Algorithmic Thinking
Algorithmic Thinking

- **Algorithmic thinking**: The ability to identify and analyze problems, and to develop and refine algorithms for the solution of those problems.

- **Algorithm**: Detailed step-by-step procedure for the performance of a task.

- Learning to program is about developing algorithmic thinking skills, *not* about learning a programming language.
Algorithms

- Ultimately, algorithms will be implemented by writing code in a particular programming language.
- Algorithm design is (mostly) language-independent.
  - A procedure that can be implemented in any language.
- Universal algorithm representations:
  - Flowcharts
    - Graphical representation
  - Pseudocode
    - Natural language
    - Not necessarily language-independent.
5

Flowcharts
Flow Charts

- **Flowcharts** are graphical representations of algorithms
- Interconnection of different types of blocks
  - Start/End
  - Process
  - Conditional
  - Input/Output
- Connection paths indicate flow from one step in the procedure to the next
- Well-constructed flowcharts are easily translated into code later
Flowchart Blocks

- **Start/End**
  - Always indicate the start and end of any flowchart

- **Process**
  - Indicates the performance of some action

- **Conditional**
  - Performs a check and makes a decision
  - Binary result: True/False, Yes/No, 1/0
  - Algorithm flow branches depending on result

- **Input/Output**
  - Input or output of variables or data
Consider the very simple example of making toast

Process flows from Start to the End through the process and conditional blocks

- Arrows indicate flow
- Conditional blocks control flow branching

Note the loop defining the waiting process

**Wait** block is unnecessary
Flowchart – Example

- Flowchart for a given procedure is not unique
  - Varying levels of complexity and detail are always possible
- Often important to think about and account for various possible outcomes and cases
  - For example, is your toast always done after it first pops up?
  - Here, part of the procedure is repeated if necessary
Taking this example further, consider the possibility of burnt toast or the desire for butter

- Another loop added for continued scraping until edible
- Also possible to bypass portions of the procedure – e.g., the scraping of the toast or the application of butter

- Can imagine significantly more complex flow chart for the same simple procedure ...
Common Flowchart Structures
Common Flowchart Structures

- Several basic structures occur frequently in many different types of flowcharts
  - Recurrent basic structures in many algorithms

- Ultimately translate to recurrent code structures

- Two primary categories
  - Conditional statements
  - Loops

- In this section of notes, we’ll gain an understanding of flowchart structures that fall into these two categories

- In the next section of notes we’ll learn how to implement these structures in code
Conditional Statements

- if statements
- Logical and relational operators
- if...else statements
Conditional Statements – *if*

- Flowcharts represent a set of instructions
  - Blocks and block structures can be thought of as **statements**

- Simplest **conditional statement** is a single **conditional block**
  - An *if structure*
  - If X is true, then do Y, if not, don’t do Y
  - In either case, then proceed to do Z
  - Y and Z could be any type of process or action
    - E.g. add two numbers, turn on a motor, butter the toast, etc.
  - X is a **logical expression** or **Boolean expression**
    - Evaluates to either true (1) or false (0)
Conditional Statements – *if* ... *else*

- Can instead specify an action to perform if X is not true
  - An *if* ... *else* structure
  - If X is true, then do A, else do B
  - Then, move on to do C
- Here, a different process is performed depending on the value of X (1/0, T/F, Y/N)
Conditional Statements – *if ... else*

- Logical expression with a single *relational operator*
  
  \[ x > 9 \]
  
  - Either *true* (Y) or *false* (N)
  - If true, \( x = 1 \)
  - If false, \( x = -1 \)

- Logical expression may also include a *logical operator*
  
  \[(x > 9) || (x < -9)\]
  
  - Again, statement is either *true* or *false*
  - Next process step dependent on value of the conditional logical expression
Logical or Relational Expressions

