CIS 573 Software Engineering
Fall 2015
Homework #5

Introduction
This assignment consists of two parts, related to the software quality topics we've recently covered in class.

You must work alone on this assignment. As before, although you are free to discuss the intent of the assignment and ask your fellow classmates for clarification, the code and report that you write must be your own.

This assignment is worth a total of 100 points and is due the day before Thanksgiving.

Part 1: Fault Tolerance & Reliability

In this part of the assignment, you will implement three fault tolerance techniques to attempt to increase the reliability of some code.

Background.
The code for which you will use fault tolerance techniques are two methods that attempt to solve the subset sum problem: http://en.wikipedia.org/wiki/Subset_sum_problem

Given an array of ints and a target value, the methods try to determine whether any combination of the ints adds up to the target. The return value is a boolean array in which the value of each element indicates whether or not the corresponding element in the int array should be included in the solution.

For instance, if the input array was {2, 5, 8, 1} and the target was 9, the correct output would be {false, false, true, true}, indicating that the first two elements are not part of the solution, but the last two are, since 8 + 1 = 9.

There are certainly cases in which there may be multiple correct solutions. For instance, if the input is {2, 3, 5, 6} and the target is 8, then both {true, false, false, true} and {false, true, true, false} should be considered correct.

Likewise, if the input is {2, 2, 4} and the target is 4, then both {true, true, false} and {false, false, true} are correct: the problem does not ask for the "smallest" subset, just any valid subset that adds up to the target.

If no solution can be found because there is no combination of ints that adds up to the target, the methods return null. Keep in mind that determining whether that is the correct output (i.e., whether no subset adds up to the target) is just as hard as deriving a solution in the first place, so for our purposes we will consider a null output as correct.

Getting started.
Download the **SubsetSum-dist** Eclipse project from Canvas. This contains the skeleton code for the methods you need to implement, as well as some helper classes.

When you set up your Eclipse environment, be sure that the **subsetsumImpl.jar** file (in the project root directory) is in your project's build path. This jar file contains a class called *SubsetSumImplementations* that contains two methods: *solveBF* and *solveDP*. The first is a brute force solution, and the second uses dynamic programming. Each takes an int array and an int target value and returns a boolean array, as described above.

For the purposes of this assignment, we will treat these methods as "black boxes," i.e. you will not have access to the source code. However, these methods both have faults, and thus you will use fault tolerance techniques to attempt to increase their reliability.

To help get you started, we have provided a class called *Driver* that is used to execute the different implementations and measure their reliability and execution time.

As you can see, there is a method called *runTest* which takes a *SubsetSumSolver* and calls its *solve* method 10,000 times with random inputs to see how frequently it is able to produce a correct result and then report the total time running the *solve* method.

The *main* method then calls *runTest* with five different implementations of *SubsetSumSolver*. The first uses only the dynamic programming solution, and the second uses only the brute force solution.

Step 1. Implement the acceptance test.
Before you begin using the fault tolerance techniques below, you need to first create the acceptance test.

Implement the *accept* method in *SubsetSumSolver* so that it returns true if the boolean[] parameter is a correct solution to the subset sum problem as described above, and false if it is not. Keep in mind that if the boolean[] is null, we consider that a correct solution for our purposes.

Once you've done that, run the program that we gave you to make sure that your code is set up correctly. You should see that the dynamic programming solution has a reliability of around 72% and finishes quite quickly, whereas the brute force solution has a reliability of around 91% and takes much longer to finish.

Aside from uncommenting the calls to *runTest* in subsequent steps in *main*, you should not need to change the *Driver* or *SubsetSumSolver* classes! Please notify the instructor if you feel it is necessary to do so.

Step 2. Retry Block.
In this step you will implement the *solve* method in the *SubsetSumRetryBlock* class using a Retry Block.

Your Retry Block should:

- use the static *SubsetSumImplementations.solveDP* method to try to find a solution
- use the *shuffle* method (which you need to implement!) in this class for data re-expression; it should randomly permute the order of the elements in the array
- use the *accept* method that you implemented in Step 1 as the acceptance test
- after the **third** attempt to find a solution, throw a *ValidSolutionNotFoundException* if the acceptance test fails

Don't forget to take a checkpoint of the "good" state and restore it before each attempt to find a solution!

You may note that the solution that comes back from *SubsetSumRetryBlock.solve* may no longer correspond to the elements in the original input if you need to shuffle it, but you do not have to worry about that for this assignment; as you can see, the test driver only cares about whether the method throws an exception, indicating failure.

After implementing the Retry Block, uncomment the call to *runTest* for Step 2 in the *Driver* class and run the program again.

You should see the reliability increase compared to the pure dynamic programming solution. Yay, fault tolerance!

Step 3. Recovery Block.
In this step you will implement the *solve* method in the *SubsetSumRecoveryBlock* class using a Recovery Block.

