SOEN 6011 : SOFTWARE ENGINEERING PROCESSES
SUMMER 2021

SUPER CALCULATOR

PROBLEM - 3
Pseudo-code and Algorithms

Authors

Rokeya Begum Keya

Kyle Taylor Lange

Sijie Min

Manimaran Palani

https://www.overleaf.com/project/610304de4e6b8d24f7c781b6

Contents

Decision on Pseudo-Code Format

Our Team conducted a brainstorming session and referred several resources [2] [3] to decide on Pseudo code algorithm/pattern. As a conclusion, Everyone had agreed to make a pseudo code of their respective algoriths in the Algorithmicx (algpseudocode) [1] format available in Overleaf Latex.

Example of Algorithmicx (algpseudocode) Pseudo code Pattern

Algorithm 1 Algorithmicx (algpseudocode) Pseudo code Pattern

Require: $n \geq 0$
Ensure: $y = x^n$

```
y ← 1
X ← x
N ← n
while N ≠ 0 do
    if N is even then
        X ← X × X
        N ← N/2                                  ▷ This is a comment
    else if N is odd then
        y ← y × X
        N ← N − 1
    end if
end while
```

Algorithm Description and Pseudo-Code

PROBLEM 3 - F2: $tan(x)$

SOEN 6011 - Summer 2021 **Rokeya Begum Keya**
Software Engineering Processes **40183615**
Repository address : https://github.com/Dakatsu/SOEN6011Calculator

Technical Reasons for selecting Maclaurin Series:

- There are many reasons for selecting Maclaurin Series for calculating the value of $tan(x)$ function. Below are some advantages for which I selected Maclaurin Series:

Advantages:

- Maclaurin series provides more approximate values for the tangent function.
- The formula to calculate the value of $sin(x)$ and $cos(x)$ function to get the value of tangent function is easy to understand.
$$tan(x) = \frac{sin(x)}{cos(x)}$$

Disadvantages:

- There is an another form of Maclaurin series to calculate the tangent function. For example: derivation of $tan(x)$ function. In this formula there are no use of $sin(x)$ and $cos(x)$ function. However, using this formula we can not get the approximate value of $tan(x)$ function.

Therefore, I select the Maclaurin series of $sin(x)$ and $cos(x)$ to calculate the tangent function.

Algorithm 1 - Maclaurin Series:

- In this project to calculate $tan(x)$ function, I select the Maclaurin Series. Maclaurin series is just a special case of taylor series where region near $x = 0$.
- The $tan(x)$ function's approximation is derived by the Maclaurin Series's explicit forms of $sin(x)$ and $cos(x)$.
$$sin(x) = x - x^3/3! + x^5/5! - x^7/7! + ... \quad (1)$$
$$cos(x) = 1 - x^2/2! + x^4/4! - x^6/6! + ... \quad (2)$$
- As, $tan(x)$ is an odd function, odd derivatives when x=0 of Maclaurin series are used to calculate $tan(x)$ function.
- The output are in integer and provide an approximate value for tangent function.

Pseudo Code for Maclaurin Series

Algorithm 2 Maclaurin Series

Require: $retVal = 1$ AND $tmpResult = 1$ AND $i = 1$
  function EXPLICIT FORM(cos(x))
    for $i \leftarrow i + 2$ **do**
      $value = (-1) * x * x/(i * (i + 1))$ ▷ $Seriesforcos(x)$
      $tmpResult = tmpResult * value$
      if check($value$)<= EPS **then**
      end if
      $retVal = retVal + tmpResult$
    end for
  return $retVal$ ▷ $getvalueofcos(x)$
  end function
Require: $retVal = x$ AND $tmpResult = x$ AND $i = 0$
  function EXPLICIT FORM(sin(x))
    for $i \leftarrow i + 1$ **do**
      $value = ((-1) * x * x/((2 * i + 2) * (2 * i + 3)))$ ▷ $Seriesforsin(x)$
      $tmpResult = tmpResult * value$
      if check($value$)<= EPS **then**
      end if
      $retVal = retVal + tmpResult$
    end for
  return $retVal$ ▷ $getvalueofsin(x)$
  end function
Require: $x = Rad(x)$ AND $SinVal = retVal$ AND $CosVal = reVal$
  function CALCULATE(tan(x))
    if $SinVal < EPSvalMini$ **then**
   return 0
    end if
    if $CosVal < EPSvalMini$ **then**
   return $undefined$
    end if
  return $SinVal/CosVal$ ▷ $calculationfortan(x)$
  end function
  $result \leftarrow tan(x)$

PROBLEM 3 - F3: Hyperbolic Sine, $sinh(x)$

SOEN 6011 - Summer 2021 **Kyle Taylor Lange**
Software Engineering Processes **27627696**
Repository address : https://github.com/Dakatsu/SOEN6011Calculator

Exponentiation of integers is simple to implement in an algorithm, with 1 being multiplied or divided by a number (e) a certain number of times. Exponentation of non-real integer numbers is much more difficult as both methods considered to calculate them involve calculating roots of numbers. Calculating the root requires repeatedly testing guesses as to whether raising them to the n^th exponent will equal input number, and it can theoretically take infinite iterations to find the root. As such, there must be a balance between the precision of this value and the time spent calculating it.

