**Concordia University**



**SOEN 6441  
Advanced Programming Practices**

**Fall 2010**

**Professor Pankaj Kamthan**

**Incredible Prime Root (IPR)**

**Artifact 2**

**Team B**

**Baljinder Kaur Multani**

**Jalal Noureddine**

**Sergey Khechumov**

**Table of Contents**

1. Root Fiding Alogorithms 3

2. Primality tests 8

3. CRC Cards 10

4. Appendix

Meetings 18

References 19

**1 Root Finding Algorithms**

There are multiple algorithms for calculating the root of a number. However, they have some differences in properties like speed, complexity, initial conditions, and difficulty to understand and to implement. We have chosen some algorithms and some important properties as criteria of choosing an algorithm. For this table we used system of marks from 0 (worst) and 5 (best). The results are shown in the table below.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | Computational complexity | Easy to understand | Easy to implement | Stability | Correctness | Average |
| Taylor's series | 4 | 3 | 1 | 5 | 5 | 3.6 |
| Shifting Nth (digit by digit) algorithm | 3 | 4 | 3 | 5 | 5 | 4 |
| Newton's method | 5 | 5 | 5 | 5 | 5 | **5** |
| Secant method | 4 | 5 | 5+ | 5 | 5 | 4.9 |

All these algorithms are stable and correct (for our functions: Xd – n = 0) with a different complexity.

As we can see from a table, the best method from all these methods is Newton's Method. It is a fast and easy method that's convergence is proved, so it was chosen by our team. Descriptions of these algorithms are presented below.

**Taylor's series**

There is a method of computation of a numerical root using Taylor's series.

**Description:**

The general formula of decomposition of the function to a Taylor's series is:

*∑∞n=0 ((ƒ(n)(a) / n!) \* (x – a)n),* where n is a number of components of the sum, *ƒ* (*n*)(*a*) denotes the *n*th derivative of *ƒ* evaluated at a point *a.* In our case *a*=0 (Maclaurin's series)*.*

For example, if we want to compute √2, the formula will be: √x+1 = *∑∞n=0  ((-1)n(2n)!) / ((1-2n)(n!2)(4n))\*xn=*1 + x/2 – x2/8 + x3/16-...

**Advantages:**

- Does not need a table of initial values to find a root.

- This method is relatively hard to understand and to implement.

**Disadvantages:**

- This method is difficult to implement for every natural power of root (d), because we will have to use different formulas for every d.

- Using such function like a factorial, the result of that grows fast with increasing n (for n!).

**Shifting Nth root algorithm (digit by digit)**

**Description:**

The shifting *n*th root algorithm is an algorithm for extracting the *n*th root of a positive real number which proceeds iteratively by shifting in *n* digits of the radicand, starting with the most significant, and produces one digit of the root in each iteration, in a manner similar to long division. [1]

**Algorithm:**

Let's *y* is a root extracted thus far, *n* is a degree of the root to be extracted*, r* is a remainder, α next n digits of the radicand, β be the next digit of the root, B is a base of a using number system.

1. Initialize *r* and *y* to 0.
2. Repeat until desired precision is obtained:
   1. Let α be the next aligned block of digits from the radicand.
   2. Let β be the largest β such that http://upload.wikimedia.org/math/9/0/8/9089ec7f54c584c1f20934112b98a620.png
   3. Let *y*' = *By* + β.
   4. Let *r*' = *Bnr* + α − ((*By* + β)*n* − *Bnyn*).
   5. Assign http://upload.wikimedia.org/math/7/e/b/7ebcc94a1414f460889f6762b9491a6d.png and http://upload.wikimedia.org/math/0/3/f/03f112b62436a12d93075b4bee682e57.png
3. *y* is the largest integer such that *yn* < *xBk*, and *yn* + *r* = *xBk*, where *k* is the number of digits of the radicand after the decimal point that have been consumed (a negative number if the algorithm hasn't reached the decimal point yet).

The complexity of the algorithm is (for square root): *O*(*k*3*n*2log(*B*)).

**Advantages:**

- There are no division operations in the algorithm.

- We don't have to recalculate digits we computed thus far.

**Disadvantages:**

- In the step 2 we need to find “the largest β that...”. It will take a time.

- This method is relatively hard to understand and to implement.

**Newton's method**

**Description:**

As we tried to explore different methods and found the best known method for finding successively better approximations to the zeroes (or roots) of a real-valued function. We decided to choose Newton’s method because Newton's method can often converge remarkably quickly. Especially if the iteration begins "sufficiently near" the desired root. Just how near "sufficiently near" needs to be, and just how quickly "remarkably quickly" can be, depends on the problem. [2]

Let f: [a,b]->R be a differentiable function. The Newton's Method is defined by the equation

xn+1 = xn - ( f(xn) / f'(xn)). [3, page 10]

This method is “mathematically proved to converge” [3, page 12].

