CS101 Introduction to computing

Problem Solving (Computing)

A. Sahu and P. Mitra

Dept of Comp. Sc. & Engg.

Indian Institute of Technology Guwahati

Outline

- Problem Solving : Process involves
  - Definition, Analysis, Solution Approaches, Correctness, Programming, Testing
- Loop invariant and loop termination
- Many Problem Solving Examples
  - 7 Problems **(Solution Method not given)**
  - 3 problems **(Solution Method given)**

Reference : R G Dromey, "***How to solve it by Computer***", Pearson Education India, 2009

Generic Problem Solving using Computer

- Step 1: Problem Definition in English
- **Step 2: Solution Approaches**
  - **Is any standard method available ? Proven to be correct.**
  - **If yes, use that to draw flow chart**
- **Step 3: Flow chart and Pseudo code**
- Step 4: Write C Code
- Step 5: Compile and Run
- Step 6: Test the code for error

Steps in Programming : Very Simplified Picture

- Problem Definition & Analysis
- **High Level Strategy for a solution**
- **Arriving at an algorithm**
- **Verification and analysis of the algorithm**
- Encoding the algorithm as a program

  (in a programming language like C)
- Testing the program

Each step **iterative** and the **whole process** also iterative

Problem Definition & Analysis

- Understanding the problem is : **Half the solution** ☺ ☺ ☺
  - **You appearing for some exam, and you are not able to understand the question**
    - **Can you find solution without understanding the question? ☺ ☺ ☺ == > NO**
- A precise solution requires a **precise definition**
- This step leads to clear definition of the problem
- The definition states **WHAT** the problem to be solved
- Rather than **HOW** the problem to be solved

Problem Definition & Analysis Cntd..

- Analysis done to get a complete and consistent specification
- Specification **precisely and unambiguously states**
  - Constraints on the inputs
  - Desired Properties of the outputs

High Level Strategy

- This is a **crucial and most difficult step**
- **Most creative part** of the whole process
- **No standard recipe** for arriving at a strategy
- **Compare** alternate techniques to arrive at the best

Analysis of Solution Approaches

- Correctness and Efficiency (C & E)
  - Algorithm/Approaches are analyzed for C & E
  - C & E are precise and detailed enough
- **Correctness analysis**
  - To ensure the **algorithm solves** the given problem
  - Involves a mathematical proof that algorithm **satisfies the specification**; termination proofs
- **Efficiency analysis** : To determine
  - **amount of time** or number of operations
  - **amount of memory** required for executing the algorithm

Algorithm

- The algorithm is part of the blueprint or plan for the computer program, an algorithm is:

  "An effective procedure for solving a class of problems in a finite number of steps."

- Every algorithm should have the following 5 characteristic features:
  - **Definiteness:** Each step must be define precisely
  - **Effectiveness :** its **operations** must be **basic enough** to be able to be **done exactly** and in **finite length of time**
  - **Termination:** must terminate after a finite number of steps
  - **Input** and **Output**

Programming (coding Approach)

- Writing programs in a programming language (in C) is the **last step**
- This step is called **implementation** or **coding**
  - No doubt it is important and one need to pay attention and care
  - But it is somewhat **straight forward**

Testing the Program

- Program is compiled to generate
  - Machine code that can run on a specific machine
- Errors could be introduced
  - In the programming process
  - Or by the compiler (suppose to be good ☺ ☺, but do many optimization)
- Hence it is essential that the generated code
  - Is run with specific set of inputs to see whether it produces the right outputs

Testing the Program

- Syntax/Grammar errors eliminated in the step
- Some logical errors may also be caught in this step
- Gives an idea about time and space requirements for executing the program

Problem Solving Strategies

- Arriving at a strategy and an algorithm is the **most crucial and difficult step**
- Crucial because behavior of the final code is dependent on this
- **Difficult because it is a creative step**
- Though many standard techniques are available **no general recipe to ensure success**

