An algorithm is a step-by-step description showing how to find the solution to a problem. Algorithms are generally written in a notation that can be understood by a human, whereas a computer program is written in a language that can be processed by the computer.

**Example**
Here is an algorithm for calculating the gross pay of a worker who gets “time-and-a-half” for hours worked over 40.

<table>
<thead>
<tr>
<th>Input</th>
<th>Algorithm</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>wages</td>
<td>Base pay is the wages times hours worked up to 40</td>
<td></td>
</tr>
<tr>
<td>hours</td>
<td>Overtime pay is wages times 1½ for each hour worked over 40</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Gross pay is base pay plus overtime pay</td>
<td></td>
</tr>
</tbody>
</table>

For a worker that makes $10 per hour and works 45 hours, you can follow the instructions of the algorithm to determine his or her gross pay:

- Base pay is $10 × 40 = $400
- Overtime pay is $10 × 5 × 1.5 = $75
- Gross pay is $400 + $75 = $475

$475
Nowadays, most computer programmers write their algorithms in *pseudo-code*, an informal notation that uses a combination of English, math formulas and computer language statements.

**Example**

Here’s the algorithm, written in pseudo-code, for finding the length of the diagonal of a rectangular parallelepiped when given its width, height and depth.

In the picture at right the width, height and depth are shown as $a$, $b$ and $c$, respectively; the diagonal is $d$.

**Algorithm to Calculate the Diagonal of a Rectangular Parallelepiped**

*Input* the width, height and depth

\[ \text{diagonal} = \sqrt{\text{width}^2 + \text{height}^2 + \text{depth}^2} \]

*Output* diagonal

For a room that is $20' \times 12' \times 30'$, you can follow the algorithm and determine its diagonal:

width is 20, height is 12, depth is 30

\[
\text{diagonal} = \sqrt{20^2 + 12^2 + 30^2} \\
= \sqrt{400 + 144 + 900} \\
= \sqrt{1,444} \\
= 38
\]

output the 38
**Pseudo-Code Conditional Statement**

Many problems require that you make a decision to determine which computations must be done.

**Example**

A company pays “time-and-a-half” for hours worked over 40. The calculation for a worker’s gross pay is different depending on whether or not he or she has worked over 40 hours or not.

Company pays $10 per hour

Worker #1 has 30 hours and so is paid
$10 \times 30 = $300

Worker #2 has 45 hours and so is paid
$10 \times 40 + $15 \times 5 = $475.

In English, you’d spell this out using some sort of conditional statement (if this then that).

**Example**

If a worker has 40 hours or less then his or her gross pay equals $10 times the number of hours.

If a worker has over 40 hours then his or her gross pay equals $400 plus $15 times the number of hours over 40.
Pseudo-code uses a very stylized version of this English construct that is referred to as a *conditional, selection or alternation*. The usual form is:

```plaintext
if condition then ... else ... end if
```

This indicates that one of two computations is to be done, depending on the condition, which is either true or false. If true, the computation after *then* (indicated by the …) is to be done; if false that after the *else* is to be done. *end if* is a pseudo-code period (.) that marks the end of the sentence.

**Example**

Here’s the gross pay algorithm written in pseudo-code.

```
Algorithm to Calculate Gross Pay Including “Time-and-a-Half”
input wages and hours
if hours ≤ 40 then
  base pay = wages × hours
  overtime pay = 0
else
  base pay = wages × 40
  overtime pay = wages × (hours - 40) × 1½
end if
gross pay = base pay + overtime pay
output gross pay
```

Let’s follow the steps of the algorithm for a worker with 30 hours at $10 per hour:

```
wages is 10, hours is 30
hours ≤ 40 is true, so
base pay = 10 × 30 = 300
overtime pay = 0
gross pay = 300 + 0 = 300
output 300
```
Now for a worker with 45 hours at $12 per hour:

- wages is 12, hours is 45
- hours ≤ 40 is false, so
- base pay = 12 × 40 = 480
- overtime pay = 12 × (45 - 40) × 1½ = 60 × 1½ = 90
- gross pay = 480 + 90 = 570
- output 570

The conditional statement can be extended to specify any number of choices.

**Example**
Here’s a pseudo-code algorithm that specifies how to determine a student’s grade on a 90-80-70-60 scale.

```plaintext
Algorithm to Determine a Student’s Grade on a 90-80-70-60 Scale
input student’s numeric score
if numeric score = 90 to 100 then
    letter grade = ‘A’
else if numeric score = 80 to 89 then
    letter grade = ‘B’
else if numeric score = 70 to 79 then
    letter grade = ‘C’
else if numeric score = 60 to 69 then
    letter grade = ‘D’
else
    letter grade = ‘F’
end if
output letter grade
```
Pseudo-Code Repetition Statements
It is also common in problem solving that a computation must be repeated over and over again.

