COMP 1012 Winter 2015 Assignment 2

Due Date: Wednesday, February 4, 2015, 11:59 PM

New Material Covered

- `while` loops
- sums
- multi-precision computation
- simple `list`

Notes:

- Name your script file as follows: `<LastName><FirstName>A2Q1.py`. For example, `LiJaneA2Q1.py` is a valid name for a student named Jane Li. If you wish to add a version number to your file, you may add it to the end of the file name. For example, `SmithRobA2Q1V2.py` is a valid name for Rob Smith's file.
- Follow the posted programming standards to avoid losing marks.
- You must submit a ***Blanket Honesty Declaration*** to have your assignment counted. This one honesty declaration applies to all assignments in COMP 1012.
- To submit the assignment follow the instructions on the course website carefully. You will upload both script file and output via a dropbox on the course website. We will demonstrate the assignment hand-in procedure in lectures. There will be a period of several days before the due date when you can submit your assignment. ***Do not be late!*** If you try to submit your assignment within 48 hours ***after*** the deadline, it will be accepted, but you will receive a penalty (roughly 1% per hour).

Question 1—Getting to the Root of Things [14 marks]

Description

Most computations can be done in various ways. They differ in computation speed and ease of programming, among other things. The virtue of programming is that once someone has a good computation scheme, they put it in a library[1] and everyone else just uses it. When you are a student, however, it is a good idea to find out what other people have done before you start using their creations. This question addresses the question of how to compute a square root.

Find the square root of your student number[2] s with Python in as many ways as you can. Here are some approaches you must use:

Simple math operations: $s^{0.5}$

Search: Using loop, keep trying consecutive integer values starting with 1 until you find a number with a square greater than s.

Functions: sqrt(s) in both the floating point math library and the complex number cmath library, and any other library you can find with a square root function.

Logarithms: Before electronic calculators became available, people would take powers on paper using log base 10 tables. For example, to calculate $7654321^{0.5}$, look up 7654321 in a log table, then multiply the

---

[1] Actually with square roots they've coded the scheme directly onto the computer chips since the late 1970s.

[2] Why your student number? It is an exact number (not an estimate), so the square root has an exact value, and it differs for each student, so you each get a different answer.

logarithm by 0.5 (or divide by 2), and then consult the log table again to determine the value of the result. This procedure uses this math identity:

$$x^y = 10^{y \cdot \log_{10} x}$$

You don't have to do it on paper, but simulate the process in Python.

Factoring: Run a `while` loop that applies the following scheme over and over until the remaining value of n_j is between 0.4 and 1.6.

$$\begin{aligned}\sqrt{n_0} &= \sqrt{4\frac{n_0}{4}} \\ &= 2\sqrt{n_1} && \text{where } n_1 = n_0/4 \\ &= 4\sqrt{n_2} && \text{where } n_2 = n_1/4 = n_0/16 \\ &= \ldots\end{aligned}$$

Babylonian method[3]**:** If y_i is an approximation to $\sqrt{x}$, then we can get a much better approximation to $\sqrt{x}$ by using the *iteration* (repeated calculation): $y_{i+1} = (y_i + x/y_i)/2$. We can keep applying this iteration over and over getting a better approximation every time until we can't tell the difference between y_i and x/y_i . The factoring process above gives us a good starting point if we assume that that the final n_i = 1.

Take for example a fictitious student, Aaron Ways, with student number 7654321, having a square root 2766.6… . If we apply the factoring above, we find that 7654321 = 4^{12} 0.45623308420181274. Assuming the last factor is 1, we can start with an approximation $y_0 = 4^{12} = 4096$, and then y_1 = (4096 + 7654321 / 4096)/2 = 2982.365…, which is much closer to the square root. The sequence converges quickly after that; how would you know when to stop?