Problem Solving Strategies

- **New problems** may require **newer strategies**
- Problem solving skills can be developed **only with experience**
- Main emphasis of the course
  - To expose you to **various problem solving strategies by way of examples**
- The programming languages is for concreteness and execution of your ideas

Problem Solving Strategies

- Given a Problem P
- You may come up many Approaches/ strategies : App1, App2, App3, App4, Appm
- If we are not able prove the correctness by loop termination and loop invariant of some approaches
  - We cannot call that Approaches as Algorithm
- Suppose App2 and App3: We are not able prove the correctness for them , then App2 and App3 are not algorithms by definition
  - Algorithms for P: App1, ~~App2~~, ~~App3~~, App4, Appm

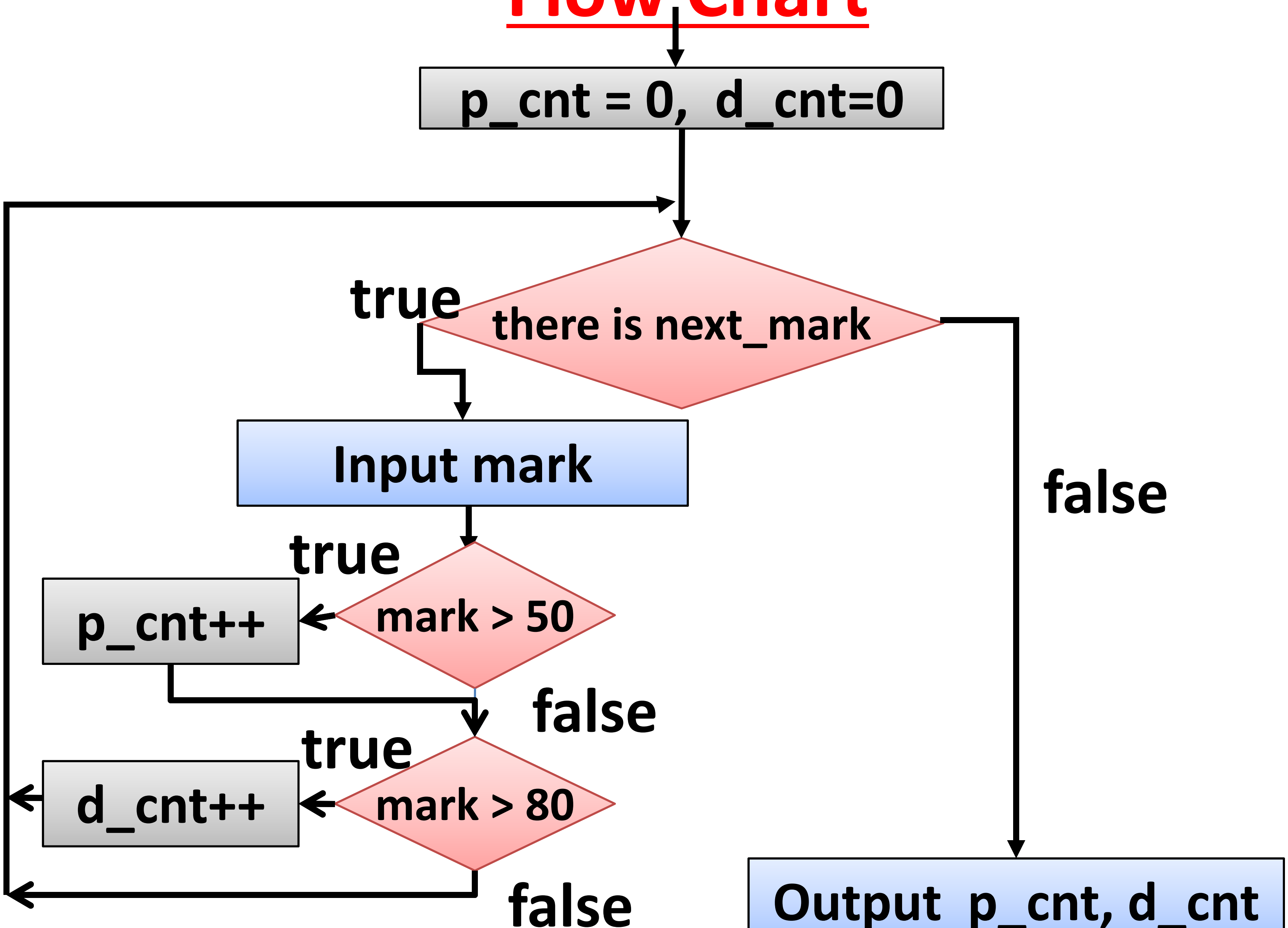
Illustrative Examples : I

- Problem: Given a set of students examination marks, (range 0 to 100), count the number of students that passed the examination and those passed with distinction
  - Pass mark: >=50
  - Distinction mark: >=80
- Study the problem and analyze
- Is the problem definition clear?

The Strategy

1. Keep **two counters** one for pass and the other for distinction
2. Read the marks **one by one**
3. **Compare each mark** with **50** and **80** and increase the appropriate counters
4. **Print** the final results

Flow Chart



p_cnt = 0, d_cnt=0
true
there is next_mark
false
Input mark
true
mark > 50
p_cnt++
false
true
mark > 80
d_cnt++
false
Output p_cnt, d_cnt

Pseudo Code

Input: List of marks

Output: pass_count, distinction_count

1. Initialize p_count, d_count to zero
2. **Do while** (there is next_mark)
   1. **If** next_mark => 50 **then** increment p_count
   2. **If** next_mark => 80 **then** increment d_count
3. Print p_count, d_count

Observations

- The algorithm is a sequence of precise instructions
  - Involves variables for storing input, intermediate and output data
  - Uses high level operations and instructions
  - Data types closer to the problem domain
- What does the algorithm do for marks that **do not lie between 0 and 100** ?
