# Definition

***algorithm***

an effective method expressed as a finite list of well-defined instructions for calculating a function

***effective method (or procedure)***

a procedure that reduces the solution of some class of problems to a series of rote steps which, if followed to the letter, and as far as may be necessary, is bound to:

- always give some answer rather than ever give no answer;
- always give the right answer and never give a wrong answer;
- always be completed in a finite number of steps, rather than in an infinite number;
- work for all instances of problems of the class.

# Properties



An algorithm must possess the following properties:

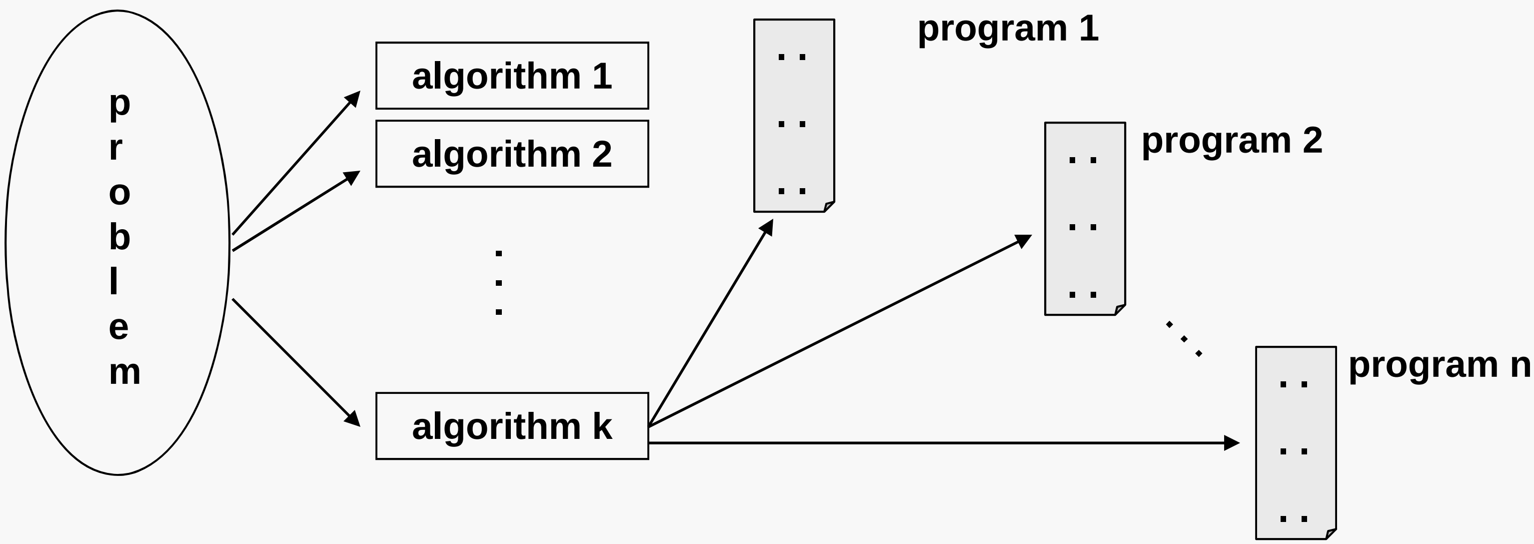
| | |
|---|---|
| *finiteness*: | Algorithm must complete after a finite number of instructions have been executed. |
| *absence of ambiguity*: | Each step must be clearly defined, having only one interpretation. |
| *definition of sequence*: | Each step must have a unique defined preceding & succeeding step. The first step (start step) & last step (halt step) must be clearly noted. |
| *feasibility*: | It must be possible to perform each instruction. |

# Problems vs Algorithms vs Programs



For each problem or class of problems, there may be many different algorithms.

For each algorithm, there may be many different implementations (programs).



