Algorithms

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

Example

- Recipe for chicken and macaroni casserole

This is a tasty casserole, and the Gouda cheese can be replaced with a sharp Cheddar. I have used packaged roasted chicken cubes or pre-cooked rotisserie chicken in this when pressed for time.
Ingredients:

8 ounces elbow macaroni or small shells, salt, 3 tablespoons butter, 3 tablespoons flour, 1 cup chicken broth,
1/2 cup heavy cream, 3 to 4 ounces smoked gouda cheese, shredded or cut in small pieces, pepper, to taste,
1 teaspoon fresh parsley, optional, 1 1/2 to 2 cups cubed cooked chicken, 1 1/2 cups frozen peas and carrots thawed, 1/2 cup soft bread crumbs, 1 to 2 teaspoons butter, melted

Preparation:
Cook macaroni in boiling salted water as package directs. Drain and set aside.

In a saucepan over medium low heat, melt butter; add flour. Cook, stirring, until flour mixture is well blended and bubbly. Gradually stir in chicken broth and cream. Stir in cheese until melted and smooth. Add pepper, to taste, along with parsley, if using. Cook, stirring, until thickened. Add the chicken and vegetables; cook for about 1 minute longer.

Combine the sauce with cooked and drained macaroni; pour into a 2-quart casserole. Toss bread crumbs with melted butter and sprinkle over the casserole. Bake at 350° for about 25 minutes, until bubbly and browned. Serves 4 to 6.

Example

Knitting pattern for a shawl

Eyelet Rib Wrap Knitting Pattern, Sally Trefftzs
Rasberry Wrap
Use any yarn, any needles. I used most of seven 50 gram skeins of Unger's Utopia Sport in bright raspberry pink and size 5 US (3.75 mm) needles for both the rectangle and lace.

The original inspiration came from Ruby Townsend of the San Diego Knitting Guild. I was at the meeting where she demonstrated what could be done with a wide border of lace all around a simple ribbing rectangle. It was joined at the corners of the rectangle to make a stretchy "wrap" with a wide collar and cuffs.

I used the single eyelet rib on page 46 of Barbara Walker's First Treasury for my rectangle and a baby blanket edging from years ago for the lace, changed to make it shorter here, longer there and fully reversible.

You need to measure your intended victim. Have her stand with hands on hips and measure from the tip of one elbow to the tip of the other. For my tiny mother this was 25". For me, 30". A swatch of your chosen ribbing is needed. Knit about a 4" wide and 2 to 3" long and take it off the needles.

Measure it relaxed, measure it stretched. Add these two numbers together and divide by two to give you the average. Multiply your average number of sts by the width in inches to tell you approximately how many sts to cast on. You want a fairly stretchy rectangle of your chosen width by about 18" tall.

For the lace edging, measure around the rectangle and knit the edging longer than that. You can use any pattern that will give you a width of 4-6 inches. I knitted a width of 5 ½" and a length of 90". It would have been nicer with about 95" so the corners could have been eased better. This lace has gathers knitted into it for extra fullness.

Reversible Lace Edging

CO 28 sts loosely and knit 1 row.

Row 1: k2, p16, (yo, k2tog) 4 times, yo, k2 (29 sts)
Row 2: k12, (yo, k2tog) 7 times, k1, turn leaving 2 sts, p17, (yo, k2tog) 4 times, yo, k2 (30 sts)
Row 3: k30
Row 4: k20, (yo, k2tog) 4 times, yo, k2 (31 sts)
Row 5: k11, p18, turn leaving 2 sts, sl 1, (yo, k2tog) 13 times, yo, k2 (32 sts)
Row 6: k11, p19, turn leaving 2 sts, k30 (32 sts)
Row 7: BO 4 sts loosely, k to end (28 sts)

Rep these 7 rows for desired length and sew or graft the ends together. Pin and sew edging to rectangle, easing corners of edging.