Example
A decimal (base 10) number can be converted to binary (base 2) by repeatedly dividing it by 2 while saving the remainders of the division from right to left in the order generated (i.e. the first remainder is rightmost). For example, $75_{10} = 1001011_2$ calculated as follows:
English words that specify repetition include the verb \textit{repeat} and the adverb \textit{while}. In pseudo-code we write these in a stylized form called a \textit{repetition} or \textit{loop}. One form is the \textit{repeat-until}:

$$\text{repeat ... until condition is true}$$

This indicates that the computation after \textit{repeat} (indicated by the \ldots) is to be done repeatedly. The \textit{condition} can be either true or false. So long as it is false, the computation is repeated.

\section*{Example}
This pseudo-code indicates that \textit{n} is to be repeatedly divided by 10 until it becomes a fraction.

\begin{center}
\begin{verbatim}
input n
repeat
    divide n by 10
until n < 1
\end{verbatim}
\end{center}

Here’s the algorithm in action for an input of 1234; it stops repeating after 4 cycles:

\begin{center}
\begin{tabular}{l}
\textbf{n's value} \\
\textbf{loop cycle} \\
input n \rightarrow 1234 \\
repeat \\
    divide n by 10 \rightarrow 123.4 12.34 1.234 .1234 \\
until n < 1 \rightarrow false false false true quit
\end{tabular}
\end{center}
Another pseudo-code loop is the `while-do`:

```
while condition is true do ... end while
```

This specifies that the computation after `do` (indicated by the `...`) is to be repeated so long as the `condition` is `true`. `end while` is a pseudo-code period (.) that marks the end of the sentence.

**Example**
This pseudo-code rewrites the previous example using the `while-do`.

```
input n
while n ≥ 1 do
    divide n by 10
end while
```

Here’s the algorithm in action for an input of 1234.

<table>
<thead>
<tr>
<th><code>n</code>'s value</th>
<th>1234</th>
<th>12.34</th>
<th>1.234</th>
<th>.1234</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>condition</code></td>
<td>true</td>
<td>true</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td><code>n</code>'s value</td>
<td>123.4</td>
<td>12.34</td>
<td>1.234</td>
<td>.1234</td>
</tr>
</tbody>
</table>

```
input n
while n ≥ 1 do
    divide n by 10
end while
```

loop cycle
**Posttest versus Pretest**
The pseudo-code `repeat` has its condition at the bottom to indicate that it is to be checked after the first cycle of the loop, thus guaranteeing that the loop always cycles at least once. A looping construct with this property is called a *posttest loop*.

The pseudo-code `while`, however, has its condition at the top to indicate that it is to be checked before the first loop cycle. If the condition is already false before the repetition even begins then the loop will not cycle at all. A looping construct with this property is called a *pretest loop*.

**Example**
This example describes a situation where it is more appropriate to use a posttest rather than a pretest loop.

How would you find the number of digits in a given whole number? One solution is to remove each digit and count it.

Removing the digit can be accomplished numerically by dividing the number by 10 (which moves the decimal point one place to the left) and removing the resulting $\frac{1}{10}$ place:

After removing the last digit, the resulting value is zero:
Here’s the repetitive computation:

![Algorithm diagram](image)

Writing the pseudo-code algorithm with a repeat yields the following:

```
Algorithm for Counting the Number of Digits in n
input n
set digit count to 0
repeat
    divide n by 10 and drop the \(\frac{1}{10}\) place
count the digit
until n = 0
output the digit count
```

And here it is in action:

```
<table>
<thead>
<tr>
<th>loop cycle</th>
<th>n's value</th>
<th>digit count</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>3186</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>318</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>31</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>4</td>
</tr>
</tbody>
</table>
```

A repeat (i.e. posttest loop) is needed for this solution because if the input \(n\) is 0, the loop must cycle once, counting the digit and outputting a digit count of 1. A while (i.e. pretest loop) would incorrectly give a digit count of 0 for the input \(n = 0\) because it would not cycle at all.
**Example**

This example describes a situation where it is more appropriate to use a pretest rather than a posttest loop.

Consider the problem of adding a list of numbers in your head, say 12, 7, 9 and 10. Most people do it by remembering the accumulated total.

```
0 + 12 = 12
12 + 7 = 19
19 + 9 = 28
28 + 10 = 38
```

The algorithm that describes this computation ought to use a *while* (i.e. pretest loop) so that it is correct even if there are no numbers to add together.

```
Algorithm for Adding a List of Numbers

start accumulator at 0

while there's another number to add do
    input a number
    add the number to the accumulator

end while

output the accumulator
```

If there are no numbers to add together, a *repeat* would incorrectly try to input a nonexistent number because it wouldn’t discover the absence of numbers until after it cycles one time.

