Here we will get some practice writing our own Matlab functions. We'll pick a simple task, then we'll try to accomplish that task in different ways, depending on which Matlab internal functions we allow ourselves to use.

One very basic internal Matlab function is \( \text{max}(x) \), which takes a vector \( x \) and finds the largest value in it. If we wanted to write our own function to do this, one possible way to do this is to loop through all the values in the vector, keeping track at all times of the largest value that has been considered up to that time. In pseudocode, this might look like:

1) Input the vector \( x \).
2) Set \( N \) equal to the length of \( x \).
3) Set \( \text{track} \) equal to \( x(1) \) (the \( \text{track} \) variable will represent the largest value encountered so far).
4) Set \( i \) equal to 2.
5) Compare \( x(i) \) to \( \text{track} \). If \( x(i) \) is larger, then set \( \text{track} \) equal to \( x(i) \).
6) If, instead, \( x(i) \) is smaller than \( \text{track} \), then leave \( \text{track} \) unchanged.
7) Add one to \( i \).
8) If \( i \) is greater than \( N \), then you’ve gone all the way through the vector, and \( \text{track} \) is the maximum value in the vector.
9) Otherwise, go back to step 5 and repeat.

We will try to write our own functions to perform a slightly more complicated task. Specifically, we’re going to write functions to find the second-highest value in a vector. We’ll assume for simplicity that no two \( x(i) \) values are the same, i.e. \( x(i) \neq x(j) \) if \( i \neq j \), and also that there are at least three values in the vector \( x \).

**Function almostmax1.** We’ll first try to write this function without using any Matlab vector operations or functions (for example, we won’t use \( \text{max} \)) except to determine the length of the vector. How might this function, which we’ll call \( \text{almostmax1} \), proceed? We can borrow some ideas from the pseudocode above that we wrote for the \( \text{max} \) function. Specifically, we can loop through the vector while keeping track of two variables, one (called \( \text{track1} \)) that represents the maximum value encountered up to a certain point, and a second (called \( \text{track2} \)) that represents the second-highest value up to that point. Any time we consider a new \( x(i) \) in the loop, we compare it to \( \text{track1} \) and \( \text{track2} \). If \( x(i) \) is bigger than \( \text{track1} \), then we set \( \text{track2} \) equal to the old \( \text{track1} \), and set \( \text{track1} \) equal to \( x(i) \); if \( x(i) \) is between \( \text{track1} \) and \( \text{track2} \), then we set \( \text{track2} \) equal to \( x(i) \); and if \( x(i) \) is less than \( \text{track2} \), then we don’t do anything.

The flowchart for the \( \text{almostmax1} \) function is shown in Fig. 1. The top part of the flowchart initializes the \( \text{track1} \) and \( \text{track2} \) variables, by comparing the first two values in the vector, \( x(1) \) and \( x(2) \). The lower part loops through the rest of the vector, and compares each \( x(i) \) with the two tracking variables.
In order to write Matlab code, we need to be able to look at a flowchart (or at pseudocode) and identify where different structures like if-statements or for-loops would be. In the \texttt{almostmax1} flowchart, the top decision diamond, comparing \texttt{x(1)} and \texttt{x(2)}, is clearly a simple if-statement. If the condition in the decision diamond is true, you go one way, if it's false, you go the other. It is also clear that there is a for-loop in the lower part of the flowchart, where we loop through \texttt{i} values from 3 to \texttt{N}. The decision diamonds inside that loop are interesting. If the condition in the first decision diamond is true, then you execute a set of commands; if it's false, then you check the condition in the second decision diamond; then if \textit{that} condition is true, you go one way, and if it's false, you go another. This logic, where you check one condition, then you check a second one if the first one is false, is exactly represented using an if-elseif-statement.

Now let's write up the function \texttt{almostmax1} in Matlab language (these commands should be saved in a file called \texttt{almostmax1.m}):

1. \texttt{function A = almostmax1(x)}
2. % This program finds the second-highest value in the vector \texttt{x}
3. \texttt{N = length(x);} 
4. \texttt{if x(1) > x(2)}
5. \texttt{track1 = x(1);} 
6. \texttt{track2 = x(2);} 
7. \texttt{else}
8. \texttt{track1 = x(2);} 
9. \texttt{track2 = x(1);} 
10. \texttt{end}
11. \texttt{for i = 3:N}
12. \texttt{if x(i) > track1}
13. \texttt{track2 = track1;}
14. \texttt{track1 = x(i);} 
15. \texttt{elseif x(i) > track2}
16. \texttt{track2 = x(i);} 
17. \texttt{end}
18. \texttt{end}
19. \texttt{A = track2;}

To see how this function works, consider the following sequence of commands entered into the main Matlab window:

\begin{verbatim}
>> y = [1 4 5 2 -1 6 8];
>> B = almostmax1(y);
\end{verbatim}
As we’ve discussed previously, the names of variables inside the functions is irrelevant. For example, even though line 1 of the function names the output as \( A \) and the input as \( x \), you don’t need to use these names when you call the function from the command window (or, for that matter, from another Matlab program or function). All that matters is that you provide the function with an input vector, and that you recognize that the output is a scalar value.