One method for calculating real exponents involves the relation of the natural logarithm, $ln\ x$, with the exponential function, e^x. The other attempts to convert irrational real exponents into rational ones, e.g. $e^{\frac{a}{b}}$, where x^a is divided by $\sqrt[b]{x}$. This latter method was chosen as it is conceptually easier to comprehend and implement in code, and issues with computation time or accuracy can be adjusted by changing how well the rational number approximates the real number.

Given the above, the subordinate functions required to calculate $sinh(x)$ are the power function and the square root function. Additionally, a function to find the greatest common denominator can help reduce fractions to make them less intensive to compute. An absolute value function can also be created for simplicity.

Algorithm 3 Hyperbolic Sine

function SINH($input$)
  if $input = 0$ **then return** 0
  end if
  $intPart \leftarrow input \div 1$, $fracNum \leftarrow input \ mod \ 1$ ▷ Split into integral and real parts.
  $fracDen \leftarrow 1$
  while $fracNum > fracDen$ **do**
    $fracDen \leftarrow fracDen \times 10$
  end while
GCD($fracNum$, $fracDen$)
  $left \leftarrow$ POWER($e, intPart$), $right \leftarrow$ POWER($e, -intPart$)
  if $fracNum > 0$ **then**
    $numPower \leftarrow$ POWER($e, fracNum$)
    $leftRoot =$ ROOT($fracDen, numPower$)
    $left \leftarrow left \times leftRoot$
    $numCalc \leftarrow$ POWER($e, -numPower$)
    $rightRoot =$ ROOT($fracDen, numCalc$)
    $right \leftarrow right \times rightRoot$
  end if
return $\frac{left - right}{2}$
end function

Algorithm 4 Root

function ROOT(n, $base$)
  $step \leftarrow 0$
  if $base$ <1 **then**
    $step$ ←1 - base$_{\overline{2}}$
  else
    $step$ ←base$_{\overline{2}}$ + 0.5
  end if
  $result \leftarrow base$
  while $step \neq 0$ **do** ▷ Not equal ± some accuracy value.
    $resultSquared \leftarrow$ POWER($result, n$)
    if $resultSquared = base$ **then**
      break
    end if
    if $resultSquared$<$base$ **then**
      $result \leftarrow result + step$
    else
      $result \leftarrow result - step$
    end if
    $step \leftarrow \frac{step}{2}$
  end while
return $result$
end function

Algorithm 5 Power

```
function POWER(base, exp)
    result ← 1
    for i ← 0 to |exp| do
        if exp > 0 then
            result ← result × base
        else
            result ← result/base
        end if
return result
```

Algorithm 6 Greatest Common Denominator

```
Require: x ∈ Z AND y ∈ Z
  function GCD(x, y)
    if |y| > |x| then GCD(y, x)
    end if
    if x = 0 AND y = 0 then return 0
    end if
    for i ← xto0 do
        if x mod i = 0 AND y mod i = 0 then return i
        end if
    end for
  return 1
  end function=0
```

PROBLEM 3 - F7 : x^y

SOEN 6011 - Summer 2021 **Manimaran Palani**
Software Engineering Processes **40167543**
Repository address : https://github.com/Dakatsu/SOEN6011Calculator

Algorithm : Montgomery's Ladder Technique[4]

- Montgomerym's ladder technique addresses defence against side-channel attacks for exponentiation computation.

  The algorithm prevents the recovery of the exponent involved in the computation which could possibly benefit an attacker

- The algorithm performs a fixed sequence of operations (up to log n): a multiplication and squaring takes place for each bit in the exponent, regardless of the bit's specific value.

Advantages	**Disadvantages**
It addresses the concern of MIM(Middle Man attack) observing the sequence of squaring and multiplications can (partially) recover the exponent involved in the computation.	Cache timing attacks are not yet protected and memory access latency might still be observable to an attacker

Algorithm : Taylor series

Taylor series is a representation of a function as an infinite sum of terms that are calculated from the values of the function's derivatives at a single point.

$$x^y = e^{y \ln x} \tag{3}$$

3 evaluation of x^y. Here, e is a mathematical constant approximately equal to 2.71828

$$e^x = 1 + x/1! + x^2/2! + x^3/3! + ... \tag{4}$$

4 express e^x using Taylor Series

$$e^x = 1 + (x/1)(1 + (x/2)(1 + (x/3)(........))) \tag{5}$$

5 The series 4 can be re-written as above

$$log(1 + x) = x - x^2/2 + x^3/3 - ... \tag{6}$$

6 express ln x using Taylor Series

Advantages	Disadvantages
Very useful for derivations	Successive terms get very complex and hard to derive
Can be used to get theoretical error bounds	Truncation error tends to grow rapidly away from expansion point
Power series can be inverted to yield the inverse function	Almost always not as efficient as curve fitting or direct approximation

