CSCI 262 Data Structures

Spring 2019

Lab 6: Ancient Algorithms

(Back to Labs)

Goals

- Learn some new (old) algorithms
- Practice implementing algorithsm from descriptions
- Appreciate the wisdom of the ancients!

Overview

According to Wikipedia, an algorithm is "an unambiguous specification of how to solve a class of problems". While we think of algorithms in connection with modern computer programming, in fact algorithms have been around for millenia. (The word "algorithm" itself comes from the name of a Persian scholar, Al-Kwharizmi, who lived roughly 780-850 C.E. His book on calculations using the Indian numerals - what we now call Arabic numerals - was translated into Latin in the 12th century, influencing European and world mathematics for all time.)

One of the more famous of ancient algorithms, still in use today, is Euclid's greatest common divisor (GCD) algorithm. You've played with that algorithm, more or less, if you did the ContinuedFractions APT. There are many other algorithms, however, which are either still in use today, or have been enormously influential along the way to modern algorithms. Below, we will explore four of these algorithms: Egyptian multiplication, the Babylonian method for finding square roots, the Sieve of Eratosthenes, and Archimedes' method for approximating pi.

Your task for this lab is to pick one of the described algorithms and implement it on the modern computer using C++. Of the four algorithms, Archimedes' approximation of pi is the least accessible, so I don't recommend implementing it unless you really want to or feel especially ambitious.

Instructions

Read through the sections on the different algorithms below, and choose one to implement. What you are expected to do for a given algorithm is detailed in the corresponding section. Your resulting program should be submitted on Canvas.

Egyptian multiplication

Variants of this approach for multiplying two integers have been discovered and rediscovered by many cultures. It was used in the building of the pyramids, yet it is still useful today in the digital logic circuits used to multiply binary numbers. It has a wonderful connection, in fact, to the binary numbering system, in that its essential mechanism is the decomposition of one multiplicand into a sum of powers of two (effectively its binary representation).

The basic method (on paper) goes as follows: create two columns; write one multiplicand (preferably the smaller) at the top of the left column, and the other multiplicand at the top of the right column. You can now fill in the left column by dividing by two repeatedly until you get to 1. On each division, you discard any remainder (i.e., when dividing an odd number), so you only have integers in that column. Meanwhile, the right column is filled by doubling its numbers until you have the same count of numbers in the right column.

To complete the multiplication, strike out any rows in your table for which the left-hand number is even. Add up the remaining numbers in the right column to get the answer. For example, if we want to multiply 238 x 13, our table would look like:

13	238
6	476
3	952
1	1904

Crossing out the rows with even numbers in the left column leaves us with the sum 238 + 952 + 1904 = 3094.

Why does this work? If you look at the left-hand column in binary, you may get an intuition:

1101	238
110	476
11	952
1	1904

Each time, the right-most digit of the left-hand column tells us whether or not we are going to keep the corresponding power of 2 times the right-hand multiplicand in our final sum! Note that the only operations we really performed involved multiplying or dividing by 2, which are easily accomplished in the computer by shifting bits left or right.

If you choose to implement this algorithm, then your program should prompt the user for two (positive) integers to multiply. Display a representation of the two columns you generate, and the final answer. E.g., your program output might look like the following:

```
Please enter two positive integers: 238 13


13  238
```

```
 6  476
 3  952
 1 1904

Answer: 3094
```

To get your numbers to line up nicely in right-aligned columns, use `setwidth` (http://www.cplusplus.com/reference/iomanip/setw/).

The Babylonian method

Also known as "Hero's method" after the Greek mathematician whose description of it survived to the modern era, the Babylonian method is an iterative approximation method for finding square roots. Iterative approximation algorithms can be used when closed-form solutions are not readily available, and work by repeated refinement of a solution towards the true value. These algorithms have many applications in scientific computing.

The Babylonian method is very simple. Given a number *n*, and a guess at its square root, *x*, we make the observation that if *x* is an overestimate of the square root of *n*, then *n* / *x* will be an underestimate of the square root, and vice versa (convince yourself of this with a bit of algebra). Then, it is reasonable to expect that the average of these two numbers is a better estimate of the square root than we started with.