- Logical expressions use **logical** and **relational operators**

<table>
<thead>
<tr>
<th>Operator</th>
<th>Relationship or Logical Operation</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>Equal to</td>
<td>x == b</td>
</tr>
<tr>
<td>~=</td>
<td>Not equal to</td>
<td>k ~= 0</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less than</td>
<td>t &lt; 12</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater than</td>
<td>a &gt; -5</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less than or equal to</td>
<td>7 &lt;= f</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater than or equal to</td>
<td>(4+r/6) &gt;= 2</td>
</tr>
<tr>
<td>~</td>
<td>NOT– negates the logical value of an expression</td>
<td>~(b &lt; 4*g)</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>AND – <strong>both</strong> expressions must evaluate to true for result to be true</td>
<td>(t &gt; 0) &amp;&amp; (c == 5)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Let $x = 12$ and $y = -3$

Consider the following logical expressions:

<table>
<thead>
<tr>
<th>Logical Expression</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(x + y) == 15$</td>
<td>0</td>
</tr>
<tr>
<td>$(y == 2)</td>
<td></td>
</tr>
<tr>
<td>$\sim (y &lt; 0)$</td>
<td>0</td>
</tr>
<tr>
<td>$(y/2 + 1 &lt; -1)$</td>
<td>0</td>
</tr>
<tr>
<td>$(x == 12) &amp;&amp; \sim (y \geq 5)$</td>
<td>1</td>
</tr>
<tr>
<td>$(y \sim = 2)</td>
<td></td>
</tr>
<tr>
<td>$(x == 2) &amp;&amp; (y &lt; 0)</td>
<td></td>
</tr>
</tbody>
</table>
Conditional Statements – *if ... elseif ... else*

- Two conditional logical expressions
  - If the X is true, do A
  - If X is false, evaluate Y
    - If Y is true, do B
    - If Y is false, do C

- The *if ... elseif ... else* structure

- Can include an arbitrary number of *elseif* statements
  - Successive logical statements evaluated only if preceding statement is false
Consider a **piecewise linear function** of \( x \)

- \( y = f(x) \) not defined by a single function
- Function depends on the value of \( x \)
- Can implement with an **if ... elseif ... else** structure
In previous examples, successive logical statements only evaluated if preceding statement is false.

Result of a true logical expression can also be the evaluation of a second logical expression.
Loops

- while loops
- for loops
Loops

- We’ve already seen some examples of flow charts that contain loops:

- Structures where the algorithmic flow loops back and repeats process steps
  - Repeats as long as a certain condition is met, e.g., toaster has not popped up, toast is inedible, etc.
Loops

- Algorithms employ two primary types of loops:
  - **while loops**: loops that execute as long as a specified condition is met – loop executes as many times as is necessary
  - **for loops**: loops that execute a specified exact number of times

- Similar looking flowchart structures
  - for loop can be thought of as a special case of a while loop
  - However, the distinction between the two is very important
while Loop
while Loop

- Repeatedly execute an instruction or set of instructions as long as (while) a certain condition is met (is true)

- Repeat A while X is true
  - As soon as X is no longer true, break out of the loop and continue on to B
  - A may never execute
  - A may execute only once
  - A may execute forever – an infinite loop
    - If A never causes X to be false
    - Usually not intentional
**while Loop**

- Algorithm loops while $x \leq 4$
  - Loops three times:

<table>
<thead>
<tr>
<th>Iteration</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
</tbody>
</table>

- Value of $x$ exceeds 4 several times during execution
  - $x$ value checked at the beginning of the loop
- Final value of $x$ is greater than 4
while Loop – Infinite Loop

- Now looping continues as long as $x < 12$
  - $x$ never exceeds 12
  - Loops forever – an infinite loop

<table>
<thead>
<tr>
<th>Iteration</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
</tr>
<tr>
<td></td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>4.5</td>
</tr>
<tr>
<td>4</td>
<td>9.5</td>
</tr>
<tr>
<td></td>
<td>4.75</td>
</tr>
<tr>
<td>5</td>
<td>9.75</td>
</tr>
<tr>
<td></td>
<td>4.875</td>
</tr>
<tr>
<td>6</td>
<td>9.875</td>
</tr>
<tr>
<td></td>
<td>4.9375</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Occasionally infinite loops are desirable