The Newton method is always linearly stable (but it may diverge if *x0* is sufficiently far from *x\**)! If *f'(x\*)* is non-zero, the absolute error *ek* reduces with iterations as *ek+1 = c ek2*, i.e. at the quadratic rate. Therefore, the Newton's method is fastly convergent (*second-order*) algorithm or algorithm of order of *O(h2)*. When the root is multiple, i.e. when *f'(x\*) = 0*, the rate of convergence becomes linear again, like in the contraction mapping method. [4]

**Advantages:**

- Easy to understand.

- Easy to implement.

- Convergence is quadratic.

**Disadvantages:**

- Needs a table of initial values.

- Have to recalculate digits we computed thus far.

- Fastest method we examined in present project.

**Secant method**

**Description:**

The secant method is a technique for finding the root of a scalar-valued function f(*x*) of a single variable *x* when no information about the derivative exists.

Formula: X=(Xn-1 \* f(Xn) – Xn \* f(Xn-1)) / (f(Xn) – f(Xn-1)).

It can be shown that the rate of convergence at a simple root is better than linear, but poorer than the quadratic convergence of Newton's method (Pizer, 1975) and is given by the golden-ratio (~1,618034). The rate of convergence is O(*h*1.618).

The method needs 2 initial approximation of the root. Suppose we have two approximations *xa* and *xb* to a root *r* of f(*x*), that is, f(*r*) = 0. We could approximate the function by interpolating the two points (*xa*, f(*xa*)) and (*xb*, f(*xb*)). [5]

**Advantages:**

- Easy to understand.

- Easy to implement (easier than Newton's method).

**Disadvantages:**

- Needs a table of initial values.

- Slower than Newton's method.

- Have to recalculate digits we computed thus far.

**2 Primality Tests**

To determine whether or not a given number is prime, we need to perform a primality test. Primality tests come in two categories: deterministic and probabilistic [6].

Deterministic tests determine with absolute certainty that a number is prime (e.g. naïve methods, Lucas-Lehmer test, and elliptic curve primality proving).

Probabilistic tests can potentially (although with small probability) falsely identify a composite number as prime (although not vice versa). However, they are in general much faster than deterministic tests (e.g. Fermat primality test, Miller–Rabin primality test, and Solovay–Strassen primality test).

Programmers should take into considerations many factors on choosing the best algorithm to implement. Some of the important factors include:

Certainty (i.e. choosing a deterministic test), implement ability (the method for primality testing must be practical to implement), efficiency (the implementation of a method must be efficient in time and space). [7]

There are infinitely many prime numbers [8]. A Java int value is a 32-bit number. MAX\_VALUE is a constant holding the maximum value an int can have, 231-1 and that's the largest value that can be represented in 32-bit two's complement. [9] In other words, performing a primality test using he Java programming language running on a modern computer has some limits.

The following are three naïve deterministic algorithms; each algorithm improves upon the previous one [7].

**Naïve Algorithms**

**Algorithm 1:**

if n is 1, return false;

else,

for all integers m from 2 to n − 1, check if m is a factor of n and, if it is, then n is a composite number;

if no factors of n are found, then n is a prime number

This algorithm becomes inefficient as n gets larger. It can be made more efficient by reducing the numbers to check by replacing n − 1 by √n. If n is not divisible by 2, then n is not divisible by any other even number. A prime number is always an odd number except 2 (the only even prime number). Therefore, all even m except 2 can be skipped.

**Algorithm 2:**

if n is 1, return false;

if n is 2, return true;

if 2 is a factor of n, return false;

else,

for all odd integers m from 3 to √n, check if m is a factor of n and, if it is, then n is a composite number;

if no factors of n are found, then n is a prime number

The algorithm 2 can be made more efficient by reducing the numbers to check. This can be done by using the following theorem:

Let n be a prime number > 3. Then, n = 6k ± 1, where k is a positive integer.

And this will result in algorithm 3.  
  
**Algorithm 3:**

if n is 1, return false;

if n is 2, return true;

if 2 is a factor of n, return false;

if 3 is a factor of n, return false;

else,

for all integers m of form 6k ± 1 to √n, where k is a positive integer, check if m is a factor of n and, if it is, then n is a composite number;

if no factors of n are found, then n is a prime number

All algorithms have limits towards efficiency as n gets larger. We decided to choose to implement the third algorithm, because it’s more efficient than others.

**3 CRC cards**

**Class Responsibility Collaborator (CRC)** model is a collection of CRC cards. A CRC cardsare a brainstorming tool used in the design of Object-Oriented Software. They are used for determining classes, their responsibilities and their collaborators which we are going to use in our programs. A CRC card is divided into three sections.

**1. Class:** - A class represents a collection of similar objects. A class name appears on the top of the CRC cards.

**2. Responsibility: -** A responsibility is anything that a class knows or does. The responsibilities of a class appear along the left side of the CRC card.

**3. Collaborator: -** A collaborator is another class that a class interacts with to fulfill its responsibilities. The collaborator of a class appears along the right side of the CRC card.

We have designed the following CRC cards for our program (IPR).