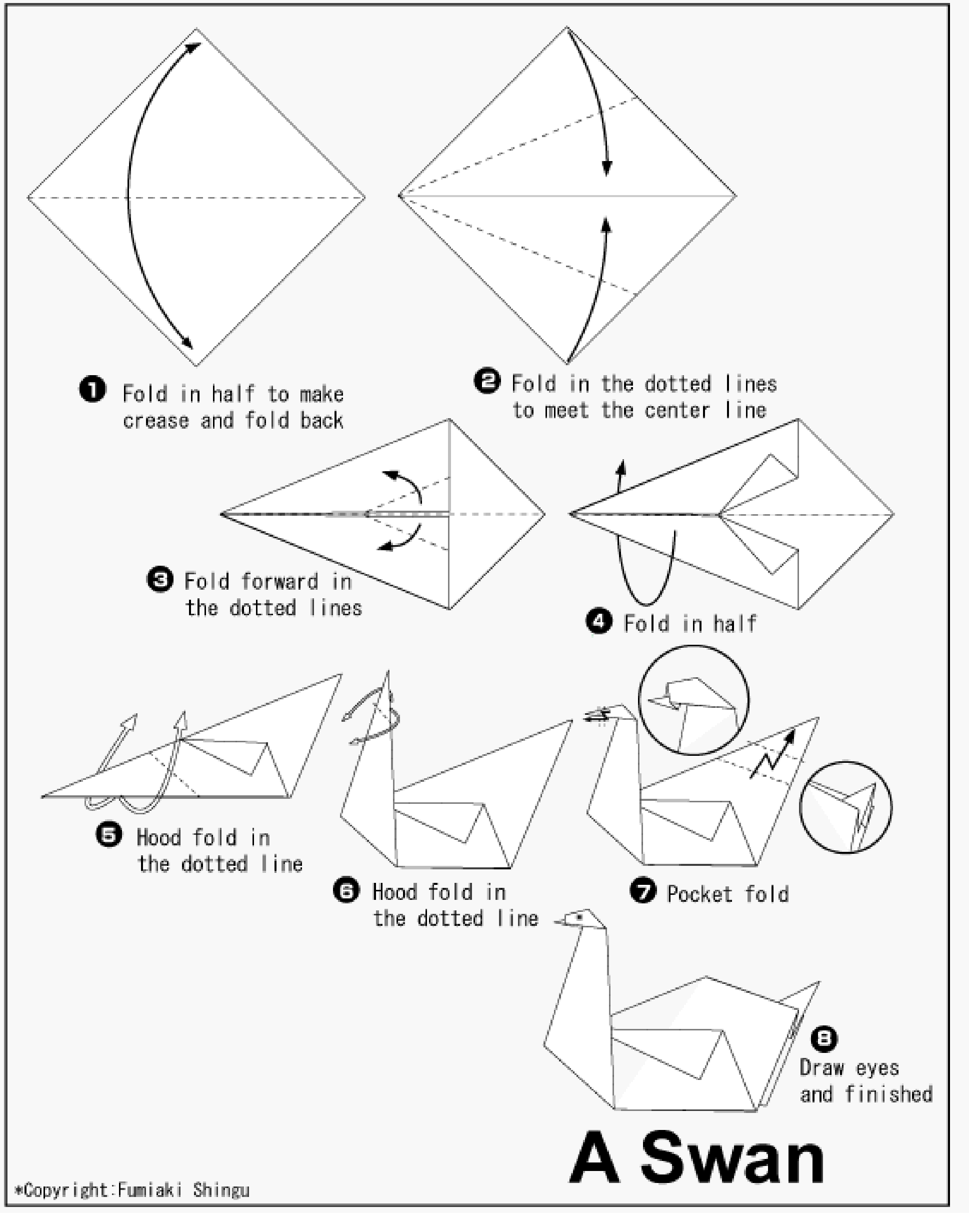
With long side of rectangle facing you, fold top corners to bottom ones and sew together just at the corners. Or make button loops out of crochet chain and sew on buttons. This option allows the wrap to become a lap-robe, too.