- Rewrite the algorithm
  - **Including above boundary case**

Correctness of the Solution

- **Is the solution correct?**
- **Show** that
  - if an input satisfies the **input constraints**
  - then output produced satisfies **required properties**
- **Input Constraints**
  - List of integers lying between 0 and 100
- **Required Property**
  - p_count contains the no. of marks >= 50
  - d_count contains the no. of marks >= 80
- **Termination** is an implicit requirement

How to establish correctness

- **Establish** that
  - if **input constraint is satisfied** then
  - the program will **terminate** producing the output that satisfies the **desired properties**
- How to establish?
  - Testing?
  - How many inputs will convince you?
  - 5, 10, 100 – **in general infinite**

Testing establish presence of bugs never their absence

Testing a simple program

```
float a, b;
scan(“%f %f”, &a,&b);
printf(“Result=%f”, a+b);
```

- How many test you require to conclude this code is working correctly.
- A is 32bit, b is 32 bit, number different options : $2^{32} * 2^{32} = 2^{64}$
- Suppose in one second you can do 2^{14} test , Still you will take 2^{50} s=2^{36}h=**7.8x10^6** years

Testing establish presence of bugs never their absence

Mathematical Argument

- **Prove the correctness** using **mathematical arguments**
- Proof of Correctness involves two-Step argument
  - **Loop Invariants**
  - **Loop Termination**

Loop invariants

- A **condition** (**logical expression**) involving program variables
  - It holds **initially**
  - If it holds **before start** of iteration, it holds at **the end**;
  - The condition remains invariant under iteration

Loop invariants

- **Loop invariant for our example**
  - **p_count** hold the total number of pass in the marks read so far
  - **d_count** hold the total number of distinction in the marks read so far

Loop invariants holds at every iteration if it holds initially

- In particular, it holds at the end
- Input constraints imply loop invariant initially
- Loop invariant at the end, implies output condition

