Assignment 1

Prof. G.Srinivasaraghavan

From: March 1, 2022	**To**: March 31, 2022, Midnight	**Max Marks**: 30

Q-1: **Final Goal**: Generate digits of π up to any given precision (number of decimal digits after the decimal point). Your implementation must be limited, in principle, only by the hardware (speed and memory) on which your code is running.

Details: There are several algorithms / numerical schemes for generating the digits of π, the current world record being 62.8 trillion digits using what is called the *Chudnovsky Algorithm* `https://en.wikipedia.org/wiki/Chudnovsky_algorithm`. You are free to use any of the several algorithms that are known as long as it allows you to generate an arbitrarily large number of correct decimal digits of π. Here's a link that gives the first 1,00,000 digits of π – `http://www.geom.uiuc.edu/~huberty/math5337/groupe/digits.html`. This assignment will certainly exercise your implementation skills and your understanding of the algorithms for integer arithmetic we discussed earlier. It will also force you to think about the issues (numeric, underflow, overflow, memory management, etc.) involved in implementing arithmetic routines on arbitrary precision numbers. Please note that the preferred implementation language is `C++`. As discussed in the class you can use the ready made arithmetic provided by `C++` only for arithmetic on individual digits of the number in some base B, if we have to preserve the digits up to any arbitrary precision.

However for this assignment I recommend the following algorithm (it is one of a family of algorithms broadly known as Borwein's Algorithms – `https://en.wikipedia.org/wiki/Borwein%27s_algorithm#Quadratic_convergence_(1984)`, that converges exponentially fast, i.e., the number of correct digits that it can generate approximately doubles in every iteration. I suggest this primarily for its ease of implementation (compared to the others that I tried) and how quickly one can see tangible results with the correct digits of π popping out. For example using my implementation I could get 256 correct digits in just 7 iterations with the code running for about 2 Minutes on my laptop. The algorithm is described below in Algorithm 1.

Algorithm 1 Borwein's *exp-2* Algorithm

```
0: $a_0 = \sqrt{2}, b_0 = 0, p_0 = 2 + \sqrt{2}$
0: while (Precision not obtained) do
0:     $a_{n+1} = \frac{\sqrt{a_n} + \frac{1}{\sqrt{a_n}}}{2}$
0:     $b_{n+1} = \frac{(1+b_n)\sqrt{a_n}}{a_n + b_n}$
0:     $p_{n+1} = \frac{(1+a_{n+1})p_n b_{n+1}}{1+b_{n+1}}$
0: end while=0
```

The p_n in Algorithm 1 will give closer and closer approximations to π as n increases – p_n will have at least N correct digits of π when n is approximately $\ln N$ or more.

Implementing this will require you to do the following.

1. Implement basic integer arithmetic (addition, subtraction, multiplication, division) on numbers represented in some (configurable) base B — assume that B is of the form 2^m

for some m. In my implementation the value of m that worked best (running time) was 8. It would help to use the `vector` class that is available in `C++` STL (Standard Template Library) as the base class for your implementation. The `vector` class in `C++` provide functionality similar to a `list` in Python — it's effectively a dynamic array to which you can add elements, remove elements, and index into any of the vector elements. These implementations must be asymptotically of the same complexity as the ones discussed in the lectures. Adding Karatsuba and Toom3 to your implementation will get you some decent brownie points!!

2. Extend the integer algorithms discussed in the course to algorithms that work with arbitrary precision real numbers. This is not as difficult as it sounds – the underlying data structure will still be an array (`vector` in `C++`); you only need to keep track of the how big is the fractional part. All arithmetic can be done by treating the numbers as integers and adjusting for the fractional part at the end. For example: $3.54*98.2307 = (354*982307)*10^{-6}$. Similar strategy should work for any base other than 10.
3. Use the real number arithmetic routines to implement an algorithm for finding the square root of a number to any given degree of precision. Again based on what I have experimented with, I suggest the Newton-Raphson iterative method for finding the square root. The algorithm is given below as Algorithm 2.

Algorithm 2 Newton's Approximation to find $\sqrt{R}$

```
0: Initialize $x_0$ Randomly (of course closer it is to $\sqrt{R}$ the better)
0: while (Precision not obtained) do
0:     $x_{n+1} = \frac{1}{2}\left(x_n + \frac{R}{x_n}\right)$
0: end while=0
```

The idea behind the Newton's method is simple to describe. If the current estimate is x_n, get the next estimate by drawing a tangent to the curve $x^2 - R = 0$ at $x = x_n$. x_{n+1} is the place where the tangent meets the x axis. It is not difficult to show that $\forall n$, $x_0 > x_1 > \dots > x_n > \sqrt{R}$ and that $x_n - \sqrt{R} \leq \frac{1}{2^n}\frac{\left(x_0 - \sqrt{R}\right)^2}{x_0}$ (try proving this yourself using induction). Of course we are only interested in the positive root of R here. Square root is in fact a good test to see if your basic arithmetic implementation is working fine. Check your implementation on $\sqrt{2}$ and compare your answer with what is given in `https://apod.nasa.gov/htmltest/gifcity/sqrt2.1mil` which has the first million digits of $\sqrt{2}$!!

4. Put all the above together to implement Algorithm 1 (or any other algorithm for π that you prefer, if any). However please do note that the algorithm you implement must be capable of producing any number of correct digits of π as needed. BTW almost all the algorithms that you come across for approximating π will involve square roots — the formula needs to have an irrational term somewhere!!
5. Allow input to be given as a decimal string and output the result as a decimal string again. Note that the internal representation is in some other base $B = 2^m$ for some m — so this requires you to convert from base 10 to base B and vice-versa.

Check out `www.sagemath.org`. This is a nice tool that incidentally also provides Python bindings. You will be expected to use some such tool for (sage-math provides a convenient wrapper to many of the other tools for arbitrary precision arithmetic) the next assignment. The ground-up implementation for this assignment is just for you to get a feel.