Infinite Series:

$$(1+x)^{1/2} = 1 + \left(\frac{1}{2}\right)\frac{x}{1} + \left(\frac{1}{2}\right)\left(\frac{-1}{2}\right)\frac{x^2}{1\cdot 2} + \left(\frac{1}{2}\right)\left(\frac{-1}{2}\right)\left(\frac{-3}{2}\right)\frac{x^3}{1\cdot 2\cdot 3} + \left(\frac{1}{2}\right)\left(\frac{-1}{2}\right)\left(\frac{-3}{2}\right)\left(\frac{-5}{2}\right)\frac{x^4}{1\cdot 2\cdot 3\cdot 4} + \ldots$$

How does this series help? It is only valid if x is small ($|x| < 1$), and student numbers are seven digits long! Consider again the fictitious student, Aaron Ways, with student number 7654321, having a square root 2766.6… . If we apply the factoring above, we find that 7654321 = 4^{12} 0.45623308420181274 and so $\sqrt{7654321} = 2^{12}\sqrt{0.45623308420181274} = 4096\sqrt{1 - 0.5437669157981873}$ which can be evaluated using the series above, using x = –0.5437669157981873. We'll talk in class about how you evaluate a series like this.

Stop the while loop used to sum this series when the term magnitude gets too small to matter. Make sure to check the small value in your convergence criterion to ensure that it is as small as necessary, but no smaller.

When evaluating this series, store the values of the terms in a list, and print out the sum of the series adding them up forwards and backwards. Since the terms are declining in magnitude, summing backwards should give a slightly more accurate answer because of less roundoff error.

Multi-Precision Babylonian method: When you use float calculations, the maximum accuracy you can ever get is about 16 digits. To get more correct digits, your computations must be done using ints, which have unlimited precision in Python (but not in most other languages). You get more precision by multiplying fractional numbers by large integers. For example, to find $\sqrt{7654321}$ to 30 decimal places, first multiply the student number by $10^{2\times 30}$, and multiply the initial approximation by 10^{30}. Here is the example with Aaron Ways again. We now want the square root of 7654321×10^{60}, starting with an approximation of 4096×10^{30}. Then y_1 = $(4096\times 10^{30} + 7654321\times 10^{60} / (4096\times 10^{30}))/2$ = 298236535644531250000000000000000000 which is much closer to the square root.

---

[3] According to clay tablets from almost 4000 years ago, similar methods were used by ancient Babylonians. This approach is also an example of Newton's method, a more general root-finding procedure.

When you reach the final estimate, you *must not* divide it back by `10.**30` to get a float because you would lose all the precision. Instead, print out `root // BIG_NUM` followed by `'_'` followed by `root % BIG_NUM`, where `BIG_NUM` is `10**30`. The result isn't a `float`, but resembles one.

Multi-Precision Infinite Series:

$$10^D(1+x)^{1/2} = 10^D + 10^D\left(\frac{1}{2}\right)\frac{10^D x}{1\cdot 10^D} + 10^D\frac{(1)(-1)}{2^2}\frac{\left(10^D x\right)^2}{1\cdot 2\cdot (10^D)^2} + 10^D\frac{(1)(-1)(-3)}{2^3}\frac{(10^D x)^3}{1\cdot 2\cdot 3\cdot (10^D)^3} + \ldots$$

Compare the series above with the previous version of the series. In the series above, each term has been multiplied by 10^D. To get 30 decimal places, you would use D = 30. If you get each term by multiplying the previous term by several factors, as demonstrated in class, you need only multiply the first term by 10^D. In addition, the value of x has been multiplied by 10^D, as it was originally a decimal fraction between –0.6 and 0.6. When multiplying by x in the loop then, it is also necessary to divide the term by 10^D in the loop as well.

If you want to match the results from the Babylonian method, you will need to use several guard digits. For instance, instead of using D=30, you could use D=36, and discard the last 6 digits of the result. This approach protects you from round-off errors when adding up the series. It was used in the sample output below.

Feel free to find other methods of computing a square root and use them in your code, but remember, you don't have much time.