Why would you even want to employ the algorithm if there are no numbers to add together? You may not know whether there are numbers or not. For instance, consider a department store that has a computer program to calculate the day’s total sales of each employee. For employees that have sales then the program must correctly add them and for employees with no sales the program has no numbers to add. The program’s logic requires a pretest loop so that it works in both situations.
## Exercises

1. Follow the instructions of the Algorithm to Calculate the Diagonal of a Rectangular Parallelepiped (see page 2) to determine the length of the diagonal of a box that is 15” × 8’ × 18’. Do it again for a room of size 25’ × 10’ × 40’.

2. This algorithm tells how to convert a Celsius temperature to its equivalent on the Fahrenheit scale.

   **Algorithm to Convert Celsius to Fahrenheit**
   
   ```markdown
   input celsius
   fahrenheit = \( \frac{9}{5} \) celsius + 32
   output celsius ° C = fahrenheit ° F
   ```

   Follow its instructions to determine the Fahrenheit equivalents of 0 °C, 20 °C, 50 °C and 100 °C.

3. This algorithm tells how to convert a Fahrenheit temperature to its equivalent on the Celsius scale.

   **Algorithm to Convert Fahrenheit to Celsius**
   
   ```markdown
   input fahrenheit
   celsius = \( \frac{5}{9} \) (fahrenheit − 32)
   output celsius ° C = fahrenheit ° F
   ```

   Follow its instructions to determine the Celsius equivalents of 0 °F, 32 °F, 100 °F and 212 °F.

4. This algorithm is for converting a Fahrenheit temperature to its equivalent Kelvin measurement.

   **Algorithm to Convert Fahrenheit to Kelvin**
   
   ```markdown
   input fahrenheit
   kelvin = \( \frac{5}{9} \) (fahrenheit − 32) + 273.15
   output kelvin K = fahrenheit ° F
   ```

   Follow its instructions to determine the Kelvin equivalents of 0 °F, 32 °F, 100 °F and 212 °F.
5. This algorithm calculates the circumference of a circle of radius \( r \) using the formula \( \text{circumference} = 2\pi r \).

<table>
<thead>
<tr>
<th>Algorithm to Calculate a Circle’s Circumference</th>
</tr>
</thead>
<tbody>
<tr>
<td>input radius</td>
</tr>
<tr>
<td>circumference ( = 2\pi r )</td>
</tr>
<tr>
<td>output circumference</td>
</tr>
</tbody>
</table>

Follow its instructions to determine the circumference of a circle of radius 10. Do it again for a circle of radius 50.

6. This algorithm calculates the area of a circle of radius \( r \) using the formula \( \text{area} = \pi r^2 \).

<table>
<thead>
<tr>
<th>Algorithm to Calculate a Circle’s Area</th>
</tr>
</thead>
<tbody>
<tr>
<td>input radius</td>
</tr>
<tr>
<td>area ( = \pi \text{ radius}^2 )</td>
</tr>
<tr>
<td>output area</td>
</tr>
</tbody>
</table>

Follow its instructions to determine the area of a circle of radius 5. Do it again for a circle of radius 25.

7. Follow the instructions of the Algorithm to Calculate Gross Pay Including “Time-and-a-Half” (see page 4) to determine the gross pay of a worker working 40 hours at $12 per hour. Do it again for a worker working 41 hours at $12 per hour. Do it again for a worker working 50 hours at $18 per hour.
8. This algorithm converts temperatures between Fahrenheit and Celsius scales.

Algorithm to Convert Between Fahrenheit and Celsius

\[
\begin{align*}
\text{input the temperature and scale (F or C)} \\
\text{if the scale is F then} \\
\quad \text{celsius} &= \frac{9}{5} (\text{temperature} - 32) \\
\quad \text{output temperature } &\degree F = \text{celsius } \degree C \\
\text{else} \\
\quad \text{fahrenheit} &= \left( \frac{5}{9} \text{temperature} \right) + 32 \\
\quad \text{output temperature } &\degree C = \text{fahrenheit } \degree F \\
\text{end if}
\end{align*}
\]

Follow its instructions to determine the output for input of 32 F, 100 F, 212 F, 0 C, 50 C and 100 C.

9. Follow the instructions of the Algorithm to Determine a Student's Grade on a 90-80-70-60 Scale (see page 5) to determine the grade of a student whose numeric score is 90. Do it again for numeric scores of 79, 60 and 59.

10. Follow the instructions of the Algorithm for Counting the Number of Digits in n (see page 10) to determine the number of digits for input values of 98765, 65 and 0.

11. This algorithm converts a decimal number (base 10) to its equivalent in binary (base 2).

Algorithm for Converting Base 10 to Base 2

\[
\begin{align*}
\text{input the decimal number} \\
\text{initialize the binary string to be null} \\
\text{repeat} \\
\quad \text{divide the decimal number by 2} \\
\quad \text{decimal number} &= \text{quotient (for next loop cycle)} \\
\quad \text{append the remainder to the front of the binary string} \\
\text{until} \text{decimal number} = 0 \\
\text{output the binary string}
\end{align*}
\]

Follow its instructions to determine the output for input of 75, 100, 0 and 20.