Loop Termination

- **Non termination** is an important **source of incorrectness**.
- **Correctness proof** includes **termination proof**
- Bound on iteration
  - An integer valued expression called **bound function** that **reduces in each iteration**
  - When the bound function reaches 0, loop terminates
- For our example, the bound function is:

  length of the input list yet to be processed

Efficiency Analysis

- How many number of **operations**?
  - In each iteration of the loop, constant number of comparisons
- Can we improve this?
  - If the number is less than 50, there is no need for comparing it with 80.
- Rewrite the algorithm

C Program

```
int main(){
  int p_cnt=0,d_cnt=0, mark;
  int there_is_next_mark=0;
  do { printf("Is there next mark..[0/1]\n");
    scanf("%d", &there_is_next_mark);
    if(there_is_next_mark==0)  break;
    printf("Enter mark..\n");
    scanf("%d", &mark);
    if(mark<0 || mark >100) return 0;//exit(0)
    if(mark>50){ p_cnt++;
     if (mark>80) d_cnt++: }
  }while(1);
  printf("pass_cnt=%d,dist_cnt=%d", p_cnt,
  d_cnt);
}
```

Problem Solving Example

- Set A **(Solution Method not given)**
  1. Nth Power of X
  2. Square root of a number
  3. Factorial of N
  4. Reverse a number
  5. Finding value of unknown by question answers
  6. Value of Nth Fibonacci Number
  7. GCD to two numbers

Problem Solving Example

- Set B **(Solution Method given)**
    1. Finding values Sin(x) using series sum
    2. Value of PI
    3. Finding root of a function Bisection Methods

Problem 1

The n^{th} power of X

The n^{th} power of X

- **Problem:** Given some integer x. write a program that computes the nth power x, where n is positive integer considerably greater than 1.
- Evaluating expression $p=x^n$

```
   Prod=1;
   for  (i=1; i<=n; i++){
       Prod= Prod * x;
   }
```

- Naïve or straight-forward approach

  How many multiplication: n

  Require n steps

Assumption : all basic operations on integers take constant time

The n^{th} power of X

- **Is there any better approach?**
- From basic algebra
  - if n is even == > $X^n = X^{n/2}.X^{n/2}$
  - If n is odd and n=2m+1 ==> $X^n= X^{2m+1}= x^m . x^m . x$
- From this above fact, can we calculate X^n in fewer steps
- Approach
  - Binary representation of n,
  - X^{23} Example $23=(10111)2=1x2^4+0x2^3+1x2^2+1x2^1+1x2^0$ $= 16+0+4+2+1$
  - Start from right to left
    - $1x2^4+0x2^3+1x2^2+1x2^1+1x2^0$

$\longleftarrow$

Approach/Algorithm

1. Initialize the power sequence and product variable *(let initial value of n is n0=n)*

Product=1; ProdSequence=x;

2. Do while n > 0 repeat

   2.1 if the next most binary digit of n is one then **Product = Product * ProdSecuence;**

   2.2 n = n /2;

   2.3 ProdSecuence *= ProdSecuence;

//**Invariant Product*ProdSecuence^{n}=x^n0 , n>=0**

Assumption : all basic operations on integers take constant time

Approach

- Binary representation of n,
- X^{23} Example
  $23=(10111)2=1x2^4+0x2^3+1x2^2+1x2^1+1x2^0$ =
  16+0+4+2+1
- Start from right to left

$$1x2^4+0x2^3+1x2^2+1x2^1+1x2^0 \xleftarrow{\hspace{10em}}$$

- Approach
  - Successive generation of x, x^2, x^4, x^8, x^{16}, …
  - Inclusion of the current power member into accumulated product when the corresponding binary digit is 1