**Function almostmax2.** Now let’s imagine that in the process of writing a function to find the second-highest value in a vector, we’re allowed to use the `max` function. We’ll name this new function `almostmax2`. By using the `max` function, we will no longer need to keep track of two variables, representing the two current highest values, as we did with `almostmax1`. Instead, we can just keep track of the current highest value that isn’t equal to the maximum value.

The flowchart of `almostmax2` is given in Fig. 2. The variable `track` keeps track of the current high value that isn’t equal to the maximum value of the vector. The first if-statement in the flowchart initializes the `track` variable, by finding the first term in \( x \) that’s not equal to `max(x)`. As in `almostmax1`, there is a for-loop in the lower part of the flowchart, which allows us to consider the values in \( x \) in order. For the two decision diamonds in the loop, a command is executed if both conditions are true, but nothing happens otherwise. This can be expressed two ways; either as an if-statement nested inside another if-statement, or as a single if-statement where the logical expression is composed of two individual expressions connected by the ‘and’ logical operator. Using the ‘and’ operator results in more compact Matlab code (just one ‘if’ line and one ‘end’ line, as opposed to two of each for nested in-statements.

The Matlab code for the function `almostmax2` (to be saved as `almostmax2.m`) is:

1. function \( A = \text{almostmax2}(x) \)
2. \% Finds the second-highest value in the vector \( x \); includes the ‘max’ function
3. \( N = \text{length}(x); \)
4. \( \text{xmax} = \text{max}(x); \)
5. if \( x(1) \sim \text{xmax} \)
6. \( \text{track} = x(1); \)
7. else
8. \( \text{track} = x(2); \)
9. end
10. for \( i = 2:N \)
11. \( \text{if} \, x(i) \sim \text{xmax} \& x(i) > \text{track} \)
12. \( \text{track} = x(i); \)
13. end
14. end
15. \( A = \text{track}; \)

Line 11 shows how the logical ‘and’ operator is used. This function is used in the same way as \texttt{almostmax1}, the use of which was demonstrated above.

**Function mysort.** Comparing the functions \texttt{almostmax1} and \texttt{almostmax2}, it is clear that we get a shorter and simpler Matlab function when we are allowed to use the \texttt{max} function in determining the second-highest value. We can cut the number of lines of code even more drastically by making use of Matlab’s \texttt{sort} function. The \texttt{sort} function takes in a vector, and sorts its values in ascending order. Before we get around to incorporating \texttt{sort} into another function for determining the second-highest value, let’s write a function \texttt{mysort} that does the same thing as the \texttt{sort} function, without using any vector operations.

There are many, many ways to write a sorting algorithm. We’ll proceed in a fairly simple (not necessarily the fastest) way. First, we find the maximum value in the vector, and swap it with the last term in the vector, \( x(N) \). Next, we find the maximum value of the other \( N-1 \) values in the vector, and swap it with value in the \( N-1 \) spot, \( x(N-1) \). We repeat this operation, considering one fewer vector value each time. By the time we get down to making sure that \( x(2) > x(1) \), the rest of the vector will be arranged in ascending order.

The flowchart for \texttt{mysort} is shown in Fig. 3. There are two for-loops in this function. The first loop is defined by \( j \), which starts at \( N \), and counts down by one each time, until it reaches \( j=2 \). This is the loop that controls how many values in the vector are under consideration. Inside the loop on \( j \), there is another loop, defined by \( i \), that indicates which term in the vector you’re considering at any time. The variable \texttt{track} is the current highest value considered, and \texttt{tracki} is the location of that value. After the loop on \( i \) is finished, the function swaps the highest value with the value at \( x(j) \).

The \texttt{mysort} code (saved as \texttt{mysort.m}) looks like:

1. \texttt{function y = mysort(x)}
2. \% Emulates the Matlab internal function ‘sort’
3. \texttt{N = length(x);}
4. \texttt{for j = N:-1:2}
5. \texttt{track = x(1);}
6. \texttt{tracki = 1;}
7. \texttt{for i = 2:j}
8. \texttt{if x(i) > track}
9. \texttt{track = x(i);}
10. \texttt{tracki = i;}
11. \texttt{end}
12. \texttt{end}
13. \texttt{x(tracki) = x(j);}
14. \texttt{x(j) = track;}
15. \texttt{end}
16. \( y = x; \)

The function is applied in Matlab as shown:

```matlab
>> y = [-10 8 -4 3 12 -15];
>> x = mysort(y);
>> x
```

```
x =
-15  -10  -4   3   8  12
```

**Function almostmax3.** Now let's write the function `almostmax3`, which does the same thing as `almostmax1` and `almostmax2`, but allowing the use of the `mysort` function. This will be so easy, it would be completely unnecessary to write pseudocode or construct a flowchart. All we have to do is read in the input vector, \( x \), sort it, then take the value in the second-to-last spot. The Matlab code is just (saved as `almostmax3.m`):

1. ```matlab
   function A = almostmax3(x)
   % Finds the second-highest value in a vector, using the 'mysort' function
   N = length(x);
   y = mysort(x);
   A = y(N-1);
   ```

So, it is clear that the `mysort` function results in a much more compact function than either of the versions we wrote before.
Figure 1: Flowchart for the almostmax1 function.
Figure 2: Flowchart for the \texttt{almostmax2} function.
Figure 3: Flowchart for the `mysort` function.