The Babylonian method proceeds, then, by letting *x* be the average of *x* and *n* / *x*, repeating until the change in values is very small, or until a fixed number of iterations have occurred. The value *x* will converge on the true square root.

For this algorithm, your program should prompt the user for a number to take the square root of (you should allow floating point or integer values) and an initial guess, and apply the Babylonian method, printing out each estimate of the square root until the change in estimates is less than 1e-8. (If you want to see more decimal places in your approximation, use the `setprecision` manipulator (http://www.cplusplus.com/reference/iomanip/setprecision/). For example:

```
Enter a number: 7234
What is your guess? 1000
503.617
258.9905451
143.4610333
96.94293788
85.78207742
85.05602362
85.05292476
85.0529247
85.0529247
```

The Sieve of Eratosthenes

The Sieve of Eratosthenes is an ancient method for finding prime numbers. Prime numbers are crucial in many algorithms, particularly in cryptography, so methods for finding them are quite important; sieve methods descended from the Sieve of Eratosthenes are still used.

To apply the sieve, one must first specify a maximum number; the sieve will find all primes less than or equal to the specified number. On paper, the sieve is applied as follows:

1. Write down all numbers between 2 and the specified maximum
2. 2 is the first prime. All numbers that are multiples of 2 are not prime by definition, so cross out every other number, starting with the second number after 2 (that is, all even numbers except 2).
3. The next number that isn't crossed out (3) is the next prime - we know this, because if it were not prime, it would have been crossed out as a multiple of some previous prime. So proceed again by crossing out each 3rd number following 3 (be sure to not cross out 3).
4. Continue this procedure - pick the next number not crossed out, say *n*, and cross out all of its multiples - every n^{th} number.
5. The numbers which remain are the prime numbers from 2 to the specified maximum.

This algorithm runs extremely fast if implemented correctly - but it is easy to make inefficient choices! The trick is to use a vector that is as large as the maximum number, and have the indices represent the number (you don't need to actually store the numbers). Each index of the vector only needs to contain an indicator of whether or not the number is still a prime candidate - so a vector of `bool` works fine.

Next, and this is important, *do no divisions or modulo operations!* Doing these is unnecessary and will kill your performance. Instead, for a given prime *n*, you only need to look at every n^{th} index after *n* itself - change the value at the index to indicate that the number is not prime.

For reference, on my laptop, my implementation can find all primes up to 10,000,000 in just over 1 second.

For this algorithm, prompt the user for a maximum number, and print out a sequence of all the prime numbers less than the specified number, e.g.

```
Input a maximum number: 30


2 3 5 7 11 13 17 19 23 29
```

Archimedes' approximation of pi

Archimedes' method is easily explained, but working out the mathematics is a bit more complicated. It is

another example of an iterative approximation algorithm.

The very short explanation of Archimedes' approach is that he starts by inscribing a regular hexagon inside a circle (touching the circle at all the vertices) and another regular hexagon outside the circle (so the circle is tangent to all of the sides of the hexagon); the area of the circle must then be between the areas of the two hexagons. Now, double the number of sides of each polygon to twelve - and do it again - and again... each time, the polygons get closer and closer to the actual circle.

Technically this approach will work with any polygons, but hexagons have some useful properties as a starting point, and Archimedes' ingenuity is really in how he works out the length of the sides and the apothems of the polygons after doubling the number of sides.

A nice video that helps illustrate the mathematics can be found here. Note that in this video, only the polygon inside the circle is used, but with the power of a computer, it is possible to make a polygon with a very large number of sides indeed, thus approximating the circle even better than Archimedes' approach would. If you implement this algorithm, you should probably use this approach (all the important math occurs by about 6:30).

Note, a proper implementation of Archimedes' method should **not** use any of the trigonometric functions from the <cmath> library! You can do all of the math using nothing more complex than `sqrt` - hey, you could even use your solution to the Babylonian method, above!

If you choose this algorithm, then simply print out your successive approximations to pi until the change in values is very small (~1e-12); it is also interesting to print out the number of sides your polygon has at each step. You'll need to use the `setprecision` manipulator (http://www.cplusplus.com/reference/iomanip/setprecision/) to see all your pi digits!

Wrapping Up

To complete this assignment, please submit your code, along with a README, to Canvas. Your README should include your name and the names of your partner(s), and any other comments you'd like to include about the lab.