Consider for example microcontroller code for an environmental monitoring system

- Continuously takes measurements and displays results while powered on

Note the logical statement in the conditional block

- Logical statements are either true (Y, 1) or false (N, 0)
- 1 is the Boolean representation of true or Y
Consider the following algorithm:

- Read in a number (e.g. user input, from a file, etc.)
- Determine the number of times that number can be successively divided by 2 before the result is ≤ 1

Use a **while loop**

- Divide by 2 **while** number is > 1
while Loop – Example 1

- Number of loop iterations depends on value of the input variable, x
  - Characteristic of while loops
    - # of iterations unknown a priori
  - If $x \leq 1$ loop instructions never execute

- Note the data I/O blocks
  - Typical – many algorithms have **inputs** and **outputs**
Consider a few different input, x, values:

<table>
<thead>
<tr>
<th>count</th>
<th>x</th>
<th>x</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>5</td>
<td>16</td>
<td>0.8</td>
</tr>
<tr>
<td>1</td>
<td>2.5</td>
<td>8</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>1.25</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>0.625</td>
<td>2</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Next, consider an algorithm to calculate $x!$, the **factorial** of $x$:

- Read in a number, $x$
- Compute the product of all integers between 1 and $x$
- Initialize result, fact, to 1
- Multiply fact by $x$
- Decrement $x$ by 1

Use a **while loop**

- Multiply fact by $x$, then decrement $x$ **while** $x > 1$
Consider a few different input, $x$, values:

<table>
<thead>
<tr>
<th>$x$</th>
<th>fact</th>
<th>$x$</th>
<th>fact</th>
<th>$x$</th>
<th>fact</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>1</td>
<td>4</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>20</td>
<td>3</td>
<td>12</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>60</td>
<td>2</td>
<td>24</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>120</td>
<td>1</td>
<td>24</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>1</td>
<td>120</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Let’s say we want to define our factorial algorithm only for integer arguments.

Add **error checking** to the algorithm:

- After reading in a value for \( x \), check if it is an integer.
- If not, generate an error message and exit.
- Could also imagine rounding \( x \), generating a **warning** message and continuing.
for Loop
for Loop

- We’ve seen that the number of while loop iterations is not known ahead of time
  - May depend on inputs, for example
- Sometimes we want a loop to execute an exact, specified number of times

A *for loop*

- Utilize a *loop counter*
- Increment (or decrement) the counter on each iteration
- Loop until the counter reaches a certain value

- Can be thought of as a while loop with the addition of a loop counter
  - But, a very distinct entity when implemented in code
for Loop

- Initialize the loop counter
  - i, j, k are common, but name does not matter
- Set the range for i
  - Not necessary to define variable istop
- Execute loop instructions, A
- Increment loop counter, i
- Repeat until loop counter reaches its stopping value
- Continue on to B
for Loop

- for loops are **counted loops**
- Number of loop iterations is known and is constant
  - Here loop executes 10 times
- Stopping value not necessarily hard-coded
  - Could depend on an input or vector size, etc.
for Loop

- Loop counter may start at value other than 1
- Increment size may be a value other than 1
- Loop counter may count backwards

<table>
<thead>
<tr>
<th>Iteration</th>
<th>cntr</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>6</td>
<td>A</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>A</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>A</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>A</td>
</tr>
<tr>
<td>5</td>
<td>-2</td>
<td>A</td>
</tr>
<tr>
<td>6</td>
<td>-4</td>
<td>B</td>
</tr>
</tbody>
</table>
for Loop – Example 1

- Here, the loop counter, i, is used to update a variable, x, on each iteration

<table>
<thead>
<tr>
<th>Iteration</th>
<th>i</th>
<th>x</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>16</td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>25</td>
</tr>
</tbody>
</table>

- When loop terminates, and flow proceeds to the next process step, x = 25
  - A scalar
  - No record of previous values of x
Now, modify the loop process to store values of $x$ as a **vector**