# Expressing Algorithms

An algorithm may be expressed in a number of ways:

| | |
|---|---|
| *natural language*: | usually verbose and ambiguous |
| *flow charts*: | avoid most (if not all) issues of ambiguity; difficult to modify w/o specialized tools; largely standardized |
| *pseudo-code*: | also avoids most issues of ambiguity; vaguely resembles common elements of programming languages; no particular agreement on syntax |
| *programming language*: | tend to require expressing low-level details that are not necessary for a high-level understanding |

# Common Elements of Algorithms

*acquire data* (input)

some means of reading values from an external source; most algorithms require data values to define the specific problem (e.g., coefficients of a polynomial)

*computation*

some means of performing arithmetic computations, comparisons, testing logical conditions, and so forth...

*selection*

some means of choosing among two or more possible courses of action, based upon initial data, user input and/or computed results

*iteration*

some means of repeatedly executing a collection of instructions, for a fixed number of times or until some logical condition holds

*report results* (output)

some means of reporting computed results to the user, or requesting additional data from the user

# pseudo-Language

See the posted notes on the level 1 pseudo-language.

# Simple Example: Area of a Trapezoid

```
# Computes the area of a trapezoid.
# Input values must be non-negative real numbers.
#
   number Height, upperBase, lowerBase
   get Height
   get upperBase
   get lowerBase

   number averageWidth, areaOfTrapezoid
   averageWidth := ( upperBase + lowerBase ) / 2
   areaOfTrapezoid := averageWidth * Height

   display areaOfTrapezoid
   halt
```

# Simple Example: N Factorial

```
# Computes N! = 1 * 2 * . . . * N-1 * N, for N >= 1
# Input value must be a non-negative integer.
#
   number N, nFactorial
   get N

   nFactorial := 1
   while N > 1
      nFactorial := nFactorial * N
      N := N - 1
   endwhile

   display nFactorial
   halt
```

# Example: Finding Longest Run

```
# Given a list of values, finds the length of the longest sequence
# of values that are in strictly increasing order.
# Input List must contain Sz elements.
#
    number Sz                    # number of elements in List
    list number List             # list of values to be processed
    number currentPosition       # specifies list element currently
                                 #   being examined
    number maxRunLength          # stores length of longest run seen
                                 #   so far
    number thisRunLength         # stores length of current run

    get Sz                       # initialize the list size and the
    get List                     # list itself
    if Sz <= 0                   # if list is empty, no runs...
       display 0
       halt
    endif

    currentPosition := 1         # start with first element in list
    maxRunLength := 1            # it forms a run of length 1
    thisRunLength := 1

    # continues on next slide...
```

# Example: Finding Longest Run

```
# ...continued from previous slide

    while  currentPosition < Sz

        if ( List[currentPosition] < List[currentPosition + 1] )
           thisRunLength := thisRunLength + 1
        else
           if ( thisRunLength > maxRunLength )
              maxRunLength := thisRunLength
           endif
          thisRunLength := 1
        endif

        currentPosition := currentPosition + 1

    endwhile

    display maxRunLength
    halt
```

# Testing Correctness

How do we know whether an algorithm is actually correct?

First, the logical analysis of the problem we performed in order to design the algorithm should give us confidence that we have identified a valid procedure for finding a solution.

Second, we can test the algorithm by choosing different sets of input values, carrying out the algorithm, and checking to see if the resulting solution does, in fact, work.

BUT… no matter how much testing we do, unless there are only a finite number of possible input values for the algorithm to consider, testing can never prove that the algorithm produces correct results in all cases.

# Proving Correctness



We can attempt to construct a formal, mathematical proof that, if the algorithm is given valid input values then the results obtained from the algorithm must be a solution to the problem.

This is complex and an area of ongoing research.

# Measuring Performance

How can we talk precisely about the "cost" of running an algorithm?

What does "cost" mean? Time? Space? Both? Something else?

And, if we settle on one thing to measure, how do we actually obtain a measurement that makes sense?

This is primarily a topic for a course in algorithms, like CS 3114 or CS 4104.