The white area represents the front side of the CRC card, and the light blue area represents the back side.

**1. Class representing a radicand**

|  |
| --- |
| **Radicand** |
| Knows radicand | | CheckForPrimality |
| Knows radicand validation criteria | |  |
| Verifies a radicand using these criteria | |  |
| Gets a radicand | |  |
| Sets a radicand | |  |
|  | |  |
|  | |  |
| Class represents a radicand the root of which the program will compute, | | |
| Verifies if a radicand is a number specified in requirements of the | | |
| program | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |

**2. Class for checking a number for primality**

|  |
| --- |
| **CheckForPrimality** |
| Knows primality criteria | | Radicand |
| Verifies if a number is prime | |  |
| Gets a radicand | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
| This class checks a number for a primality | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |

**3. Class representing a root degree**

|  |
| --- |
| **RootDegree** |
| Knows root degree | |  |
| Knows root degree verification criteria | |  |
| Verifies a root degree using these criteria | |  |
| Gets root degree | |  |
| Sets root degree | |  |
|  | |  |
|  | |  |
|  | |  |
| The class represents a root degree that will be used by a program. | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |

**4. Class representing a precision**

|  |
| --- |
| **Precision** |
| Knows precision | |  |
| Knows precision validation criteria | |  |
| Verifies a precision using these criteria | |  |
| Gets precision | |  |
| Sets precision | |  |
|  | |  |
|  | |  |
|  | |  |
| The class represents a precision of calculating the prime root. | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |

**5. Class representing a prime root**

|  |
| --- |
| **PrimeRoot** |
| Computes a root of a prime number | | Radicand |
| Gets a computed root of a prime number | | RootDegree |
|  | | Precision |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
| The class represents a prime root. The heart of the program. It will | | |
| compute this number with collaboration of other classes. | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |

**6. Class for saving the result in a file**

|  |
| --- |
| **SaveResult** |
| Gets a file type in which a number will be | | PrimeRoot |
| saved | |  |
| Creates a file | |  |
| Saves a root of a prime number in a file | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
| The class is responsible for saving numbers into files of different types. | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |

**7. Class representing a help of the program**

|  |
| --- |
| **Help** |
| Shows help topics | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
|  | |  |
| This class is responsible to show help topics to the user. | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |
|  | | |

**4 Appendix**

**Meetings**

We have used Doodle [10] to organize meetings times for A2.

1. **Meeting number 1**: <https://www.doodle.com/8n22qphecpytd3g8>  
   Wednesday 13 October 2010 6:00-9:07 PM

In the first meeting regarding A2, we spend all the time discussing the root-finding algorithms.

1. **Meeting number 2**: <https://www.doodle.com/whttq58fgdaaq28a>  
   Saturday 16 October 2010 12:30-3:35 PM

In this meeting we discussed primality tests as well as root-finding algorithms.

1. **Meeting number 3**: <https://www.doodle.com/4t3kp45qbt4r8c7v>  
   Monday 18 October 2010 5:30-8:20 PM

Most of the time in this meeting was spent on the CRC cards. We also reviewed the primality tests.

1. **Meeting number 4**: <https://www.doodle.com/n43ckrnesuxww5dz>  
   Wednesday 20 October 2010 6:00-7:15 PM

That was the last meeting, in which we reviewed A2 part by part, and fixed some errors in CRC cards and in the document’s structure. We also assigned more works to be done at home.

**References**

[1] Wikipedia contributors. "Shifting nth root algorithm." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 6 Oct. 2010. Web.

[2] Wikipedia contributors. "Newton's method." *Wikipedia, The Free Encyclopedia*. Wikipedia, The Free Encyclopedia, 4 Oct. 2010. Web.

[3] Pankaj Kamthan, On computing the square root of two. URL: http://users.encs.concordia.ca/~kamthan/courses/soen-6441/square\_root\_2.pdf.

[4] Convergence and stability of iterative methods, URL: http://dmpeli.math.mcmaster.ca/Matlab/CLLsoftware/NumMethods/Lecture1-3.html.

[5] University of Waterloo, Department of Electrical and Computer Engineering, Numerical Analysis for Engineering, The Secant Method. URL: http://www.ece.uwaterloo.ca/~dwharder/NumericalAnalysis/10RootFinding/secant/.

[6] Wolfram MathWorld, Primality Test. URL: http://mathworld.wolfram.com/PrimalityTest.html.

[7] Pankaj Kamthan, Primality. URL: http://users.encs.concordia.ca/~kamthan/courses/soen-6441/primality.pdf.

[8] Euclid's Proof of the Infinitude of Primes. An English translation of Euclid's proof that there are infinitely many primes can be found here: http://aleph0.clarku.edu/~djoyce/java/elements/bookIX/propIX20.html.

[9] URL: http://download.oracle.com/javase/1.4.2/docs/api/constant-values.html#java.lang.Integer.MAX\_VALUE.

[10] Doodle homepage URL: <http://www.doodle.com/>.