For novice lace makers, I found that I got the best results when knitting loosely on my chosen needle. Going up a needle size made the lace look coarse. If you don't bind off loosely the points tend to buckle and fold up. As an error check: for any row that includes the instruction (yo, k2tog) 4 times, yo, k2, there should be 10 sts remaining on the left needle when you reach that instruction.

See Abbreviations and the Glossary for help.
http://www.knittingonthenet.com/patterns/shawlraspberry.htm

Example

- Diagram for making a origami swan:



Properties of an Algorithm

An algorithm must possess the following properties:

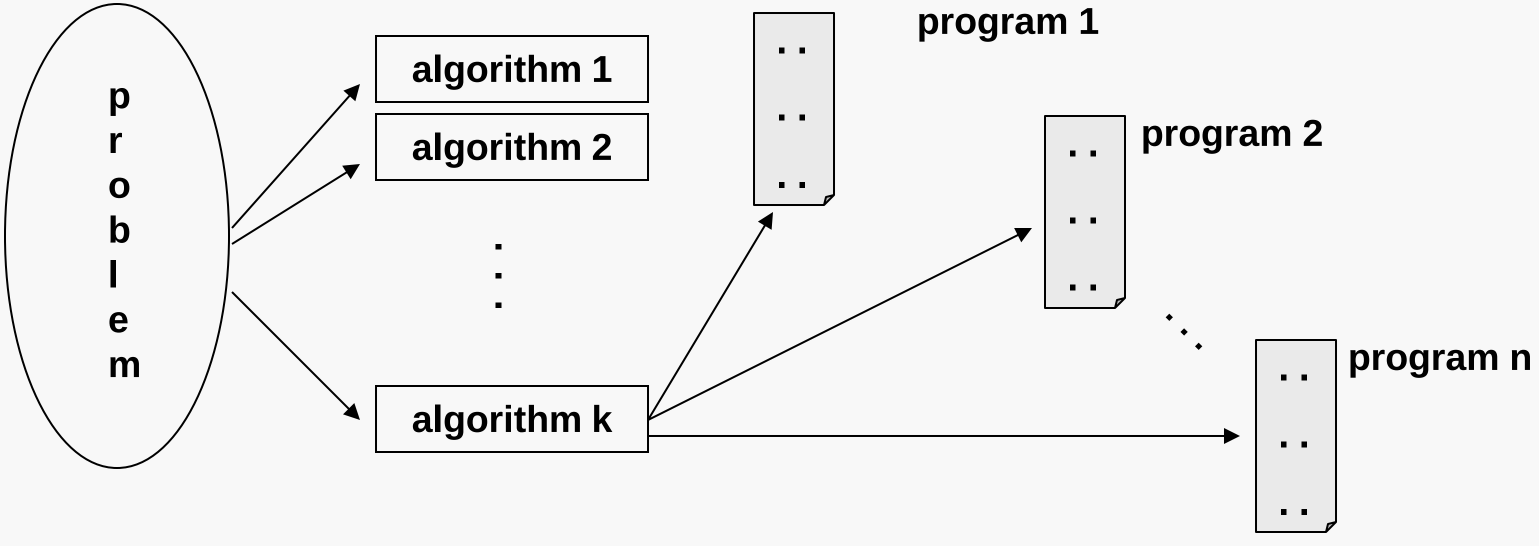
finiteness:	The algorithm must always terminate after a finite number of steps.
definiteness:	Each step must be precisely defined; the actions to be carried out must be rigorously and unambiguously specified for each case.
input:	An algorithm has zero or more inputs, taken from a specified set of objects.
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.

Knuth

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).