Algorithm 7 Montgomery's ladder Exponential Function

Require: $x_1 = x$; $x_2 = x^2$

for $i = k - 2$ to 0 *do* **do**
    if $n_{\mathrm{i}} = 0$ **then**
        $x_2 = x_1 * x_2$ ▷ $x_1 = {x_1}^2$
    else:
        $x_1 = x_1 * x_2$ ▷ $x2 = {x_2}^2$
        return x_1
    end if
end for

Algorithm 8 Exponentiation by Taylor Series

Require: $x \neq 0$ AND $y > 0$

function LOGARITHM(n) ▷ $algorithmforlog(n)$
    $sum \leftarrow 0$
    while $n > 1$ **do**
        $n \leftarrow n/e$ ▷ e is a constant approximately equal to 2.71828
        $y \leftarrow y + 1$
    end while
return y
end function
function EXPONENTIAL(x) ▷ $algorithmfore^x$
    $sum \leftarrow 1$
    $n \leftarrow 10$
    for $i \leftarrow n - 1, 1$ **do**
        $sum \leftarrow 1 + x * sum/i$
    end for
return sum
end function
$logx$ ←LOGARITHM(x)
$result$ ←EXPONENTIAL(y*logx)

Bibliography

[1] Algorithmicx (algpseudocode)
https://www.overleaf.com/latex/examples/pseudocode-example/pbssqzhvktkj

[2] Pseudo-Code Standard
http://users.csc.calpoly.edu/j̃dalbey/SWE/pdl_std.html

[3] Szasz Janos, The algorithmicx package
http://tug.ctan.org/macros/latex/contrib/algorithmicx/algorithmicx.pdf

[4] Montgomery, Peter L. (1987). "Speeding the Pollard and Elliptic Curve Methods of Factorization" (PDF)
https://www.ams.org/journals/mcom/1987-48-177/S0025-5718-1987-0866113-7/S0025-5718-1987-0866113-7.pdf

PROBLEM 3 - F5

SOEN 6011 - Summer 2021 **Sijie Min**
Software Engineering Processes **40152234**
Repository address : https://github.com/Dakatsu/SOEN6011Calculator

The difficulty in implementing function 5 is to implement bx.
When X is an integer, there are two alternatives. One is direct accumulation. The advantage is that the code is easy to implement, but the disadvantage is that the running efficiency is not high (x times of multiplications are required); the second is to reduce the number of multiplications by decomposing the exponent. The advantage is that it runs more efficiently, but the disadvantage is that the code implementation is more complicated. For function 5, use the second method to achieve.
When X is a decimal, use Taylor series, because bx=exlnb, Taylor series can calculate ln(x) and ex, and can provide high-precision results. The disadvantage is that the function approximation requires a large amount of calculation. C) When x is an integer, if x is an even number, bx can be decomposed into (b2)(x/2); if x is an odd number, bx can be decomposed into b×(b 2)((x-1)/2). Continue to decompose until the exponent part is 1. When x is a decimal, it can be calculated according to the following formula
When x is a decimal, it can be calculated according to the following formula:
(1) abx=aexlnb
(2) ex = 1 + x/1! + x2/2! + x3/3! + ...
(3) ln(1 + x) = x x2/2 + x3/3 ......

Pseudo code Pattern

Algorithm 9 Power function whose exponent part is an integer

Require: $retVal = 1 And exp = y And tmp = x$

$functionExplicitFORM(power(x, y))$

if exp<0 **then**

  $exp = -exp$

end if

while $exp > 0$ **do**

  if exp%2==0 **then**

    $exp = exp/2$

    $tmp = tmp * tmp$

  else

    $y \leftarrow y \times X$

    $retVal = retVal * tmp$

  end if

end while

if y<0 **then**

  $retVal = 1/retVal$

  $return retVal$

Algorithm 10 Taylor Series

Require: $retVal = 1 And tmp = 1 And i = 1$

$functionExplicitFORM(ex(x))$

for $i ¸ i + 1$ **do**

$tmp = tmp * i$

$retVal = retVal + power(x, i)/tmp$

end for

return retVal

Require: $retVal = 0 And tmp = 1 And And i = 1 And x(0, 2]$

$functionExplicitFORM(lnBase(x))$

for $i ¸ i + 1$ **do**

$retVal = retVal + tmp * power(x, i)/i$

$tmp = -tmp$

end for

return retVal

Require: $retVal = 0 And LN2 = ln(2)$

$functionExplicitFORM(ln(x))$

while $x > 2$ **do**

$retVal = retVal + LN2$

$x = x/2$

end while

$retVal = retVal + lnBase(x)$

return retVal

=0