Algorithms are the threads that tie together most of the subfields of computer science.

Something magically beautiful happens when a sequence of commands and decisions is able to marshal a collection of data into organized patterns or to discover hidden structure.

Donald Knuth
**Definition**

**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.

**algorithm**

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

An algorithm must possess the following properties:

**finiteness:** The algorithm must always terminate after a finite number of steps.

**definiteness:** Each step must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.

**input:** An algorithm has zero or more inputs, taken from a specified set of objects.

**output:** An algorithm has one or more outputs, which have a specified relation to the inputs.

**effectiveness:** All operations to be performed must be sufficiently basic that they can be done exactly and in finite length.
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, including:

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>natural language:</td>
<td>usually verbose and ambiguous</td>
</tr>
<tr>
<td>flow charts:</td>
<td>avoid most (if not all) issues of ambiguity; difficult to modify w/o specialized tools; largely standardized</td>
</tr>
<tr>
<td>pseudo-code:</td>
<td>also avoids most issues of ambiguity; vaguely resembles common elements of programming languages; no particular agreement on syntax</td>
</tr>
<tr>
<td>programming language:</td>
<td>tend to require expressing low-level details that are not necessary for a high-level understanding</td>
</tr>
</tbody>
</table>
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
See the posted notes on pseudo-language notation.
Simple Example: Area of a Trapezoid

```plaintext
algorithm AreaOfTrapezoid takes number Height, 
                 number lowerBase, 
                 number upperBase

# Computes the area of a trapezoid.
# Pre:  input values must be non-negative real numbers.
#

    number averageWidth, areaOfTrapezoid

averageWidth := ( upperBase + lowerBase ) / 2

areaOfTrapezoid := averageWidth * Height

display areaOfTrapezoid
halt
```
Simple Example: N Factorial

algorithm Factorial takes number N

# Computes N! = 1 * 2 * . . . * N-1 * N, for N >= 1
# Pre: input value must be a non-negative integer.

number nFactorial

nFactorial := 1

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

display nFactorial
halt
Example: Finding Longest Run

```plaintext
algorithm LongestRun takes list number List, number Sz

# Given a list of values, finds the length of the longest sequence
# of values that are in strictly increasing order.
# Pre: input List must contain Sz elements.
#
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

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

```

QTP: is this algorithm correct?
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.
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.

We should expect that such a proof be provided for every algorithm.

In the absence of such a proof, we should view the purported algorithm as nothing more than a heuristic procedure, which may or may not yield the expected results.
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.