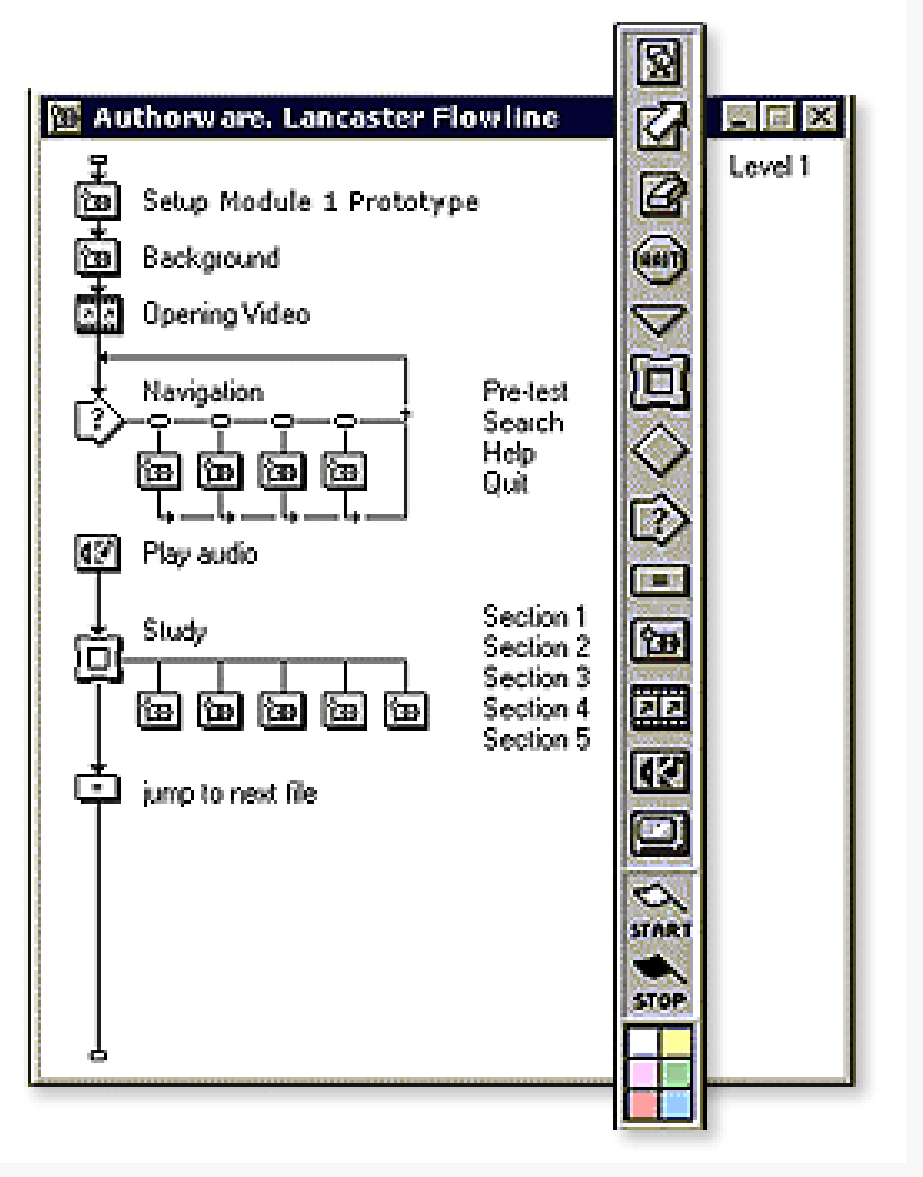
Expressing Algorithms

An algorithm may be expressed in a number of ways, including:

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

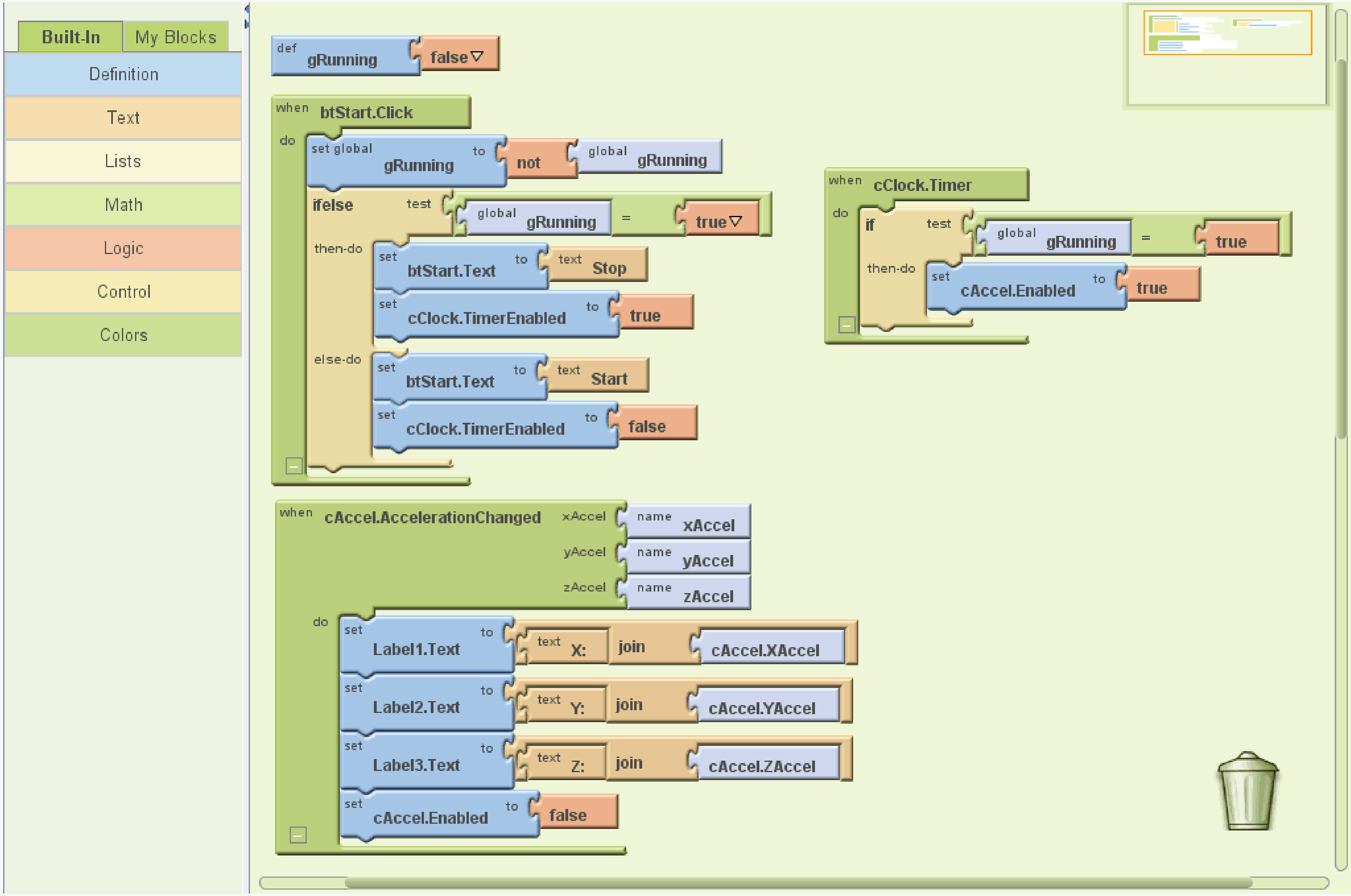
Example

- Authorware



Example

- App Inventor



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 pseudo-language notation.

Simple Example: Area of a Trapezoid

```
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

```
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.

Program testing can be used to show the presence of bugs, but never to show their absence!
- Edsger Dijkstra

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.

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.

Be careful about using the following code -- I've only proven that it works, I haven't tested it.
- Donald Knuth

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.

The inside of a computer is as dumb as hell but it goes like mad!
- Richard Feynman