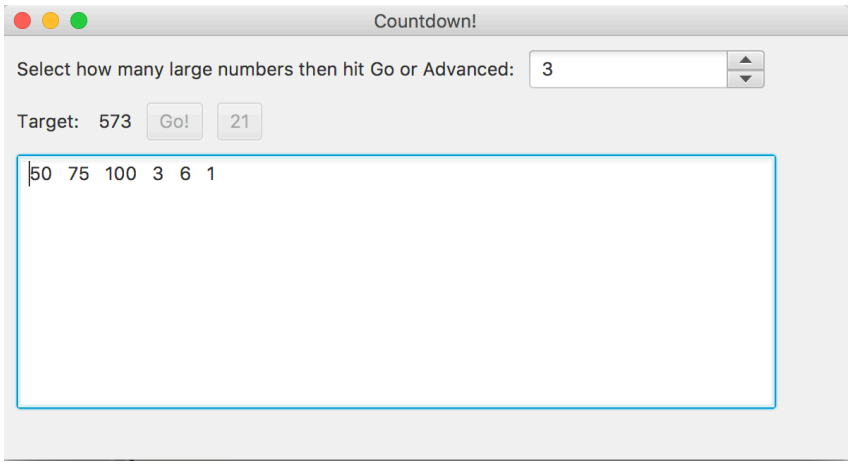


Figure 5 The application during the "Advanced" Countdown.

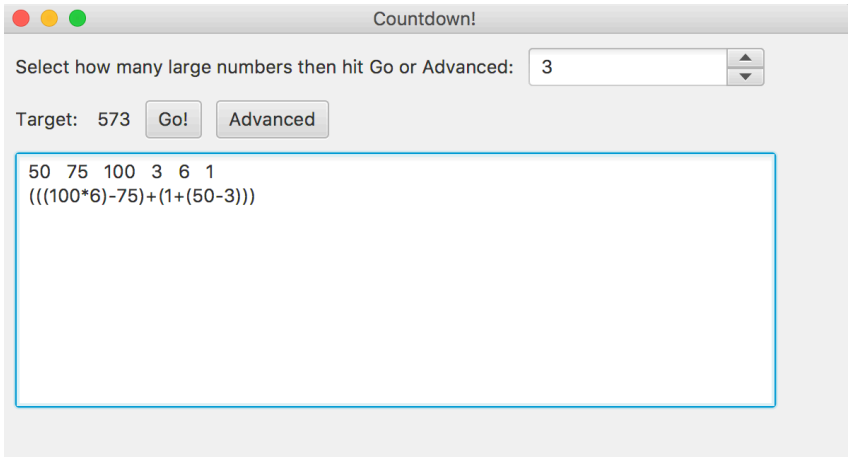
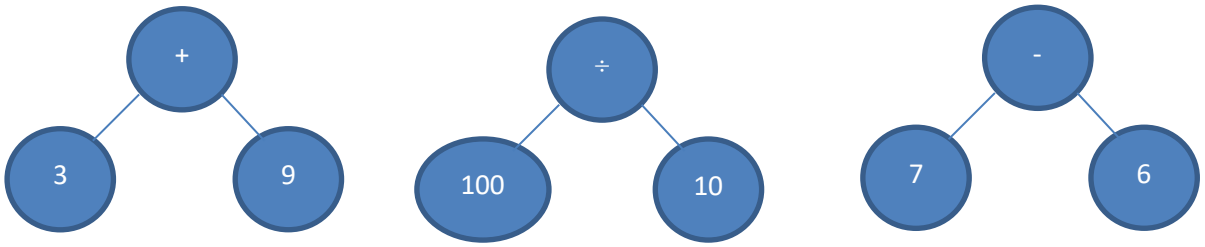


Figure 6 The application after the "Advanced" solution is displayed.

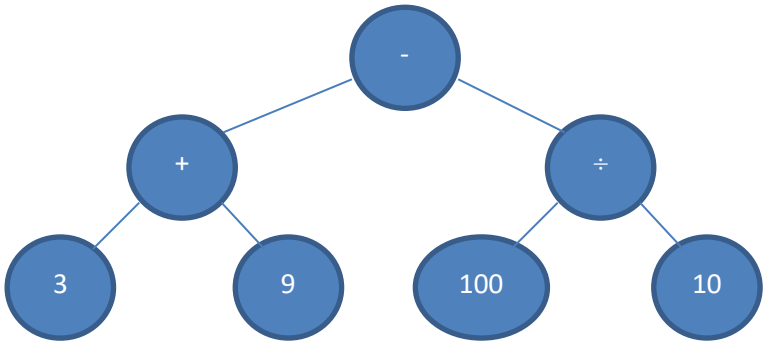
Appendix 2 – Example expression trees

A single value could be stored in your expression tree e.g. 5, 25 or 7:

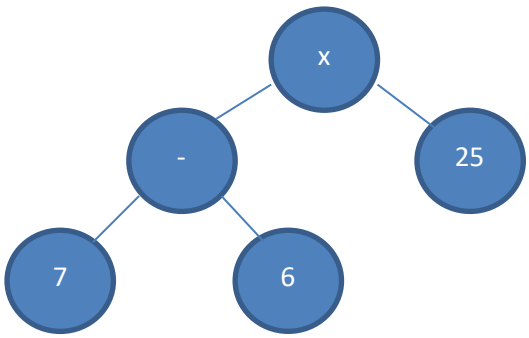
Simple expressions such as 3+9, 100÷10 or 7-6:



Slightly more complex expressions, such as (3+9)-(100÷10):



Or (7-6)x25:



Marking grid

(The marker will use a more detailed marking grid, this is for illustration).

Aspect	70-100%	60-70%	50-60%	40-50%	0-40%
IExpression interface methods (30%)	All interface methods implemented, class has excellent structure, excellent choice of internal representation.	All interface methods implemented, class has sensible structure, good choice of internal representation.	All interface methods implemented. Class structure may have some odd or incorrect aspects. Mostly good choice of internal representation.	Code compiles and some methods function as expected, but may have some serious bugs. Internal representation may have some problems or omissions.	Code may not compile or data structure has serious flaws.
Create Random Equation method (30%)	Random equations correctly generated with efficient and well-structured code	Method operates mostly correctly, but some very minor bugs. Code reasonably efficient and well structured	Method operates mostly correctly, or a serious attempt to write code that has some flaws.	Code functional in places, but has some significant bugs or omissions.	Code largely missing or non-functional.
Find best solution method (20%)	A very good attempt at finding the best solution to arbitrary problems that is fully functional.	A good attempt at finding the best solution to arbitrary problems that is partially functional.	A reasonable attempt that has some significant problems or limitations.	A partial attempt, but some significant problems or limitations.	Code missing or not a serious attempt.
Documentation based report (20%)	Clear and detailed description of algorithms / implementation, javadoc correctly formatted and complete, good use of English, good use of citations. Clear and well justified self-evaluation.	Mostly clear and detailed description of algorithms / implementation, javadoc correctly formatted and complete, mostly good use of English, good use of citations. Good self-evaluation.	Description of algorithm / implementation that may be lacking in detail, javadoc mostly correctly formatted and complete, reasonable use of English, some citations. Reasonable self-evaluation.	Some documentation, but may have gaps or lack detail of the algorithm or implementation. Some poor use of English and citations may be limited or missing. Self-evaluation may be limited or missing.	Documentation largely missing, incorrectly formatted or missing Javadoc. Poor use of English, citations and self-evaluation may be missing.