- Use loop counter to index the vector

<table>
<thead>
<tr>
<th>$i$</th>
<th>$x(i)$</th>
<th>$x$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>[1]</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
<td>[1, 4]</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>[1, 4, 9]</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>[1, 4, 9, 16]</td>
</tr>
<tr>
<td>5</td>
<td>25</td>
<td>[1, 4, 9, 16, 25]</td>
</tr>
</tbody>
</table>

When loop terminates, $x = [1, 4, 9, 16, 25]$

- A **vector**
- $x$ grows with each iteration
for Loop – Example 3

- The loop counter does not need to be used within the loop
  - Used as a counter *only*
- Here, a random number is generated and displayed each of the 10 times through the loop
  - Counter, i, has nothing to do with the values of the random numbers displayed
for Loop – Example 4

- Have a vector of values, x
- Find the **mean** of those values
  - Sum all values in x
    - A for loop
    - # of iterations equal to the length of x
    - Loop counter indexes x
  - Divide the sum by the number of elements in x
    - After exiting the loop
Nested Loops
Nested Loops

- A loop repeats some process some number of times
  - The repeated process can, itself, be a loop
  - A nested loop

- Can have nested for loops or while loops
  - Can nest for loops within while loops and vice versa

- One application of a nested for loop is to step through every element in a matrix
  - Loop counter variables used as matrix indices
  - Outer loop steps through rows (or columns)
  - Inner loop steps through columns (or rows)
Recall how we index the elements within a matrix:

- $A_{ij}$ is the element on the $i^{th}$ row and $j^{th}$ column of the matrix $A$
- Using MATLAB syntax: $A(i,j)$

Consider a $3 \times 2$ matrix

$$B = \begin{bmatrix} -2 & 1 \\ 0 & 8 \\ 7 & -3 \end{bmatrix}$$

To access every element in $B$:

- start on the first row and increment through all columns
- Increment to the second row and increment through all columns
- Continue through all rows
- Two nested for loops
Nested for Loop – Example

\[ B = \begin{bmatrix} -2 & 1 \\ 0 & 8 \\ 7 & -3 \end{bmatrix} \]

- Generate a matrix \( C \) whose entries are the squares of all of the elements in \( B \)

- **Nested for loop**
  - Outer loop steps through rows
    - Counter is row index
  - Inner loop steps through columns
    - Counter is column index
Pseudocode & Top-Down Design
Flowcharts provide a useful tool for designing algorithms
- Allow for describing algorithmic structure
- Ultimately used for generation of code
- Details neglected in favor of concise structural and functional description

Pseudocode provides a similar tool
- One step closer to actual code
- Textual description of an algorithm
- Natural language mixed with language-specific syntax
Consider an algorithm for determining the maximum of a vector of values

Pseudocode might look like:

```
N = length of x
max_x = x(1)
for i = 2:N
    if x(i) is greater than current max_x, then set max_x = x(i)
end
```

Note the for loop syntax

- We’ll cover this in the following section of notes
Flowcharts and pseudocode are useful tools for *top-down design*

- A good approach to any complex engineering design (and writing, as well)
- First, define the overall system or algorithm at the top level (perhaps as a flowchart)
- Then, fill in the details of individual functional blocks

Top-level flowchart identifies individual functional blocks and shows how each fits into the algorithm

- Each functional block may comprise its own flow chart or even multiple levels of flow charts

*Hierarchical design*
Top-Down Design - Example

- Let’s say you have deflection data from FEM analysis of a truss design
  - Data stored in text files
    - Deflection vs. location along truss
  - Parametric study
    - Three different component thicknesses
    - Two different materials
    - Six data sets
- Read in the data, calculate the max deflection and plot the deflection vs. position
Top-Down Design - Example

**Level 1:**
- Start
- Read in the data
- Calculate the maximum displacement
- Plot the data
- End

**Level 2:**
- Read data for material A, thickness 1
- Read data for material A, thickness 2
- Read data for material A, thickness 3
- Read data for material B, thickness 1
- Read data for material B, thickness 2
- Read data for material B, thickness 3

**Level 3:**
- Open file for read
- N = # of data point in the file
- i = 1
- i ≤ N
- Read in row i
  - Store values as x(i) and d(i)
- i = i + 1