1. Introduction

Algorithm: a finite set of instructions that specify a sequence of operations to be carried out in order to solve a specific problem or class of problems. [Zwass]

Properties of algorithms:

- **Finiteness:** Algorithm must complete after a finite number of instructions have been executed.
- **Absence of Ambiguity:** Each step must be clearly defined, having only one interpretation.
- **Definition of Sequence:** Each step must have a unique defined preceding and succeeding step. The first step (start step) and last step (halt step) must be clearly noted.
- **Feasibility:** All instructions must be able to be performed. Illegal operations (division by 0) are not allowed.
- **Input:** 0 or more data values.
- **Output:** 1 or more results.

3-Iteration Newton-Raphson Algorithm

1. Input: a real number X.
2. If (X < 0) Then
   2a. Output: X " cannot be negative."
   2b. STOP.
   Endif
3. Set a variable, S, to 1.0.
4. Set a counter variable, I, to 3.
5. While (I > 0) Do
   5a. Calculate the value \((S + X / S) / 2.0\).
   5b. Set S to the value calculated in step 5a.
   5c. Subtract 1 from I.
   Endwhile
6. Output: "The square root of " X " is about " S
7. STOP.

Does this possess the properties of an algorithm listed on the previous slide?
The Programming Process

Development Process

Problem-Solving Phase

1. Analysis & Specification
2. Design of a General Solution
3. Verification.

Implementation Phase

1. Production of a Specific Solution
2. Test

Maintenance Phase

1. Use (Install, Execute)
2. Maintain (Modify)

These are not really just a simple sequence of actions. Discovered errors will force a re-examination of the algorithm, or perhaps the underlying analysis or even the problem specification.

Maintenance often costs more than development and marketing together. Especially if the design is ill-considered or the testing is inadequate.

Pólya's Four-Step Process

1. Understand the problem
   a. Know the boundaries of the problem
   b. Know the constraints on the solution
   c. Know what actions are allowed

2. Devise a plan
   a. Organize thoughts to develop a detailed algorithm
   b. Use tools such as: outlining, flowcharting, and pseudocode.

3. Implement the plan
   a. Carry out the steps in the algorithm
   b. Translate the problem into a language understandable by the device to be used.

4. Test the plan
   a. Did the solution yield appropriate results?
   b. Can the solution be improved?

There is no recipe for solving general problems. However these ideas are often useful:

George Pólya (1887-1985)
1. Introduction

**Understand the Problem**

- What are the data?
- What is the required result?
- What is the starting point (initial conditions)?
- Is it at all possible to get the result from the data?

Useful approach: solve the problem by hand.

- What steps did you take? Can you write them down?
- Is the problem divided into major parts? Can they be identified?
- Have any problem assumptions been made? What are they?
- Is the solution general?

**Devise a Plan**

- Have you seen a similar problem before?

- Do you know of a related problem with a useful solution?
  - Can you use part of the related problem?
  - Can the solution of the related problem be modified and used?

- Look at the data... repetitions hint at loops in the solution.
  - Identify the patterns in the data.

- If you can't solve the problem, can you solve part of the problem?
Implement the Plan

Check each step of the plan.

Have you considered all the special cases?

Can you arrive at a reasonable result given...
  reasonable data?
  unreasonable data?

Make certain your solution works for "boundary conditions".
  Empty data?
  Data set too large?
  Data sets that results in illegal operations (division by zero)?

Test the Plan

Did you compute any intermediate results that were not used later?
  Can they be eliminated?

Can the result be derived differently?

Can the solution be made simpler or more general?

Can the solution method be used for other problems?
Machine Language: (very low-level language)
- consists of strings of 0's and 1's (binary digits or bits); often expressed in base 16 (hexadecimal) for clarity
- different for each machine (machine dependent)
- all programs must be written in machine language (code) OR be translated into machine code before being executed

Assembly Language: (low or intermediate level)
- short abbreviations (mnemonics) are used for instructions; for the assembly code above, one might have:

```
mov edx, DWORD PTR _SumOfScores$[ebp]
add edx, DWORD PTR _nextScore$[ebp]
mov DWORD PTR _SumOfScores$[ebp], edx
```

High Level Languages:
- Logical and relatively English-like
- machine independent
- must be translated before execution (compiled or interpreted)

Some common high-level languages:

- COBOL: Common Business-Oriented Language
- Fortran: Formula Translation
- Pascal: designed for programming instruction
- Ada: intended for general DOD contractor use
- C: intended for system software development
- C++: object-oriented extension of C language
- Java: pure object-oriented design, portable but interpreted
1. Introduction

**source code**  
high-level language instructions (C++) written by the programmer

**object code**  
low-level machine instructions readable by the hardware

**Compilation**

**Linking**

**Execution**

**Building**

**Compilation**

**Linking**

**Execution**

**source code**

**compiler**

**object code**

**linker**

**executable**

**Input Data**

**executable**

**results**

**loading**