Your Recovery Block should:

- first use the static *SubsetSumImplementations.solveDP* method to try to find a solution
- use the *accept* method that you wrote in Step 1 as the acceptance test
- if the acceptance test fails, try the *SubsetSumImplementations.solveBF* method
- if the acceptance test fails again, throw a *ValidSolutionNotFoundException*

Don't forget to take a checkpoint of the "good" state and restore it before each attempt to find a solution!

After implementing the Recovery Block, uncomment the call to *runTest* for Step 3 in the *Driver* class and run the program again.

You should see the reliability increase compared to the previous solutions, and it should be faster than the brute force solution but still slower than the dynamic programming solution.

Step 4. Parallelized Recovery Block.
Finally, you will implement the *solve* method in the *SubsetSumParallelRecoveryBlock* class using a Recovery Block in which the two implementations (*SubsetSumImplementations.solveDP* and *SubsetSumImplementations.solveBF*) are run in parallel in separate threads.

If you are not sure about how to use threads in Java, there is a good tutorial at http://docs.oracle.com/javase/tutorial/essential/concurrency/

You can assume that, if the first implementation to finish returns a solution which passes the acceptance test, you do not need to wait for the other; however, it is okay to wait for both to finish, as this makes things a bit simpler (though slower).

As in Step 3, if neither solution passes the acceptance test, you should throw a *ValidSolutionNotFoundException.*

After implementing the Recovery Block, uncomment the call to *runTest* for Step 4 in the *Driver* class and run the program again.

You should see the reliability stay about the same as in the Recovery Block. As for the execution time, try to get an understanding of what you're observing compared to what you would expect. Although you do not need to analyze the execution time, and this part will be graded on the correctness of your implementation, you should try to figure out what's happening in case… for instance… it comes up on an exam. ;-)

Final Note.
After finishing the three implementations, please copy and paste the output of the *Driver* program into a plaintext or PDF file and submit that along with your code as described below.

Part 2: Efficiency

In this part of the assignment, you are asked to improve the performance (execution time) of a Java program that attempts to detect plagiarism in a corpus of documents.

Plagiarism detection is a very difficult problem to solve, but a simple approach is to just look for common words and phrases between documents. If two (or more) documents contain many of the same phrases, then there is a good possibility that one author copied from the other.

The program you will improve detects common phrases of size *windowSize* in a corpus of documents, and then report pairs of documents for which the number of common phrases is equal to or greater than some *threshold*, sorted by number of common phrases.

Getting started.

Download the **Plagiarism-dist** Eclipse project from Canvas and note that the *Main* class contains a main method that creates a *PlagiarismDetector* object and specifies the directory where the corpus of documents is located. It then invokes the *detectPlagiarism* method and specifies the *windowSize* and *threshold*.

Aside from the general *detectPlagiarism* method, the *PlagiarismDetector* also has helper methods to:

- read a file and store its contents in a List of Strings
- create the distinct phrases, each of which is of size *windowSize*
- find the common phrases between two documents
- sort the results

The main method also has some code to measure the execution time and to print out the results that come back from the *PlagiarismDetector*.

You can, of course, create your own corpus as you modify this program, but we will evaluate the performance of your program using the corpus and configuration (*windowSize* and *threshold* both 4) that we provided.

A modified version of this problem (and the corpus of documents) was originally used at the University of Chicago and presented as a Nifty Assignment at SIGCSE 2008 (http://nifty.stanford.edu/2008/franke-catch-plagiarists/), and the implementation was done by the CIS 573 instruction staff. We will assume that this implementation is correct for our purposes, but if you believe you have found a critical bug, please notify the instructor.

Also download the **inefficiencies.xlsx** spreadsheet so that you can record your changes and observations.

Note initial execution time.
Before you begin modifying the code, run the program and then make a note of the time reported on your system in cell B6 of the spreadsheet.

Obviously, the time will vary slightly on each execution, particularly as the program is using "wall clock time," but try to get a sense of the approximate time it takes to run the program on your computer.

Before you begin modifying the code, be sure to make a backup copy of the original code so that you can compare the two versions as you go along and then compare them once you're done.

Also make a note of the program's output, which we assume to be correct. Make sure that your changes to the code don't accidentally break it!

Objectives.
Your goal in this assignment is to improve the execution time of the code, using the various techniques seen in class.

Keep in mind that you are asked to focus only on improving the execution time of the code. It is okay if your changes have a negative effect on things like memory usage or other aspects of quality (*except* for correctness, of course!), and it is also okay to make an improvement that makes performance initially worse if you can justify that it would reasonably make performance better later on.

Restrictions.
Please do not change the code in Main.java; it is only provided to you as a test driver. Even though it may contain some inefficiencies, you should only be focusing on the PlagiarismDetector class.

Also, you must **not** change the "public interface" of the PlagiarismDetector class, i.e. you cannot change the method signatures of the public methods and you may not remove them. Additionally, you must implement all code in Java.

