6.005 Elements of Software Construction | Fall 2011
Problem Set 1: Pi Poetry

The purpose of this problem set is to give you practice with test-first programming: given specifications, writing unit tests and implementing the code to meet the specification.

The problems are presented in a logical order, but you will receive credit for any parts of the problem set completed, even if you did not complete the prerequisites for that part. So, work ahead if you are stuck on any component of the problem set.

This problem set will have you write tests for methods, then implement those methods. We will be grading both the tests you wrote, as well as the methods themselves. We expect you to write as many tests as necessary to test the methods as we have defined them in the specifications, keeping in mind that many small tests are more useful than a few large tests.

Do not change the signatures or specifications of any methods, classes, or packages that we have provided you. Your code will be tested automatically, and will break our testing suite if you do so.

This problem set is designed so that it can be done with completely stateless functions. All of the functions we have provided you to implement are labeled static. You shouldn't need to introduce class variables or non-static functions in this problem set.

Overview

The theme of this project is to find English words in the digits of Pi. To do this, we'll:

1. Compute the fractional digits of Pi
2. Convert the digits of Pi into a numeric base more suitable for word-finding -- i.e., base 26.
3. Transform the digits of Pi into an alphabetic String of letters
4. Find words in that particular encoding of Pi

After the basic version of the code works, we'll work on improving it to get better word coverage -- changing the mapping of digits to letters so that you're more likely to find interesting words.

This problem set is designed to acquaint you with the notion of test-first programming, so your workflow for this problem set should be:

1. Carefully look at the specification of the method you are looking to implement.
2. Write a battery of tests to test the method you are about to write, checking all of the edge conditions for the specification.
3. Write the actual method we specified.

Problem 1: Pi Generation

This part of the problem set will generate an arbitrary number of digits of Pi, so we have data to search through. We will be implementing the Bailey–Borwein–Plouffe formula. Because this algorithm lets us generate arbitrary digits of Pi, it is much easier to test than other Pi generation methods.

Note that the BBP algorithm returns digits of Pi in base-16. Throughout Computer Science, base-16 is commonly referred to as hexadecimal (or hex). Hex is usually expressed using 0-9 and A-F to represent digits, with A = 10, B = 11, ..., F = 15. You'll see that terminology used in the problem set.

We have provided you with almost all of the implementation of the algorithm, the bulk of which can be found in PiGenerator.java. The implementation was ported from the integer-version of the Python code located here. You can read the explanation at that link for the spirit of the implementation we have provided you.

For this part of the problem, you'll have to test and implement computePiInHex() and powerMod() in PiGenerator. Assume that the implementation of piTerm() and piDigit() we've provided you are correct.

a. [5 points] Implement tests for powerMod() and place them in PiGeneratorTest. One test has been added for you already.

b. [10 points] Implement powerMod() in PiGenerator.

c. [5 points] Implement tests for computePiInHex() and place them in PiGeneratorTest. You

can find some of the hexadecimal expansion of Pi here, if you want to use it as a reference for your test cases.

d. [5 points] Implement computePiInHex() in PiGenerator. Note that this function should only return the fractional digits of Pi, and not the leading 3.

Executing Main.java now should print you some of the hexadecimal digits of Pi. You should verify that the output is what you expect. You can modify PI_PRECISION at the top of the file to change how many digits of Pi to generate; generating less digits will be faster.

Commit to Subversion. Once you're happy with your solution to this problem, commit your code! Committing frequently -- whenever you've fixed a bug or added a working and tested feature -- is a good way to use version control, and will be a good habit to have for your team projects.

Problem 2: Transforming Pi

In order to find words in the digits of Pi, we'll first need to convert the hex digits of Pi into something more useful. As a first try, we'll convert the hex digits into base-26, then do the straightforward mapping of numbers to letters. This part of the problem set will implement BaseTranslator.convertBase()

a. [10 points] A starting test has already been written for you in BaseTranslatorTest.java. Add additional tests to BaseTranslatorTest that tests the functionality of convertBase(). Ensure that all boundary conditions are tested, and that the behavior described in the specifications for those functions are complied with.

b. [15 points] Implement convertBase(), verifying that the tests you've written for it pass.

Executing Main.java should print you the base-26 digits of Pi. Verify that the numbers are what you expect.

Problem 3: Converting Pi to Characters

Now that we have Pi in base-26, it is a straightforward task to convert it to a string of letters. This part of the problem set will implement DigitsToStringConverter.convertDigitsToStrings().

a. [5 points] A starting test has already been written for you in DigitsToStringConverterTest.

java. Add additional tests to DigitsToStringConverterTest that tests the functionality of convertDigitsToString().

b. [10 points] Implement convertDigitsToString()

Executing Main.java should print you the translation of base-26 Pi into a-z characters.

Problem 4: Finding Words

Now that we have an alphanumeric string, we want to find words in it.

a. [5 points] A starting test has already been written for you in WordFinderTest.java. Add additional tests to WordFinderTest that tests the functionality of getSubstrings(). Ensure that all boundary conditions are tested, and that the behavior described in the specifications for those functions are complied with.

b. [5 points] Implement getSubstrings(), verifying that it conforms to the listed specification, and that all of your tests pass.

Executing Main.java now should show you a list of the words that were found in the digits of Pi! It should also tell you what percentage of words were found from the word list included in the assignment.

Problem 5: Alphabet Generation Revisited

As you can see, we don't do very well with finding most words in the digits of Pi. Part of the reason is that the alphabet we're using is not very smart; "z"s occur with roughly the same frequency as "e"s. This problem will explore one way to improve our implementation.

We'll use the word list that is included in the code to take a guess at how often each character occurs relative to the other characters, and we'll weigh the output alphabet we're translating with in favor of more frequently occurring characters.

a. [10 points] Implement tests for generateFrequencyAlphabet() in AlphabetGeneratorTest.

b. [15 points] Implement generateFrequencyAlphabet().

Executing Main.java now should show a list of words that were found in the digits of Pi using

the alternative alphabet. The coverage you should get for this implementation should be higher than in the basic one.

Before You're Done...

Double check that you didn't change the signatures of any of the code we gave you.

Make sure the code you implemented doesn't print anything to System.out. It's a helpful debugging feature, but writing output is a definite side effect of methods, and none of the methods we gave you to write should produce any side effects.

Make sure you don't have any outdated comments in the code you turn in. In particular, get rid of blocks of code that you may have commented out when doing the pset, and get rid of any TODO comments that are no longer TODOs.

Make sure your code compiles, and all the methods you've implemented pass all the tests that you added.

Does your code compile without warnings? In particular, you should have no unused variables, and no unneeded imports.

Make sure you check the last version of your code in SVN is the version you want to be graded.

6.005 Elements of Software Construction

Fall 2011