Sample Output

Our hypothetical student Aaron Ways with student number 7654321 completes and runs his code, and produces the following output. Your output should look quite similar (but with correct answers for your student number). That is, your output should have the same words and the same formatting (including blank lines, number of digits, alignment), but different numbers. You may also have additional lines of output for different square root methods. Put them after the required ones.

```
Finding the square root of 7654321

Using the power operator: 2766.644357339772

Using search with increment 1, the root lies between 2766 and 2767

Using math.sqrt:          2766.644357339772
Using numpy.sqrt:         2766.6443573397719
Using cmath.sqrt:
    sqrt( 7654321) =      (2766.644357339772+0j)
    sqrt(-7654321) =      2766.644357339772j
Using numpy.lib.scimath.sqrt:
    sqrt( 7654321) =      2766.6443573397719
    sqrt(-7654321) =      2766.6443573397719j

Simulating the manual calculation of square root using logarithms:
    log10(7654321) =      6.8839 rounded to 4 decimals
    log10(root) =         3.44195
    root =                2767 to 4 significant figures

Finding an approximate root by factoring; the root is:
                            1 * sqrt(7654321)
                            2 * sqrt(1913580.25)
                            4 * sqrt(478395.0625)
                            8 * sqrt(119598.765625)
                           16 * sqrt(29899.69140625)
                           32 * sqrt(7474.9228515625)
                           64 * sqrt(1868.730712890625)
                          128 * sqrt(467.18267822265625)
```

```
                            256 * sqrt(116.79566955566406)
                            512 * sqrt(29.198917388916016)
                           1024 * sqrt(7.299729347229004)
                           2048 * sqrt(1.824932336807251)
                           4096 * sqrt(0.45623308420181274)

Using the Babylonian method:
     1: root estimate:    4096
     2: root estimate:    2982.3653564453125
     3: root estimate:    2774.4461428176182
     4: root estimate:    2766.6553267105055
     5: root estimate:    2766.644357361518
  final root estimate:    2766.644357339772

Using infinite series with x = -0.5437669157981873:
    count:   0      term: 1.0
    count:   1      term: -0.27188345789909363
    count:   2      term: -0.03696030733958411
    count:   3      term: -0.010048896164499378
    count:   4      term: -0.0034151607965912873
    count:   5      term: -0.0012999360173221273
    count:   6      term: -0.000530146649155674
    count:   7      term: -0.00022650273511809783
    count:   8      term: -0.00010007131362720523
    count:   9      term: -4.534622464244875e-05
    count:  10      term: -2.09591102093787e-05
    ... [lines removed]
    count:  51      term: -2.5021075547401773e-17
    count:  52      term: -1.3213162895353598e-17
    Sum forwards is     2766.644357339771
    Sum backwards is    2766.644357339772

Using the Babylonian method and integer calculations:
     1: root estimate:    4096_00000000000000000000000000000000
     2: root estimate:    2982_36535644531250000000000000000000
     3: root estimate:    2774_44614281761808150491105000 8847
     4: root estimate:    2766_65532671050566523651953 8024543
     5: root estimate:    2766_64435736151794556145971 5109927
     6: root estimate:    2766_64435733977119960153570033 00840
  final root estimate:    2766_64435733977119960152715412 28274

Using infinite series with x = -0_54376691579818725585937500000000000000:
    count:   0     total: 0_00000000000000000000000000000000
    count:   1     total: 1_00000000000000000000000000000000
    count:   2     total: 0_72811654210090637207031250 0000
    count:   3     total: 0_69115623476132226343793263 3678
    count:   4     total: 0_68110733859682288622535472 0854
    count:   5     total: 0_67769217780023159890019075 3765
    count:   6     total: 0_67639224178290947164572784 3431
    count:   7     total: 0_67586209513375379760008597 0590
    count:   8     total: 0_67563559239863569975537129 4326
    count:   9     total: 0_67553552108500849452477106 4472
    count:  10     total: 0_67549017486036604577849632 0799
    ... [lines removed]
    count: 115     total: 0_67545028255365527246466590 3620
    count: 116     total: 0_67545028255365527246466590 3620
    count: 117     total: 0_67545028255365527246466590 3620
    count: 118     total: 0_67545028255365527246466590 3620
```

```
    Final estimate is     2766_644357339771996015271541228274

Programmed by Aaron Ways
Date: Thu Jan 22 23:36:12 2015
End of processing
```

Handin

You will hand in your program script file. It should take only one file to produce all the output demonstrated above. If you cannot get all the parts to work, hand in what you have. Also, as in Assignment 1, use one txt file to hand in the output produced by your code for the sample student number 7654321, and for your own student number.. In the multi-precision part, print your results to 40 decimal places, rather than 30 as shown in the example.