In making changes to the code, you may not assume that:

- the program will always use the same set of documents that was provided to you
- the number and size of documents in the corpus will always be the same as the one that was provided to you
- the *windowSize* and *threshold* will always be the same as in the driver code provided to you
- the code is always executed on a multi-core/multi-processor machine

In some cases, you may need to make a decision that is somewhat dictated by the input, e.g. it may be better to use one data structure for small window sizes and a different one for large ones. In such cases, choose the one that works best for the inputs we provided, but ask a member of the instruction staff if you need help.

If you have any questions about what you can and cannot do to the code and what you can and cannot assume, please contact a member of the instruction staff.

Improving efficiency.
Part of the challenge of this assignment is knowing when you can stop improving the code. You may be tempted to ask "how fast is fast enough?" or "what percent improvement should I be getting?" These are impossible to answer, particularly as execution time will vary on different machines with different processor loads, and different changes may have different effects on different types of hardware and in different situations.

In order to get full credit for this assignment, you need to identify and fix at least **10 inefficiencies** in the code, as described below.

There are at least **three major inefficiencies** that, if fixed, would each individually lead to at least a 10% improvement in execution time. To receive full credit on this assignment, you must identify and fix three of these major inefficiencies. You should describe these major inefficiencies in rows 10-12 of the spreadsheet, and modify the code as needed.

The three fixes you make should each regularly and consistently improve the execution time (compared to the original, unchanged version) by over 10%. If you find on multiple executions that sometimes performance improves by a little more than 10%, and sometimes a little less, that's probably not what we're looking for, but ask a member of the instruction staff if you need help.

Note that each change should *on its own* lead to at least a 10% improvement *compared to the original implementation* but you may not see a 10% improvement each time you make one of the changes. That is, if you make change #1 and then make change #2, you may not see an additional 10% improvement, even though change #2 *alone* would improve the efficiency compared to the original.

In addition to making those three changes, you must identify and fix **at least seven minor inefficiencies** in the code that, although they may not have significant impact on the execution time or may only improve it in certain circumstances, still fall under the category of "doing unnecessary work" as discussed in class.

The improvements you may make in these cases include things like:

- not checking conditions that cannot be true at that point
- using lazy evaluation
- not calling methods repeatedly when their return values can't change
- not creating unnecessary objects

Although you only need to identify and fix seven minor inefficiencies, you may list up to 10 of them in rows 15-24 of the spreadsheet. However, only the “best” seven of these (as determined by the TA) will be graded; this means that you can still get full credit even if three of the things you list would not actually improve efficiency.

Please note that there will be no extra credit for identifying and fixing more than seven of the minor inefficiencies, and submissions that list more than 10 minor inefficiencies will not be considered for grading.

Last, keep in mind that for this assignment, it’s not enough to just make the code faster, e.g. by reimplementing it or finding another version online (which… uh… you really shouldn’t be doing) that happens to be faster. A significant part of the grade comes from the inefficiencies in the existing code you identify, not just the overall improvements you make.

Measure final execution time.
After you’ve finished modifying the code, record the execution time (cell C28) as reported by the *Main* program.

You do not need to report the new execution times for each intermediate change that you make (even for the major inefficiencies), just the execution times for the original code and for the final code that you submit.

A few words about using Piazza...

Please be careful about how you use Piazza for this assignment. It is important that you do not reveal (accidentally or intentionally) your solutions.

Please use Piazza for (a) clarification questions regarding the intent of the assignment, (b) clarification questions regarding the specification, and (c) getting help with things like using Java threads.

Please do **not** post questions that reveal your solutions about the implementation decisions you made in Part 1 or the inefficiencies you found and how you fixed them in Part 2. These are things that all students need to figure out on their own.

Before you post anything on Piazza, think to yourself “is my question giving away an answer?” If you think it is, please email the instruction staff directly and we’ll post it if we think it’s appropriate.

Deliverables
Your final deliverable should consist of a *single* zip file containing the following:

- the three .java files in which you implemented the fault tolerance techniques for Part 1
- a plaintext file or PDF containing the output of the *Driver* program for Part 1 showing the reliability and execution times for your fault tolerance implementations
- the modified PlagiarismDetector.java for Part 2
- the spreadsheet for Part 2 (please use .xls or .xlsx, and **not** .ods or .numbers)

Please do not submit the entire Eclipse projects, the test drivers, the corpus of documents, etc. Only submit a zip file containing the six files listed above.

Submission
All deliverables are due in Canvas by **Wednesday, November 25, at 5:00pm**. Late submissions will be penalized by 10% if 0-24 hours late, 20% for 24-48 hours late, and so on, up to one week, after which they will no longer be accepted.

Academic Honesty and Collaboration
You are expected to work **alone** on this assignment.

You are free to discuss the intent of the assignment with your classmates and ask clarification questions.

However, you may not discuss or share solutions or findings with other students. In particular, *you absolutely must* ***not*** *share or discuss your code or your spreadsheet documenting the changes you made*. Please see the course policy on academic honesty (posted in Canvas on the “Syllabus” page) for more information.

As always, if you run into problems, please ask a member of the teaching staff for help before trying to find solutions online or asking your classmates!

Last updated: 8 November 2015, 10:15pm