Approach

Odd number or Right Most Bit

Before Loop

1	0	1	1	1	
$P=X^7.X^{16}=X^{23}$	$P=X^7$	$P=X^3.X^4$ $=X^7$	$P=X.X^2$ $=X^3$	$P=P.PS$ $=X$	**$P=1$**
X^{32}	X^{16}	X^8	X^4	X^2	**$PS=X$**
$N=0$	$N=1$	$N=2$	$N=5$	$N=11$	**$N=23$**
$X^{23}.(X^{32})^0$ $=X^{23}$	$X^7.(X^{16})^1$ $=X^{23}$	$X^7.(X^8)^2$ $=X^{23}$	$X^3.(X^4)^5$ $=X^{23}$	**$X.(X^2)^{11}$** $=X^{23}$	**$P*PS^n$ $=1.X^{23}$**

Loop Invariant

C –Code for X^n

```
int n, x, Prod, ProdSeq;
// Put code for Input n, x
Prod=1; ProdSeq=x;
while(n > 0)    {
  if ((n%2)==1){
     Prod=Prod*ProdSeq;
     }
  n=n/2;
  ProdSeq = ProdSeq* ProdSeq;
}
//Put code to Display Prod as Xn
```


Assumption : all basic operations on integers take constant time

Problem 2

The square root problem : sqrt(X)

The square root problem

- **Problem:** Write a program that computes the square root of a given number.
- Is the problem definition **clear?**
  - If 25 is the input, then 5 is the output
  - If 81 is the input, then 9 is the output
  - If 42 is the input, then ?
- For non perfect squares, the square root is a **real number**
- So the output should be **close to the real square root**
- How close? to a given accuracy

A more precise specification

- **Problem:** Write a program that given a number ***m*** outputs a real value ***r*** such that
  - ***r*r*** differs from ***m*** by a given accuracy value ***e***
- More precisely, the program outputs ***r*** such that

$$|r*r - m| < e$$

Solution Strategy

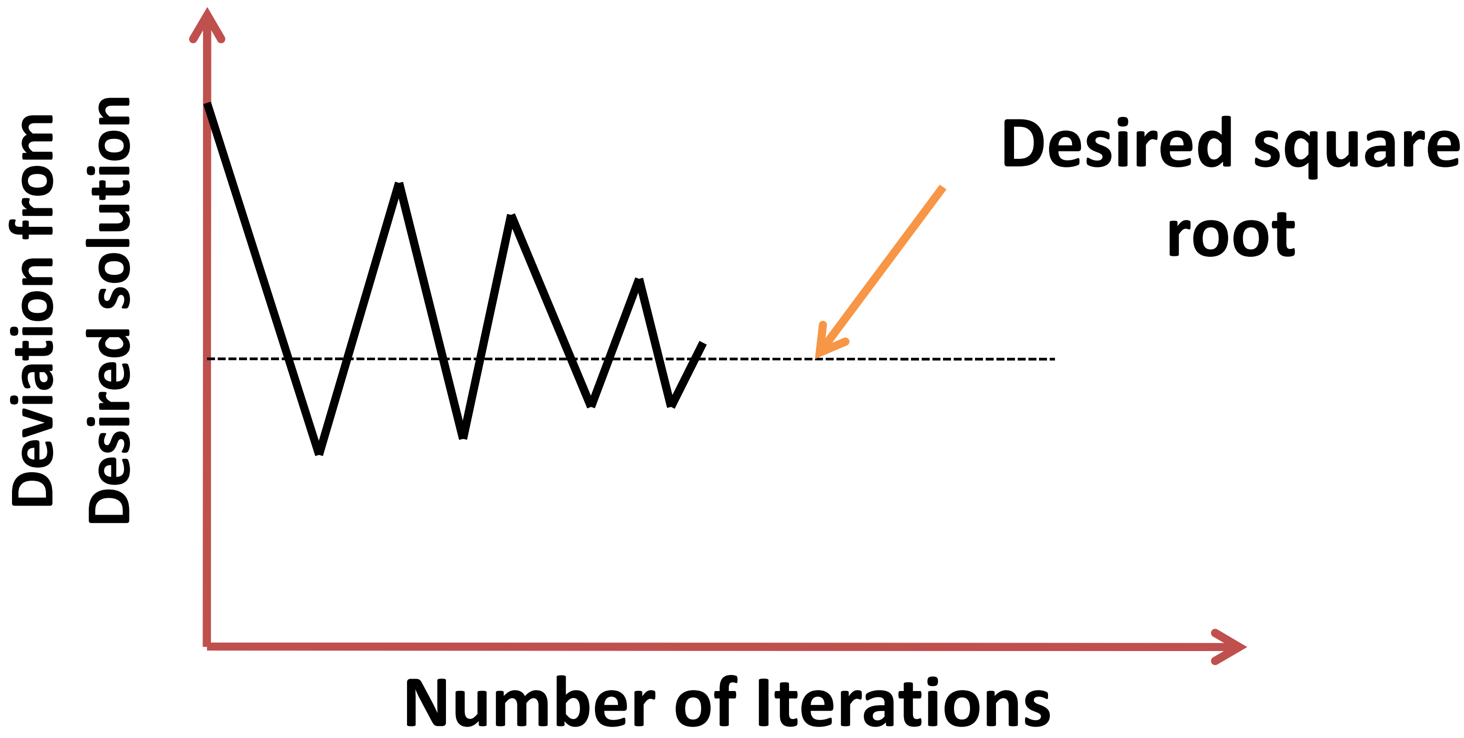
Guess and Correct Strategy:

1. Choose an initial guess ***r*** less than ***m***
2. If ***r*r > m*** then keep decreasing ***r*** by **1** until ***r*r*** is less than or equal to ***m***.
3. If ***r*r < m*** then keep increasing **r** by **0.1**, … until ***r*r*** exceeds or equals ***m***
4. If ***r*r > m*** then decrease ***r*** by ***0.01*** until ***r****r exceeds or equals ***m***.

…

- Terminate the computation when ***r*r*** equals ***m*** or differs from ***m*** by a given small number.

Idea



- Number of iteration depends upon the initial guess
- If m is 10,00,000 and the initial guess is 300 then over 700 steps are needed
- Can we have a better strategy?

Towards a better strategy

- The basic idea of the strategy is to obtain a **series of guesses** that
  - **falls on either side** of the actual value
  - **narrows down** closer and closer
- To make the guess fall on either side
  - **increase/decrease** the guess systematically
- To narrow the guess
  - the amount of increase/decrease **is reduced**
- Improving the strategy
  - **faster ways** of obtaining new guess from the old one

One Strategy

- Given a guess ***a*** for square root of m
  - ***m/a*** falls on the opposite side
  - *(a + m/a)/2*, can be the next guess
  - **Why this guess? Make next guess closer to sqrt(m) based on current guess.**
- This gives rise to the following solution
  - start with an arbitrary guess, r_0
  - generate new guesses r_1, r_2, etc by using the averaging formula.
- When to terminate?
  - when the successive guesses **differ by a given small number**

The Approach

Input float m, e, **assume:** m>0, 0< e > 1
Output float r_1, r_2

Loop Invariant :

|(r2*r2-m) | <= |(r1*r1-m)|, |r1- r2|> e

1. $r_1 = m/2$, $r_2 = r_1$
2. **Do**
    2.1 $r_1 = r_2$
    2.2 $r2 = (r_1+m/r_1)/2$

while ($|r_1 - r_2| > e$)

C Code : Square root of m

```
float m, e, r1, r2;
// Put code for Input m, e
r1=m/2;  r2=r1;
do    {
    r1=r2;
    r2=(r1+m/r1)/2;
} while(abs(r1-r2) > e)
//Put code to Display root as r2
```

Analysis of the Approach

- Is it **correct**? Find the **loop invariant** and **bound function**
- Can the algorithm be **improved?**
- More general techniques available
  - **Numerical analysis**
- NA: Newton Raphson's for square root

  $F(x) = x^2-m=0$

  $x_{k+1}=x_k-F(x_k)/F'(x_k) =x_k-(x_k^2-m)/2x_k$

  $\mathbf{x_{k+1}=(x_k+m)/2}$